National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

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

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

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

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

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

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

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

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

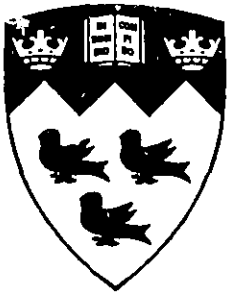
SPECIFICATION DRIVEN ARCHITECTURAL MODELLING ENVIRONMENT FOR TELECOMMUNICATION SYSTEMS SYNTHESIS

ORYAL TANIR

Department of Electrical Engineering

McGill University, Montreal

October 1994



A Thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfilment of the requirements of the degree of

Doctor of Philosophy

ISBN 0-612-00138-5

SPECIFICATION DRIVEN ARCHITECTURAL MODELLING ENVIRONMENT FOR TELECOMMUNICATION SYSTEMS SYNTHESIS

Oryal Tanir
Ph.D. Thesis May 1994
Department of Electrical Engineering

SHORT TITLE:

Architectural modelling environment for telecommunication systems

Abstract

Design automation has steadily contributed to improvements witnessed in the system design process. Initial applications were to address low level design concerns such as transistor layout and simulation; however the focus of tools has slowly been progressing up the design abstraction scale. The current state-of-the-art provides modelling capabilities at different levels of abstraction, but solutions for synthesis issues at the register-transfer and lower levels are the norm. The proliferation of design description languages at different abstraction levels has prompted the need for standardization (VHDL and Open-Verilog) to promote design migration and re-use.

While design automation has helped in reducing design time-lines and design churn, a major source of design difficulties is just recently being addressed and promise to be the next wave in design automation applicability. The problems arise within the architectural (or system) level of abstraction very early in the design cycle. The recent research in this field attempts to bridge the design process gap between specification and design, and provides a platform for experimenting with hardware and software trade-offs.

This dissertation studies the requirements for an environment for architectural design. In particular, an environment specific to the telecommunications domain is proposed in order to limit the potentially large design exploration space. An intermediate design language is also introduced to accommodate both high level modelling and synthesis driven by the user and environment. Finally a Design Analysis and Synthesis Environment (DASE) is described to facilitate the architectural level activities. The environment, a proof of concept, provides generic model library, simulation, synthesis and Petri-net analysis support. Realistic design examples are explored, to illustrate architectural design activities with the environment.

Résumé

Les techniques d'automatisation pour la conception des systèmes digitaux jouent un rôle important dans l'avancement du processus de développement de ces systèmes. Les premiers outils automatiques apparus s'adressaient surtout aux problèmes physiques et géométriques de la conception, c'est-à-dire les questions de bas niveau d'abstraction comme la simulation et l'emplacement des transistors. Cependant l'évolution des outils se marque par une progression du niveau d'abstraction employé par ceux-ci. Présentement les outils les plus sophistiqués offrent un éventail de niveaux d'abstraction variant des transistors jusqu'à la modélisation architecturale en passant par les abstractions au niveau des portes logiques, des transferts entre registres et des algorithmes. Non pas comme les outils de modélisation qui varient sur toute la gamme des niveaux d'abstraction, la grande partie des outils de synthèse se limite actuellement aux niveaux d'abstraction entre ceux des transistors et des transferts entre registres. L'apparition d'une panoplie de langages de spécification sur plusieurs niveaux d'abstraction souligne le besoin d'une standardisation (VHDL et l'Open-Verilog) pour faciliter la réutilisation et la migration des conceptions.

Même si l'automatisation de la conception a déjà produit des résultats pour la réduction de l'intervalle temporel requis pour la conception ainsi que la simplification des étapes itératives nécessitées par les rajustements, une source importante de difficultés retrouvées en conception n'a été adressée que récemment et les solutions sont prometteuses pour la prochaine génération des outils pratiques en conception automatisée. Les problèmes surviennent tôt dans le cycle de conception au niveau d'abstraction architectural. Les recherches récentes en ce domaine tente de faire le lien entre la spécification et la conception tout en permettant une expérimentation sur les conséquences de la division d'un système en une partie logiciel et une partie matérielle.

Cette thèse étudie les exigences d'un environnement de conception au niveau architectural. Plus particulièrement, un environnement spécifique aux systèmes de télécommunications est introduit pour amoindrir les espaces de solutions vastes qui sont reliés au problème plus général. De plus un langage intermédiaire est présenté pour la modélisation et la synthèse exercées par l'utilisateur et son environnement. Finalement, un environnement d'analyse et de synthèse (DASE) est exposé pour faciliter les activités au niveau architectural. Cet environment est une preuve du concept émis dans la thèse et il soutient une librairie de modèles génériques, la simulation, la synthèse et l'analyse par réseau de Petri. Des exemples réalistes sont présentés pour démontrer l'efficacité de l'environnement au niveau architectural.

Acknowledgments

This dissertation would not have been possible without the support and guidance of many different individuals. First, I wish to thank and acknowledge my academic advisors Vinod K. Agarwal and Pramod C. P. Bhatt for their invaluable contribution to this work and my personal development. Their foresight and consistent enthusiasm has been a constant inspiration. Through numerous interactions, Prof. Agarwal has helped forge my reasoning and thinking processes that are essential in tackling current and future problems that I may encounter. Prof. Bhatt has been a relentless, organized and pragmatic mentor who has deeply influenced the way in which I now perceive problems.

I would also like to thank Eric Masson from the MACS laboratory for his help in the french translation of the abstract, as well as numerous information sessions which contributed to the development of my work.

On another level, none the less important, I would like to thank my parents for their love, support and sacrifices that have led me to the point I am at now. I am also indebted to my wife, who has contributed to my development as a person over the last few years. Her constant understanding and love has been a beacon of light to my work and will guide me brightly into the future. I would also like to thank someone very special, my son Dilhan, who's recent appearance to this world has inspired my work.

This work has been generously supported and made possible with the understanding of various people at Bell Canada. I would like to thank Francois Coallier's encouragement, and I would also like to thank Jim Holz and Martine Corriveau-Gougeon for their support and patience during the thesis work.

Table of Contents

List of Figures

List of Tables

Claim of Originality

The author claims originality for the following contributions of the dissertation.

- In chapters 1 and 3. The proposed environment for design automation of telecommunication systems (DASE) is original as it integrates modelling, analysis and synthesis at the architectural level of abstraction within one seamless framework. All components of DASE have been implemented as a proof of concept. These include: the DSL processor, simulator and synthesizer (DSL-VHDL translator).

- In chapter 2. The intermediate language DSL is an original contribution to architectural modelling. It's capability to permit definition of constructs that can evolve with additional design detail, permitting both abstract representation and synthesis, is novel.

- In chapter 2. The translation rules and algorithm for DSL modules to predicate/transition nets is novel.

- In chapter 2. The representation of software and hardware entities as an indistinguishable module is original. This permits the modelling of such entities before design partitioning.

- In chapter 4 and 2.4. The case studies are extracted from real world problems. In that sense they are not unique, however their representation as DSL modules is novel. Included is a generic module construction example (DTSS) providing substantial model re-use.

- In chapter 3. The library support description and implementation is original. In particular, the ability of library modules to configure their resources and interfaces to a given model environment permits generic models to be managed by the library.

- In chapter 3. The translation algorithms presented for DSL to VHDL (both mode 1 and 2) are original contributions.

Chapter 1 - Introduction

The telecommunication network is the largest manmade machine in the world - composed of an elaborate blend of hardware and software elements. As with other high technology driven fields, the market pressures force the telecommunication industry to produce higher quality products in shorter times. However, the increased complexity of telecommunication systems has made them very difficult to verify. Rapid system prototyping aids can automate different aspect of the design process and help address these concerns. This dissertation presents a rapid prototyping environment to help designers describe, model and explore design tradeoffs at the more abstract architectural level of representation and synthesize in the domain of telecommunication systems. In contrast, traditional environments have focussed on design support at abstractions that are at the register-transfer level or lower.

This chapter is divided into four sections. The first section is a perspective on the product design cycle for telecommunication systems. The section creates a premise for the use of rapid system prototyping and in particular, potential benefits of use early in the design cycle. In the second section an overview of current research in the rapid system prototyping field is presented. A need is defined for an internal architectural level modeling language which possesses synthesis constructs to permit model refinement to register-transfer level representations. The third section introduces such a language as part of a Design Analysis and Synthesis Environment (DASE), the core of this thesis. The final section concludes with an overview of the remainder of the dissertation.

1.1 Perspective

A typical design flow for large system design is represented in figure 1-1. The figure highlights the high level flow of information from design concept to implementation. Four phases of a product's early life-cycle are also shown: The specification, design, implementation and test phase. The phases do not necessarily have well-defined boundaries - for example specification and design may be considered as one phase within a given engineering group. Listed above each phase are the related design activities that contribute to the overall time to complete a phase. The activities represent a typical development process and may differ in each organization, depending upon their maturity level.

Initial system requirements in the specification phase generate design specifications that are input to the design phase. The design phase encompasses many activities related to modelling, design exploration and synthesis. The modelling activity can be composed of model creation and analysis. A design exploration activity can use the model to simulate different scenarios or apply formal verification methods to obtain an acceptable representation of the system. The designer can also partition the design into the respective hardware and software components and, through the use of synthesis tools, generate a prototype (or product) in the implementation phase. For hardware designers, description

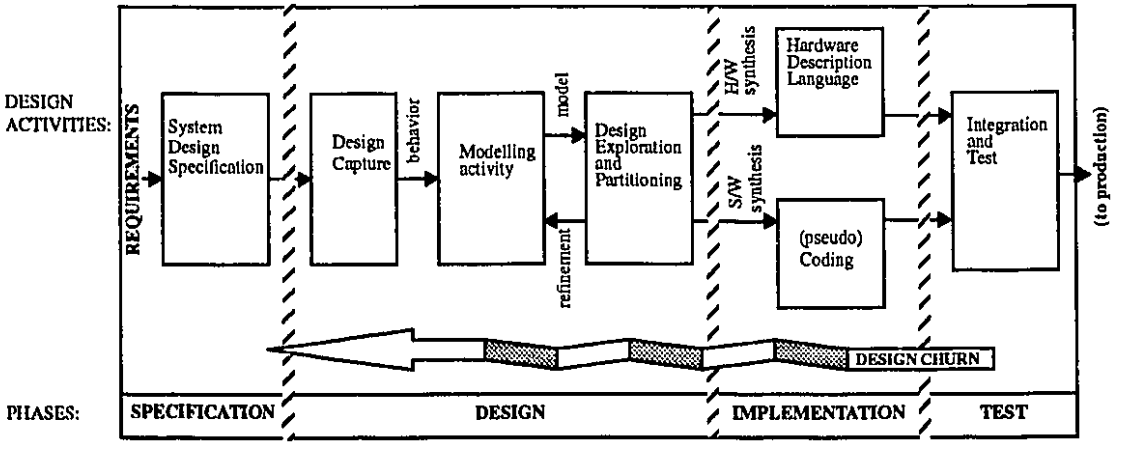


FIGURE 1-1. Typical Design Flow for Large Telecommunication Systems

languages exist which can represent systems at various level of abstraction. The proliferation of these languages has prompted the need for standardization (IEEE 1076 VHDL and Verilog) [VHDL 87] [Verilog 91] to promote design migration and re-use. Current commercial design environments permit synthesis of hardware described in a subset of VHDL or Verilog, where the initial design is at a register transfer level of abstraction. The software developers may utilize methodologies, such as Shlaer/Mellor [Shlaer 88] or Ward/Mellor [Ward 85], to describe the software constructs in more detail. Each phase in the product lifecycle can potentially flow back to the previous phase(s) indicating potential design revisions. This creates design churn and adds to the overall time to deliver a product.

Intuitively, the time period to create a product (from requirements to test) can be influenced with integrated tools to shorten various phases. The tools can either reduce the duration of activities (such as modeling) within phases or reduce the design churn time. Traditional design approaches have focussed upon such tools. Integrated across several design activities, the tools can form a Rapid System Prototyping (RSP) environment to quickly produce a prototype system - which is a scaled down version of the final product. The prototype is a model of a conceptual system to be fabricated and relies upon the environment to expeditiously produce sound systems.

Although automation is one way to affect design times, further motivation exists to apply this as early as possible in the product life-cycle. The progression of tool capabilities extending toward initial specifications promises to reduce design ambiguities and errors early in the product life-cycle. Such tools must be capable of design capture at increased levels of abstraction to support modelling at the architectural design level. The architectural level is the most abstract design level defined in [Bell 71] where system elements are viewed as communicating processes.

The desire to move tools up the abstraction scale is attractive for several reasons. It is well accepted that design errors uncovered early in the life-cycle are orders of magnitude less expensive to fix than those detected downstream [Boehm 81]. Early error detection can be

a significant benefit of architectural level automation. Figure 1-2 further emphasizes the impact of design choices made in the early phases. The figure depicts product development in terms of a project within a corporation [Saultz 92] and illustrates the typical resource and personnel effort requirements during a product life-cycle. The horizontal axis depicts the different phases that the product may progress through, whilst the vertical axis indicates a percentage total cost of the project in terms of total effort and resources (equipment, personnel, money). The diagram demonstrates that more than half of the total cost is already committed to the project after the specification phase whereas only 15% of the cost is incurred. It should also be noted that resources freed from previously completed phases may be allocated to other projects. This makes them difficult to access for potential design reworks. Before reaching the production phase, nearly 90% of all the total cost is already committed. As a result, significant design errors detected downstream can negatively impact the resource allocation to the project resulting in cost overruns and delays. Hence the resolution of design ambiguities at this early stage ensures that a project reduces potential design reworks and churn so that the product can be developed with the limited resources available. Hence,the point in time where design aids are used in the products' development, can significantly curb development costs [Hayes 88].

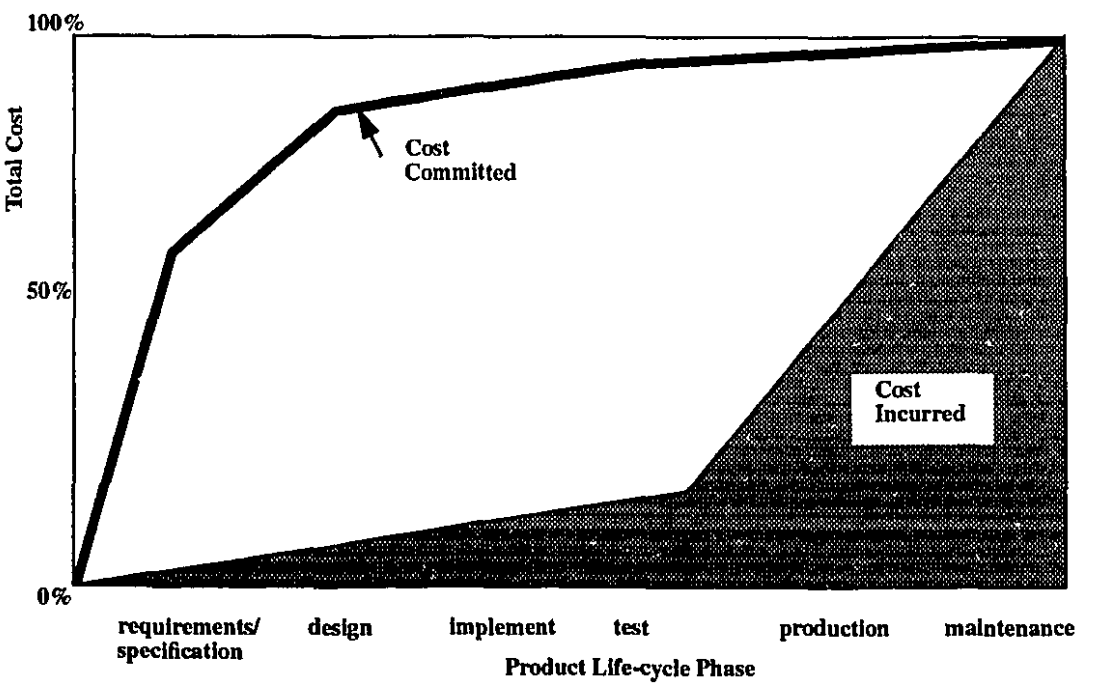


FIGURE 1-2. Typical Project Cost Demands

In summary, an RSP environment applied to architectural level design of large systems can have a significant impact upon the design cycle such as:

i. Shorter design cycle: Generating a quick yet reasonable design from the very conception of the product can weed out poor design decisions from good ones. This reduces expensive design reworks later in the design stage. As a result, the time from system conception to integration is greatly reduced.

ii. Verification of designs: Current designs are verified later in the design stage. At that point, poor architectural decisions can be hidden in the complexity of the complete design details. By synthesis of a system design one obtains a formal representation or model. Algorithms or rules can be applied at this level to test the correctness of the design versus the design specifications. Ideally, synthesis would generate designs that are correct by construction. These lead to less design errors later on in the design cycle.

iii. Solidification of design requirements: Vague or ambiguous design specifications can be identified and corrected during this early stage. Traditionally this has been an area where designers would interpret the specifications. This may not necessarily have been the intent of the specifications [Srivas 90] [Moore 90]. The level of abstraction in the architectural level is closest to the conceptual model a designer would work on, minimizing loss of information.

iv. Procedures for formal design specification: With the advent of system level synthesis tools, formal requirements for design specifications can be defined. Formal specifications following a well-defined structure or methodology is currently lacking in the design community.

v. More design alternatives: If design space exploration can be performed quickly, different design alternatives that may have otherwise not been feasible could also be explored. This helps guide designer in the conceptual creation or evaluation of their designs.

1.2 Rapid system prototyping

RSP environments provide support for at least three main activities: system design, execution and synthesis. The design support is typically through an internal modelling or specification language that can describe a given system at the desired level of abstraction. The language must also be sufficiently versatile to accommodate the other two key activities. Execution of the design entails the exploration of the design space and further refinements to the design - in the case of architectural modelling, this would also involve design partitioning and co-design. This activity may utilize simulation and formal methods as design tools. Finally a synthesis activity uses the refined design to produce the target product. The source and target model for synthesis reflects the level of synthesis being undertaken in the design. Four different design abstraction levels are commonly depicted as concentric circles radiating from the center of a three-dimensional graph [Gajski 92] as shown in figure 1-3. The intersection of these circles with the three axises represents three possible views of design domains (behavioral, structural and physical) at a given level of abstraction. As the circles move away from the center, the level of representation becomes more abstract. Synthesis can typically occur from a given behavioral representation to a structural one.

Prototyping systems have been developed mostly to address synthesis at register-transfer and lower levels. For example, IDEAS [Kumar 89] is an environment that allows RTL synthesis of designs. CATHEDRAL [Lanneer 91], ISPS [Barbaci 81], and HIS [Bergamaschi 93] are examples of high level synthesis tools. As the lower level design tools have matured, research focussed on architectural level synthesis (transforming a system level design specification into an algorithmic level specification of the behavior) is finally starting to become feasible and is drawing more attention. Some success for reasonably complex systems has been obtained with systems dedicated to specific problem domains such as Digital Signal Processing where the design space is relatively homogeneous [Lanneer 91].

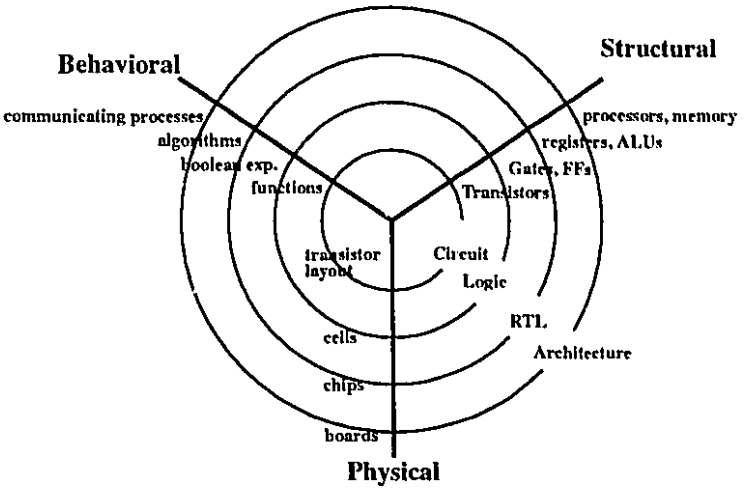


FIGURE 1-3. Different Views of Design

With the increased level of abstraction, the design space for a "general purpose" architectural level RSP tool is heavily heterogeneous making it infeasible to derive a model from most initial specifications [Ramming 93]. As a result, it is generally agreed that successful architectural level RSPs will be domain specific (exhibiting a reasonably homogeneous design space) or support a wide variety of specification paradigms.

The demands upon the modelling capabilities of an RSP environment imply that the internal representation or design language employed by the environment can greatly affect the acceptance of the tool within a given domain of application. At the architectural abstraction, design partitioning and co-design are major activities involving both software and hardware designers. Therefore a design language is needed to accommodate both disciplines transparently. In the hardware community, design languages have evolved from logic gate oriented formalisms to those capable of modelling up to the algorithmic abstraction level. However they rarely support architectural level representation (although attempts are being made to extend VHDL into this level [Jerraya 91]). Most hardware description languages are simulation oriented and interleave simulation and modelling semantics - making them difficult for use in synthesis systems. An example is VHDL - where only subsets of the language are applicable for synthesis by commercial tools.

Within the software design area, the trend has been toward a parallel and concurrent programming view. This has made the use of specification paradigms, that support communicating processes, highly amiable. A similar trend can also be observed within the hardware design community, with their desire to find better ways to specify the existing parallelism inherent in hardware. Hence at the architectural level, where software and hardware design concerns are first addressed, a formalism based upon communicating processes appears natural to designers of both disciplines [Koomen 91]. For example, the Specification and Description Language (SDL) [CCITT 88] is based upon communicating processes and is in use in telecommunication software development groups [Klick 91][Jacobson 92]. Although descriptive, the language is undergoing revisions to adopt object-oriented views and a timing model. Another language is Statecharts [Harel 87] that allows description of systems in terms of hierarchical communicating finite state machines expressed in a graphical notation.

Apart from facilitating co-design, there is an added onus on the architectural design language: the output of the internal representation must also be synthesized to the desired target model. This implies that synthesis constructs be part of the internal representation from the onset, with added modelling support to refine the models. Economic considerations also justify the re-use of components in a modelling framework. Such an environment requires a flexible library support system capable of management of models. Typically, abstract models must be capable of being stored, retrieved, configured and organized hierarchically with object oriented capabilities (such as inheritance) to facilitate model construction and re-use.

1.3 Architectural design and modelling

The Design Specification Language (DSL) introduced within this dissertation, is the internal representation language used within the Design Analysis and Synthesis Environment (DASE). The language addresses the representation issues identified in the

preceding section and provides a platform for architectural modelling of telecommunication systems as well as support for synthesis constructs. The restricted domain permits the use of a model library within DASE to aid in the DSL model refinement through simulation. Although described in detail in the dissertation, DSL is internal to the environment and is not necessarily the language that must be used by a user. A language translator can be employed between an existing specification language and DSL - reducing the need for training the user on a new language. The DSL representation can then be used to facilitate the refinement of the design.

DSL is currently implemented in Prolog. This implies that model behavior can be described as a set of clauses, communicating processes or finite state machines - suitable to represent both hardware and software at the architectural level. Prolog also provides a suitable implementation platform for facilitating model refinements during the development of a DSL model. Refinements are carried out under environment support until the model achieves a state where it can be translated to a corresponding VHDL representation.

The DASE environment is shown in figure 1-4 as part of an architectural design automation framework being developed by the MACS laboratory [Tanir 92]. The implemented environment provides the necessary support for DSL to bridge the gap between specification and synthesis. Designers can conveniently create architectural models in a top-down fashion, incorporating further detail as required through a model library or during design exploration using simulation.

A DSL design is input to the environment through a user interface that is then interpreted by a DSL processor unit. This unit interacts with all other components as well as managing a DSL model library [Tanir 93a]. The model library is organized to allow design exploration within the identified domain of telecommunication system [Tanir 93b] [Tanir 93c] [Tanir 93d]. Models within the library are termed generic and can be re-used and re-configured with a large degree of freedom. During model execution, the initial DSL model may undergo refinements under the guidance of the DSL processor as necessary model

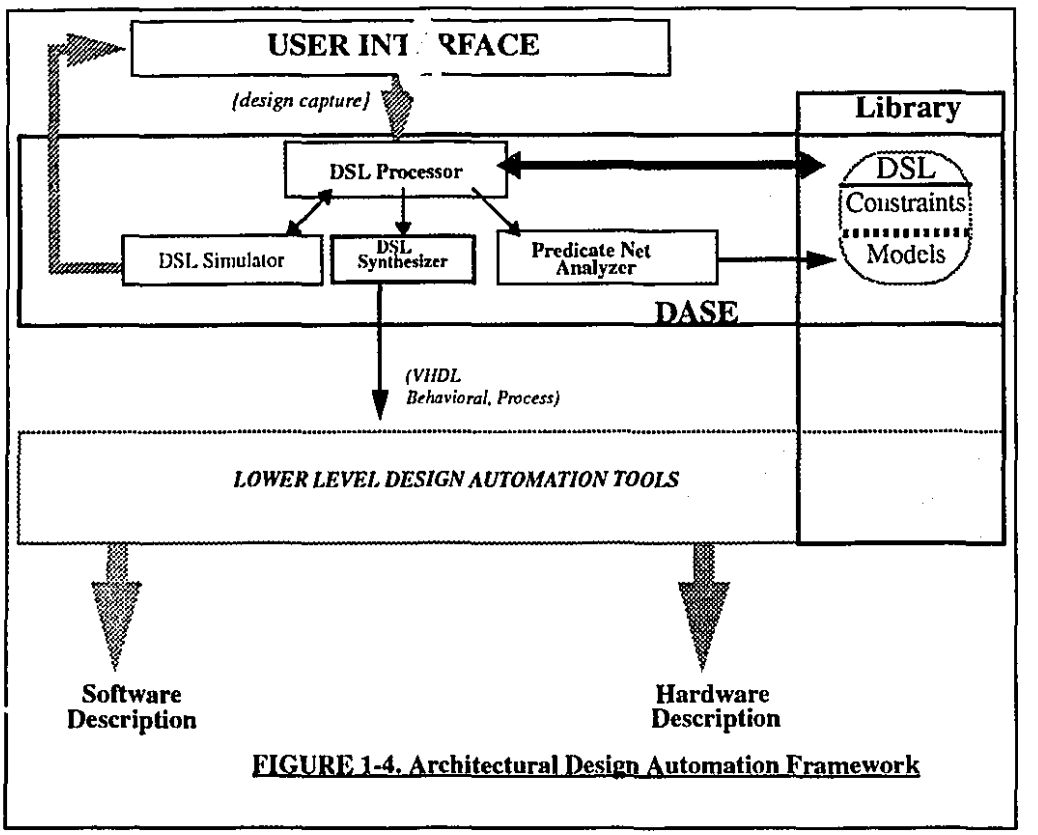


FIGURE 1-4. Architectural Design Automation Framework

details are configured through the library.

An interface is defined with a Predicate Transition Net tool to allow the verification of properties of DSL model components. Design exploration is achieved by using a simulator which provides the timing model for DSL and permits observation of events at different levels of detail. Finally a synthesis component allows the DSL models to be synthesized to an RTL behavioral VHDL model. The VHDL model can then be used by lower level design aids to optimize and eventually synthesize to hardware.

1.4 Dissertation outline

This dissertation presents an architectural rapid prototyping environment for telecommunications systems. It does not claim to resolve all design support issues at this level. However this is a good starting point for constructing a potential environment that

can be amiable to system design.

The dissertation is divided into five chapters. The following chapter will present the DSL language used by the environment. The chapter will also introduce an example of a design of a digital telephone switch using DSL, demonstrating the modelling capabilities of the language. Chapter 3 will describe the environment support capabilities for automation including simulation and synthesis. The switch example will be used to demonstrate the use of the different elements of DASE to impact model development. Chapter 4 will provide two detailed case studies demonstrating the further capabilities and features of the proposed environment.The examples are based upon the design of an ATM switching network and an implementation of a distributed broadcast protocol executing over the ATM network. Finally, chapter 5 will provide conclusions and suggestions for further extensions to this work.

Chapter 2 - The Design Specification Language

Design languages exist for a myriad of different applications. As opposed to register-transfer / circuit level design, the architectural level blurs the distinction between software and hardware, therefore requires elements common to languages applied in both fields. Furthermore, the design language must be able to facilitate the three RSP activities introduced in chapter 1; systems design, execution and synthesis.

To emphasize the architectural RSP needs for a design language, the first section of this chapter will address language issues for each of the three RSP activities and their implications on architectural design. An overview of the applicable research is also provided for each RSP activity. The overview provides the motivation for the use of an internal design specification language: DSL. The language description is initiated in section 2.2 with an introduction to several modelling constructs. The language is presented in a BNF notation along with a corresponding formal Predicate net model. Section 2.3 introduces the design support related constructs of DSL that aid in architectural design exploration and synthesis. The chapter concludes with an example of a design of a digital telephony switch to illustrate the DSL modelling approach. This example is also utilized in subsequent chapters to reaffirm various features of the DASE environment.

2.1 Introduction - Architectural Issues

As introduced in the first chapter, RSP can impact the design of a product mainly within the design phase of its life-cycle. The various activities (for the design phase) that are typically

pursued within the context of architectural design are shown in figure 2-1. The three major RSP activities are broken into more detailed ones to highlight the progression of the design. The design flow is from design capture to synthesis. The execution phase may undergo several iterations that are a part of a model refinement exercise. This requires the support of a model library capable of providing refinements to progress the model to a more detailed one which is suitable for synthesis [Booch 91].

2.1.1 System design

The system design activity is generally the starting point within the design phase of a product life-cycle. The activity involves the design capture of specifications into a model which can then be executed. Hence the language requirements at this stage encompass modelling capabilities. A model represents an abstraction of a domain under study and a large body of knowledge exists with respect to this abstraction. In the context of architectural level design, an abstraction is closely linked to specification capture, re-usability, and object-oriented features.

Specification capture has been an evolving discipline in system engineering. Today's system engineer will find many different mechanisms for design capture [Rattray 89]. Of these, an underlying formal basis is desirable to ensure consistent properties of a model

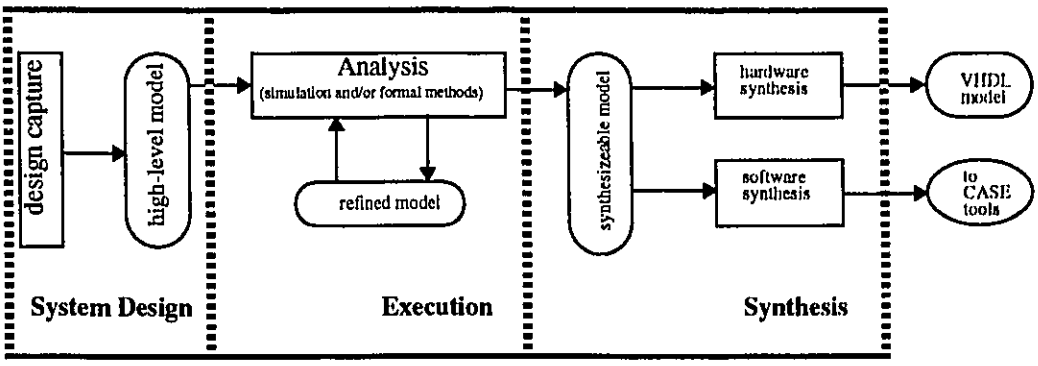


FIGURE 2-1. The Typical Architectural Design Phase

[Jacobson 92]. For example, LOTOS [ISO 88] and SDL utilize a process algebra based formalism [Milner 80] [Hoare 85] and have been developed to specify telecommunication systems. They have gained a considerable degree of attention in the protocol specification field. Hardware oriented formalism, such as HOP [Ganesh 89], exist based upon a functional programming paradigm. More general formalisms such as Petri-Nets (and their extensions) [Peterson 81], Temporal Logic [Moszkowski 85] and queueing networks [Kleinrock 76] are also applied to capture and analyze different aspects of a design. Additional paradigms can be found in [Gupta 92].

Most of the above formalisms have object-oriented capabilities or extensions permitting the modelling of high level communication paradigms. However, model re-use is still a significant shortcoming in this area. Although the theoretical notions exist, in practice the languages appear to require more consideration.

2.1.2 Design execution

After a suitable model is defined, the language must provide support for experimentation and analysis. These execution activities are crucial for a designer to explore a given design space, make appropriate trade-offs and partition the design to different hardware/software configurations. Such activities can be supported through design simulators and formal methods.

Formal methods refer to the application of techniques to prove properties of a system model (such as the existence of deadlocks). Whereas it is highly desirable to use formal analysis in all aspects of design, current methods make it very difficult to formally analyze large systems defined at detailed levels - where possible states in the system are just too large to manage. Additional problems also persist within current formal verification approaches. The formal techniques apply to the verification of a model of the system - not the system itself. The relevance of the verification is always dependant upon how well the model represents all aspects of the system. A considerable effort can be placed upon formally verifying a model, yet the model may not adequately describe the behavior of the

system in sufficient detail as to address all design queries. The amount of time to formally verify a complex system is currently unacceptably long.

Simulation is another technique for understanding system behavior where a given model is executed through predefined test scenarios depicting typical (and worst case) operations the system may encounter. For large systems, the scenarios are not exhaustive and provide a limited degree of confidence in the system. Hence the validity of simulation results are heavily dependent upon the assumptions the designer provides. Simulation can be employed in cases where insufficient detail is available for a formal method to be used effectively.

Simulation is widely deployed as a major aid and many general purpose simulators exist such as GPSS [Schriber 74], SLAM-II [Pritsker 86], and SIMSCRIPT [CACI 87] to provide reasonable queuing based system simulation. Other systems such as Designers Workbench also provide analysis support for small systems [Thomas 91]. Large complex simulation that require parallel or distributed processors and techniques like Time-Warp [Jefferson 83] have been utilized in generating speed-ups in most cases [Reed 87]. Petri-net based simulators such as Voltaire [Parent 91] and Loopn [Lakos 91] are also applicable for modelling concurrency.

With the assortment of simulation languages, model re-use becomes very difficult. Model interchange between different tools is generally not available and no standard exists for these languages to permit such an operation - although work is under way within the IEEE standards working groups to alleviate this problem and define requirements for a standard simulation environment [Tanir 94a].

2.1.3 Synthesis

Model execution activities are repeated, refining the model until a final model is derived which meets the designer's expectations and can also be synthesized. Synthesis implies taking a set of behaviors, constraints and goals and generate a suitable structure that can

implement the behavior while satisfying the constraints and goals. At the architectural level, synthesis can implicate both hardware and software. Software synthesis at this point requires significant research to integrate high-level specifications towards computer aided software environments and methodologies. The hardware field has seen a proliferation of design languages (along with the IEEE standard VHDL), which can be used as potential target languages for hardware synthesis.

The hardware design community has actively defined many design description languages pre-dating the standard VHDL. For example CASCADE [Borrione 93], CONLAN [Piloty 80], ELLA [Morrison 93] and Verilog [Verilog 92] are samples of languages available. Synthesis has been one of the most important applications of hardware description languages. Initially targeted to circuit level synthesis, tools and methodologies have evolved to synthesize designs from the algorithmic level.

Although highly significant, synthesis has played a secondary role to the design language definition. For example, only restricted subsets of VHDL and Verilog are suitable for synthesis. These languages have powerful constructs for simulation of hardware, however (due in part to their low level modelling features) automatic refinement of models has proven to be elusive at the very high levels of abstraction used within architectural level of design.

2.1.4 An Intermediate language

System designers have typically utilized many different languages to capture relevant aspects of a system. It is also generally agreed that one "unified" language or methodology is not capable of representing systems for all levels of design abstraction. It is also observed in the milieu that as the abstraction level increases, the analysis methods used in tools shift from a simulation oriented one to a formal basis [Ward 85]. Hence it is difficult to use an existing paradigm to address RSP concerns across a broad range of activities at the architectural level of design.

The issues reflected in the previous sections indicate an applicability of different paradigms at different stages of design. Consequently a possible solution to support architectural design is an intermediate language within the RSP to obtain the various input and output forms. Such a language would support inputs in the form of specifications (possibly defined in an existing specification language) and output a lower level synthesized design in a usable representation (such as VHDL). The intermediate language would also require modelling capabilities for high level description to facilitate specification input, as well as design exploration and synthesis constructs to produce the outputs.

This dissertation presents the Design Specification Language (DSL) as a potential intermediate language for architectural RSP within the DASE environment. The language primarily captures specifications by use of abstractions and is free from rigid disciplines of simulator oriented hardware languages. In addition, since the language must bridge the abstraction gap between high level system design notions to low level hardware descriptors, it must also possess the flexibility of re-defining and altering its model interfaces during design exploration.

This function of the language helped motivate the use of Prolog as the language for implementation of DSL. It may be noted that DSL is a meta-language in the sense that it is based upon Logic programming semantics, utilizing built-in predicates to define its own constructs. A major contribution of DSL is in its ability to capture high level specifications within a re-usable model and, with the aid of a library support system, refine the model to a state where it can be synthesized into an executable lower level representation language such as VHDL (refer to appendix D for DSL and VHDL differences).

DSL borrows the typeless notation to provide facilities for powerful abstraction and data manipulation. The language is designed to have a corresponding structural graphical correspondence. The graphical representations of the main constructs are shown in figure 2-2 and will be further elaborated in the upcoming sections. The language description is partitioned in two sections: sections 2.2 and 2.3. The first describes the modelling primitives (for design capture) of the language whereas the second section focuses upon

the experimentation and synthesis constructs to support design development.

2.2 The DSL modelling primitives

This subsection presents the DSL modelling primitives required for design capture, addressing the design input concerns. The basic syntax is presented in BNF notation with some regular expression shorthand to ease legibility. The shorthand symbols are:

{ } : encloses comments.

* : any number of sequences of the preceding expression.

+ : at least one or more sequences of a preceding expression.

[]: Sets of characters enclosed by square brackets imply that any one of the characters within the brackets are applicable. A range of possibilities is indicated with "-" within square brackets: i.e. [0-9a-z] indicates any single digit and lower case alphabet characters can be satisfied.

Characters in bold fonts are reserved words in DSL.

DSL adheres to Prolog's naming and syntax conventions. Hence the language primitives are defined by:

integer ::=[0-9][0-9]*

real ::= *integer* . *integer*

number ::= *integer* | *real* | *float*

rel_op ::= < | > | == | =< | >=

num_op ::= + | - | * | / | // | **is**

operator ::= *rel_op* | *num_op* | |

literal ::= [a-z][a-z0-9]*

atom ::= *literal* | *number*

list ::= [*member**] | [*member* | *member*]

member ::= <*null*> | *dsl_name* | *variable* | *number*

variable ::= [A-Z][A-Za-z0-9]* | [_][A-Za-z0-9]*

dsl_name ::= *literal* (*parameter*$^{+}$)

| *literal*

parameter ::= *atom*

| variable

A DSL program consists of a combination of DSL constructs defined as:

dsl_program ::= *dsl_model_constructs*$^{+}$ *dsl_experiment_constructs**

The *dsl_model_constructs* constitute the applicable commands that are used for model description and the *dsl_experiment_constructs* represent the commands used for experimentation and synthesis support. The former is defined below whereas the latter is visited in section 2.2.

dsl_model_constructs ::= *module_definition*

| *resource_definition*

| *ho_module_definition*

| *inheritance_definition*

| *persistent_port_definition*

| *path_declaration*

Each one of the possible DSL modelling primitives are described in the following subsections.

2.2.1 Modules: DSL building blocks

The basic construct within DSL is a modular object oriented design entity called *module* - the primitive building block of the language (refer to figure 2-2 for the graphical depiction). These are constructs that possess a name, a set of possible behaviors, and resources. Modules communicate with one another through messages - which trigger a defined behavior within destination modules.

The basic DSL notation for a module is:

module_definition ::= **module(** *dsl_name* ,[*behavior**]).

where *dsl_name* within the *module_definition* represents a unique identification of a mod-

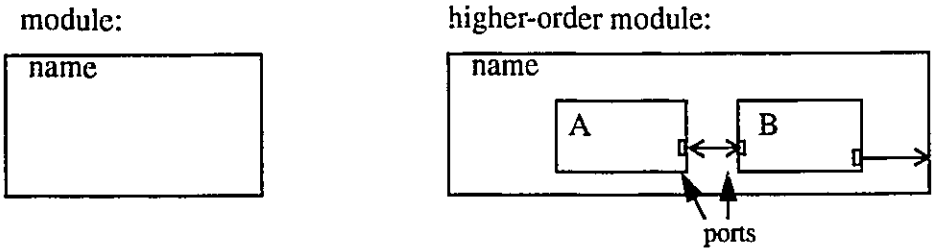


FIGURE 2-2. DSL Constructs

ule and is used extensively by DSL to resolve communication issues within a model. *behavior* is a list of different behavior(s) the module is capable of interpreting. The behavior(s) of the module represent the actions undertaken by the module when an event occurs. The occurrence of an event implies the arrival of a message from one module to the other module - which then attempts to execute the behavior associated with the message.

Modules operate within three conceptual states: *ready, busy* and *delayed* which are depicted in figure 2-3. *Ready,* indicates that the module can process a message. *Busy* implies that it is currently processing a message, and a *delayed* state indicates that a module is suspended (delayed) and will commence processing the message after a time period

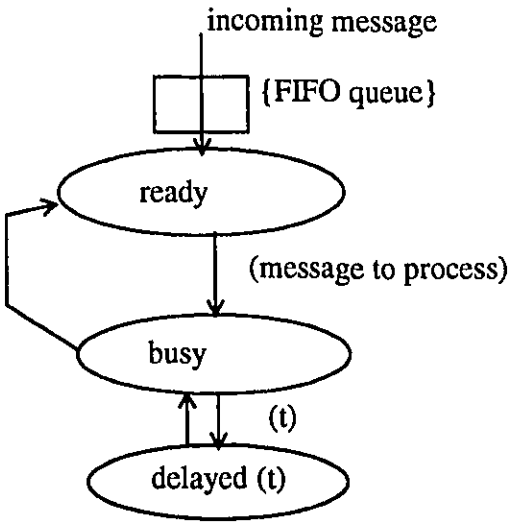


FIGURE 2-3. Basic states of a module

(t). When a module is in the *busy* or *delayed* state, arriving messages are queued within an input message queue to be processed in turn by the module. Message execution within a module in the *busy* state is atomic. However, pre-defined interrupt messages (described later) can interrupt the execution process if the module is in a *delayed* state.

The description of behavior and the respective possible actions will be deferred until a later section on behavioral description. However, some actions may use or manipulate data structures local to a module to model states, local variables etc. Such local data structures are facilitated within DSL with the *resource* statement. Adhering to the Prolog philosophy, local variables can also be lists - facilitating the management of structures such as arrays and queues. DSL commands to manipulate resources are defined in the data manipulation section later in the chapter. A resource is (a predicate) of the form:

resource_definition ::= **resource(** *dsl_name* , *dsl_name* , *values* **).**

values ::= *variable*
| *atom*
| (*values*$^{+}$)
| *list*

For example, a statement such as **resource***(module_name, res(P1,..Pn), (V1,..,Vn))* describes a resource local to a module (with name *module_name*). The resource (*res*) may be parameterized as above, and defines a set of variables (*Vn*) where *Vn* can be any Prolog element such as integer, lists and strings. The applicable operations that can be performed upon resources will be described within the behavioral modeling section.

2.2.2 Model composition

DSL permits composition of modules into higher-order (HO) modules. Connections between modules are established through the use of "ports". A port is a virtual communication channel between the module and its environment. The language deduces direction from information flow across ports. Port specification is not typed so that different levels of abstract information may flow through the same port.

The graphical representation of a higher order module (refer to figure 2-2) is similar to that of a module. The underlying subtlety is that higher order modules encapsulate a set of modules and define specified interconnections. Hence using these constructs, more complex (higher-order) models can be composed from simpler ones. Higher order modules possess no pre-defined behavior of their own - however the interaction of modules composed through such a higher order module establish the underlying behavior. A DSL model defines a higher order module in the form

ho_module_definition ::= **ho_module(** *ho_name* , *module_list*).
ho_name ::= dsl_name
module_list ::= [*dsl_name**]

The *ho_name* is a *dsl_name* (Prolog structure) and the *module_list* is a list of modules composed within the higher order module. Composition is not restricted just to modules, but is also applicable to higher-order modules. Hence members of a higher-order module may contain other higher-order modules as well as modules - however when referring to the constituent members of a given higher-order module, the term *module* will be used loosely to imply both. The interconnection of the constituent modules is achieved through a path declaration:

path_declaration ::= **path(** *module_x* , *module_y* ,[*port_x* , *port_y*]).
module_x ::= dsl_name | *variable*
module_y ::= dsl_name | *variable*
port_x ::= dsl_name | *variable*
port_y ::= dsl_name | *variable*

port_x and *port_y* are port names used to define a communication path between the two named modules. The connection implies a default direction from *port_x* to *port_y* - hence *module_x* is the name of a source module. The port names can also be written as variables, in which case no direction or specific port name is defined. In this case, the existence (or

need) for a communication path between the two modules is conveyed to the environment. The details of the connection manipulation by the environment will be elaborated in chapter 3.

Port names can also be associated to local variables - which result in *persistent* ports. Such a port is maintained as a resource within a module under the reserved resource name: persistent_port:

persistent_port_definition::= **resource(persistent_port** ,(*port_name* , *values*)).
port_name::= *dsl_name*

This capability permits ports to assume a value or state dictated by the most recent message that was sent from the port. Hence a state or value is said to persist on the port (similar to the behavior of a wire in hardware design). Persistent ports are suitable to model lower level constructs such as wires and buses where the value of the entity (voltage levels or high impedance) may be of concern. This also permits different levels of representation to coincide within a given model.

Since paths are equivalent to Prolog predicates, unification and variable instantiation can provide compact notation in defining some structures. For example, in the case of symmetrical connections, the path statement can be used concisely to define all module interconnections with a statement such as:

path(proc(X), mem(X), [port(X), mem_port(X)]).

This statement will interconnect all modules named proc(X) and mem(X) through their respective ports (i.e. proc(1) will connect to mem(1) etc.).
Relations can also be established through the port definitions. For example,

path(proc(X), mem(X+1), [port(X), mem_port(X+1)]),

indicates that proc(1) is connected to mem(2) through their respective ports. The path statements help define the necessary interconnections desired by the designer.

2.2.3 Module behavioral description

A module's list of behaviors describe all possible actions that may be undertaken by the module in response to a given message. The resultant behavior of a higher-order module composed of modules is a result of all possible interactions between the modules. Module behavior is a procedural description of actions which, within satisfaction of a set of constraints, may consist of:

i). communication initiation with other modules or internally,
ii). modification of resources or temporary variables associated with the module, and
iii). timing related directives.
More precisely, the form is defined as:

behavior ::= (*b_name* :- *condition** , *action*$^{+}$)
b_name ::= *dsl_name*
action ::= *communication* *
| *data_manipulation* *
| *timing*

Behaviors can be viewed as a set of Prolog clauses where each behavior can also be multiple clauses - applicable under different conditions. The head of a clause consists of the behavior name (*b_name*) - a unary or N position predicate. The body is a set of conditions and actions which are compound sub-goals of the clause. It should also be noted that DSL conditions are Prolog sub-goals to be satisfied, and the actions are sub-goals that are always satisfied, resulting in some desired side effects (such as message generation and resource manipulation).

For a given behavior, a set of conditions can be evaluated before further execution of the behavior is attempted. This facilitates the description of if-then-else type constructs as well as supporting state oriented behavioral description. The behavior selection may be based on built-in DSL predicates or Prolog conditional operators. Formally conditions are

described as:

condition ::= variable rel_op variable
| number rel_op variable
| variable rel_op number
| number rel_op number
| **check_res**(dsl_name , value)

The *check_res* statement is used to test for a resource (*dsl_name*) or its value. For example, ***check_res****(counter, X)* will return the value (*X*) of a resource named *counter*. Alternatively ***check_res****(Caller, telephone(5551211))* could return the name of a subscriber (*Caller*) associated with a specific telephone number. In either case, if there is no match the predicate will fail and another satisfaction of the behavior will be attempted.

DSL uses the Prolog programming style for resolving ambiguities and testing conditions. Hence a declarative and procedural style of writing behavior is possible. A behavior with the name *b(X)* for example is an unconditional attempt to satisfy the behavior statements. However, code in the form:

b(1):-c1, a1.
b(wait):-c2, a2.
b(X):-c3, a3.

are multiple behavior names that DSL will attempt to satisfy for the arguments "1", "wait" or otherwise. In addition a set of conditions c1, c2 and c3 determine if the action (a1, a2, a3) for the given clause will be attempted (since actions are always true - the conditions act as guards against erroneously selecting actions).

Having introduced the basic behavioral styles, the possible actions available within a behavioral description are described below:

i. Communication primitives

Communication is the main means for changing state within DSL models. The main action that invokes communication is the *send* statement. It's format is as follows:

communication ::= **send(** *destination* , *port* , *message*)
| **send(** *message*)
| **execute(** *message*)

destination ::= dsl_name | *variable* | *list* / [*]
port ::= dsl_name | *variable*
message ::= dsl_name

The first form of the *send* statement has three possible parameters: *destination*, *port* and *message*. The *destination* is the name of a destination module. The destination may be a multiple one, expressed as a list such as *[a, b]*. In this example the respective module will send the same message to modules a and b. An empty list [] will result in a message sent to the first module capable of interpreting the message. The destination can also be an [*] which has the potential effect of sending a message to all modules capable of processing the message.

The *port* is the name of a communication port of the source module that is identified as the desired source point from which the communication is to initiate. The *message* is the name of the message to be sent. This can be any valid behavior name that is comprehensible by the receiving module.

The second form of send is for internal messages, and is essentially a short notation of the first - where the destination is the source module.

The first two fields for the first form of *send* are optional, which provides extreme flexibility in the way communication can be initiated. The possible actions are described by the

following cases (parameters are undefined unless they begin with a lower case letter):

Case 1. All parameters are given: **send**(dest, port, message).

Action: *message* will be sent from the source module to the module with a name *dest*, through the source's port called *port*. A special case is when the destination is [*]. In this situation, the message will be sent to all modules that are connected to the source's port. This is equivalent to a broadcast message in some systems.

Case 2. No destination name is given: **send**(-, port, message).

Action: *message* will be sent to the first module capable of executing it and is connected directly or indirectly to the source module's *port*.

Case 3. No port name is given: **send**(dest,_, message).

Action: Essentially the same as case1. Ports are not necessary for communication, only for synthesis later in the process. This type of send statements will generate pseudo-communication channels within the DSL environment which can be used by a synthesizer to construct data paths.

Case 4. No port or destination given: **send**(_,_, message).

Action: This is similar to case 2. however any module that can interpret the message will be selected. This mode is an aggressive one and should be used with caution - since the communication paths are determined solely by the environment and not by the designer.

A message generated through a send statement will always be placed in the input (message) queue of the destination module. Hence there is an implicit causality associated with message communication.

The other communication primitive available is the *execute* statement. This statement is similar to an internal send statement, however it by-passes the input message queue of the module. Consequently the message will be processed immediately.

By using the *send* and *execute* primitives, different message ordering can be implicitly

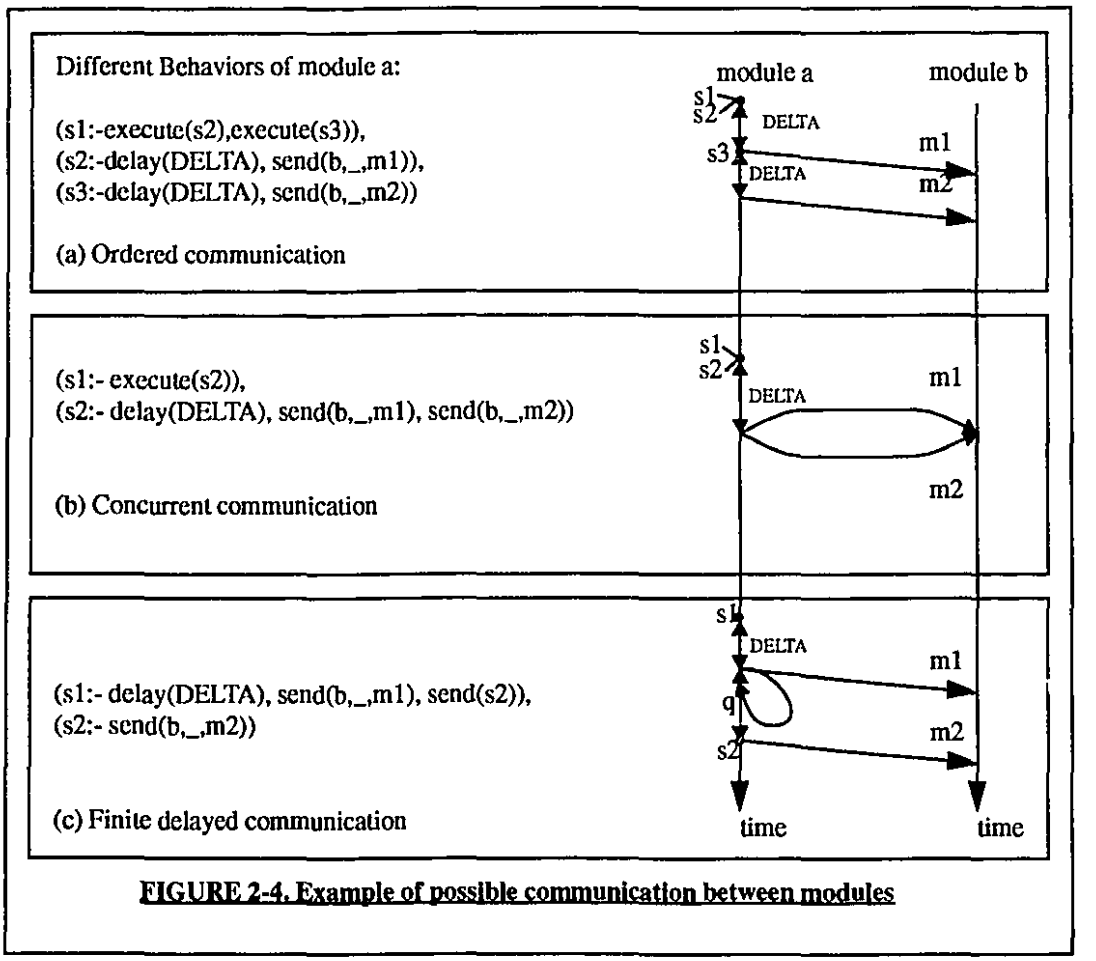


FIGURE 2-4. Example of possible communication between modules

established. Figure 2-4 presents some possible message combinations. The figure assumes that two modules (A and B) are communicating. In particular, module A is actively sending two types of messages and the diagram indicates the relative times at which the messages arrive at module B's input queue.

The first case shown is the effect of multiple execute and send statements. Three sets of behavior, each with a discrete delay of *delta* is assumed. The first behavior (*s1*) generates two execute statements for behaviors *s2* and *s3*. The latter two behaviors use send statements to generate messages *m1* and *m2* respectively. The destination module will receive the two messages *delta* time units apart in the order they are sent.

The second case depicts a concurrent transmission of messages *m1* and *m2*. This is achieved through behavior *s2*. Although message *m1* is listed before *m2* within the

behavior, the order of arrival at the destination is non-deterministic.

The final case shows the use of a combination of send statements. Behavior *s1* sends message *m1* after a *delta* delay, then sends an internal message (*s2*). The latter will be queued and processed after a finite queueing time (*q*) and *s2* will generate message *m2*. The effect of the internal send is to schedule *m2* after *m1*, but with an unknown delay (*q*) between the two.

ii. Data manipulation

Data manipulation actions can change values of internal variables defined by resources or temporary variables local to a module. The actions can utilize any valid arithmetic and list operators defined in Prolog. In addition, data values used within resources can also be manipulated within defined DSL statements. The following comprises a description of all the actions.

data_manipulation ::= *set_definition*
| *create_definition*
| *remove_definition*
| *probe_definition*
| *arithmetic*
| *built_ins*

set_definition ::= **set_res(** *dsl_name*, *values* **)**
create_definition ::= **create(** *resource_definition* **)**
remove_definition ::= **remove(** *resource_definition* **)**
probe_definition ::= **probe(** *dsl_name* , *variable* **)**
arithmetic ::= *variable operator expression*
expression ::= *number*
| **(** *expression operator expression* **)**

The first four data manipulation statements are those related to resources. The *set_res* statement permits the assignment of a value to the respective resource. For example, *set_res(state, busy)* will associate *busy* to the resource *state*. The *create* action will instantiate a new resource. Hence, *create(buffer(1), empty)* will define a new resource named *buffer(1)* with contents *empty*. The *remove* action will remove (or disassociate) a resource from a module. For example, *remove(buffer(1), full)* will remove the named resource whose contents are defined to be full. It should be noted that this operation has the effect of removing a defined fact from the underlying Prolog database. Hence if there are multiple entries with the same resource name, only the oldest and least used copy of the resource will be removed.

The probe action is reserved for resources defined as persistent ports. The invocation of the action results in the variable to be bound to a local value maintained for the persistent port. The statement permits a module connected to another module's persistent port to obtain the state of the port. The nuance of this representation is illustrated in figure 2-5. In the example, module (A) has a persistent port (pa) connected to another module (B). The implication is that there is a resource associated with module A, containing a state variable V. At some point in module B's behavior, a probe statement is encountered. The effect of this is to access the value V from the persistent port and bind it to the local variable ST. Persistent ports can only be read by other modules. Changing the value can only be performed by the module associated with the port (in this case module A).

The probe statement is useful for depicting lower level interactions. For example consider

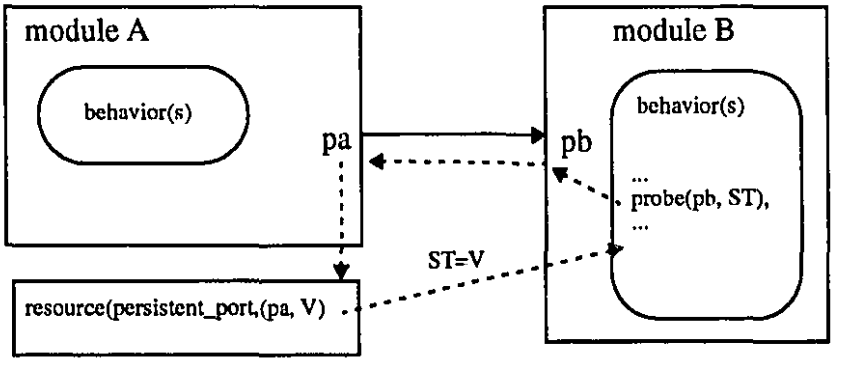


FIGURE 2-5. Example of Persistent Port

a model of a processor connected to a bus and memory. Assume that significantly detailed operations of the memory is of interest and the processor model generates messages representing a state of the bus (for example read_memory could imply a "0010" bit level on the control bus). If the bus were to be represented by a bus with a persistent port, the processor could affect the state of the port with its messages to the bus, and the memory module could utilize probe statements to latch onto the required values.

There are additional behavior predicates defined with DSL to help within the manipulation of data and creation of standard data structures such as queues. These are summarized below:

built_ins::=

not(*dsl_name* **)**	: negation of X (as in Prolog).
| **member(** *literal* , *list* **)**	: true if *literal* is a member of the list *List*.
| **remove_element(** *literal* , *list* , *new_list*)	: Removes the first occurrence of an element *literal* within the list *list* and returns the sub-list *new_list*.
| **list_length(***list*, *variable* **)**	: determines the number of elements *variable* within the list *list*.
| **append(***atom* , *list*, *new_list* **)**	: appends *atom* to the end of the list *List* to create the new_list (as in Prolog).
| **iqsort(** *list*, *new_list*)	: a quick-sort algorithm for the sorting of the elements of *List*. Elements are sorted in ascending order an placed in *new_list*.
| **last_list(** *list*, *atom*)	: identifies the last element *atom* from a queue named *list*.

iii. Timing

Up until now timing has not been elaborated upon. Timing behavior in DSL is encountered through simulation, hence there is no explicit formal timing model within the language semantics. However module delays can be simulated with the use of the *delay*

action. A *delay* action suspends the module for a given period of simulated time. After the elapsed time, the next consecutive action in the behavior is executed. The syntax of this action is:

timing ::= **delay(** *number* **)**
| **delay(** *variable* **)**
| *c_function* **, delay(** *variable* **)**
c_function ::= **call_c(** *atom* **,** *variable* **)**

The action can utilize a numeric value explicitly, or implicitly (as an unbound variable). The former simply implies the use of a numeric parameter such as *delay(5.4)* which will suspend further operation within a module for 5.4 time units. The latter is a case where a variable may be dependant upon an externally passed parameter - for example (Behavior_name(T1):- delay(T1),..). Here T1 is passed to the module as a delay parameter.

The *call_c* action can also be used in conjunction with a delay action to access random variables from a C library. Two examples of how random messages may be generated are given in figure 2-6. The first uses a list (RAND|REST) to select a random number (a table look up scheme could also have been used), suspend operation for that time period, generate a message (stimuli) and then access the next number. This structure has the advantage of replicating the same sequence of random numbers for every modelling scenario. The second example is that of a random number generation. A system call to C is used to access a random number generator (rand) and obtain a number between 0 and 1

```
module(random, [
(start([]):-
        print('Finished random messages')),
(start([RANDOM|REST]):-
        delay(RANDOM),
        send(_,_,stimuli),
        send(start(REST)))
]).
```

(a) Predetermined random numbers

```
module(random, [
(start:-
        call_c(rand, RANDOM),
        delay(RANDOM),
        send(_,_,stimuli),
        send(start))
]).
```

(b) Utilizing function calls

FIGURE 2-6. Examples of random number generation

(which is scaled afterwards). A Prolog predicate call_c(Function, X) is used to pass the random value from C to Prolog (some implementations of Prolog have predefined random value predicates that can be used directly). The value is then used to delay and generate a message as before. The difference here is that each simulation scenario will utilize different random values and the timing delays will not be identical.

Timing issues for modules are resolved through scheduling during simulation. The execution of a communication statement within a module's behavior is handled directly through the simulator after the destination name has been resolved (chapter 3 describes the simulator data structures in detail). Similarly, a delay statement will cause a re-schedule of the current message being processed, until a specified time period. During this period no further messages may be processed (the module's delayed state). There is however one exception to this rule. A default class of messages are defined within DSL to act as interrupt messages. These are defined through the statement:

interrupt_declaration ::= **isa(** *message* **,dsl_interrupt).**

In this declaration, *message* is defined as a priority message. If a module is in a delayed state, such a message will cause the simulator to immediately remove the current message, and schedule it after the priority message. After completion of the priority message, the interrupted message will once again delay the module. In the busy state, messages cannot be interrupted. A priority message can be interrupted by another priority message (there is no notion of levels of priority).

2.2.4 Inheritance and hierarchy

DSL permits modules to inherit behavior (and structure) from other modules (and higher order modules) - which expedites model re-use. A module that is written to be sufficiently generic and can be re-used by different parents is termed a *generic* module. By convention, such module names are superseded with a "generic_" to stress this property.

Creation of generic modules is simplified within DSL due to the relaxed coupling scheme between modules. A module does not necessarily require any knowledge about its destination, hence communication can be defined without destination names. Also, since ports permit different assortments of messages to pass through them, interconnection is also simplified. The model coupling is achieved by a higher order module accessing the generic components through the library. Generic modules will be explained in detail within the library support sub-section in chapter 3 - however it should be noted that if an inheritance mechanism exists and if the super-class (generic) module does not use communication primitives that embody destination names, then many different sub-classes of modules may re-use the super-class definition. Expanding upon this notion, different re-use capabilities can be observed as shown in figure 2-7.

The figure shows three different cases of inheritance. In each case, module A is the super-class module and B the sub-class. In the first case, the communication primitives do not specify destination nor port names. Hence the definition of module A resembles a truly generic module - whereby all the behavior is inherited as is.

The second case assumes the use of port names (but still no destination names) within the send statements. This form is still a useful generic module within DSL. The environment has the ability to adapt the sub-class to the additional demands imposed by the super-class. In this case, new port definitions may be required, hence additional port connections can be generated by DASE (this is described in detailed in the next section).

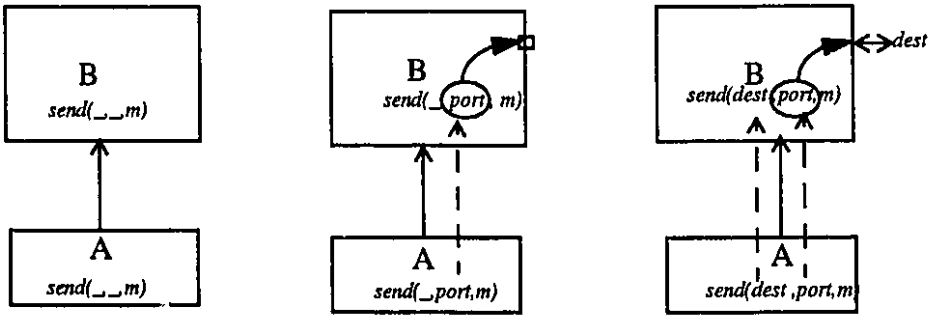


FIGURE 2-7. Module inheritance and re-use

The final case is not amiable to generic module design. The communication primitives refer to destination names, hence there is an added assumption that the sub-class must be able to communicate with pre-defined modules. If however, the communication is restricted to the boundaries of the super-class (within a higher order module), then there is no problem in inheriting.

The mechanism for inheritance is defined through the *isa* statement as below:

inheritance_definition ::= **isa(** *module_x* , *module_y* **).**

The statement is used to establish that entity *module_x* is a sub-class of entity *module_y*. *module_x* can also be defined to possess additional behavior local to its particular function. The statement also permits multiple relationships to be defined in one statement. For example ***isa****(processor(X), cpu(risc))* identifies that all modules defined with the name *processor(X)* (where *X* is a variable) inherit properties associated with a super-class module called *cpu(risc)*. Module inheritance is not limited to a single source - hence multiple inheritance is allowed as in the following:

isa(processor(xyz), cpu(risc)).
isa(processor(xyz), memory(32M)).

In this example, *processor(X)* is defined as before, however a module *processor(xyz)* is also indicated as inheriting some additional properties from another module (*memory*).

The *isa* statement can also be used to identify relationships between messages. For example the following two lines define relationships between three different messages:

isa(update(ADDRESS,VALUE), msg_reqst(SITE, component(ADDRESS), update(VALUE))).
isa(msg_reqst(SITE, component(ADDRESS), update(VALUE)), memory_write(SITE+ADDRESS, VALUE)).

The first identifies a relationship between *update* and *msg_reqst*. This may represent a correspondence between a high and low level protocol where an additional parameter (*SITE*) is provided by the respective module to interpret the message. In this case, the higher level update message will be executed as a *msg_reqst* message at the destination module. There is a further relationship also identified by the second line above; message *msg_reqst* can also be broken down to another message *memory_write* - which may be a message executable by a memory model. In this manner a high level representation of a message can be sifted through different levels of abstraction.

The inheritance scheme for messages is adequate for messages that can be mapped to one another. However, if a message requires more complex interactions at a lower level such as multiple messages, acknowledgments and error checking conventions, it is best to use a module that specifically models the desired protocol.

2.2.5 Module behavior as Predicate/Transition nets

The semantics of most elements of a DSL module can be described in terms of a Predicate/ Transition Net (PrTN) [Genrich 81]. The implication of this is that some properties of modules can be formally examined using analysis methods available to PrTNs. The analysis possibilities are described in chapter 3 which can provide for some verification of modules before placing them in a module library. This sub-section will present the PrTN model of a module and its limitations.

Petri-nets have been used in many areas to analyze hardware and software systems. The formalism provides for a relatively powerful way for defining asynchronous concurrent communication. One of the main complaints voiced about Petri-nets has been the problem of state-space explosion and complexity of the generated graphs. These concerns are minimized within DASE with the use of predicate nets to reduce net sizes and also by observing some restrictions to the general Petri Net formalism.

1. The first restriction is that modules exhibit a safeness property akin to that of Petri Nets.

This is observed from the fact that a module processes one behavior at a time - which is equivalent to a known bounded maximum number of tokens in a Petri-net or a strict PrTN.

2. The possible type of messages a module can accept or generate is finite and can be parsed. This implies that a finite number of input and output places for each module can be constructed. If DSL messages are represented as PrTN tokens, this also permits the definition of finite token sets.

Since the original introduction of Petri-Nets [Petri 62] several extensions have been proposed to increase their modelling power or expressiveness [Jensen 81][Jensen 89] [Murata 89] [Reisig 82]. Predicate/Transition Nets is one of the major extensions found in the literature. The extension permits the specification of different types that can be used to represent the different behaviors within a module definition. The following is an overview of the definition of a PrTN as defined in [Genrich 81], a complete formal definition of the work can be found in the reference.

An ordinary Petri Net graph (PNG) is defined as a triple PNG = (P, T; F) where P is a set of places, T is a set of transitions, F is the flow relation (elements of F are arcs between places and transitions) such that $F \in (PxT) \cup (TxP)$, $P \cap T = 0$, and $P \cup T \neq 0$.

Markers called tokens, move from place to place through the "firing" of a transition. The firing rule for a transition t in a Petri Net is satisfied when all input places contain at least one token. The execution of the firing rule results in the removal of a token from each input place connected to t, and the placing of a token in each output place connected to t.

A PrTN is composed of an underlying PNG, a set of annotations A (applied to the PNG) and a representative marking M. Hence, $PrTN = (PNG, A, M)$.
A is defined to represent four types of annotation, $A=(A_N, A_P A_T A_F)$:
i. A_N is termed the support structure of the PrTN. It is composed of a finite set of constants U, set of variables V (ranging over U), functions fi in U and relations Ri in U. It annotates the whole net rather than specific elements, describing static aspects of the net.

ii. A_P is a bijection between the set of places P and a set of variable predicates.
iii. A_T is a mapping of the set of transitions T, into the set of formulae (called transition selectors) employing only static predicates and operators.
iv. A_F is an arc labelling function which associates formal sums of tuples of variables and constants to the arcs. Hence the mapping is defined as: $A_F \rightarrow W \cup W^2 \ldots \cup W^n$ where $W = U \cup V$ and W^k denotes the set of all k-tuples in W.

The marking M represents the displacement of tokens within the net. PrTN tokens constitute separate individuals, whereas in ordinary Petri Nets they are indistinguishable from one another - with their count at places being of interest. Hence token sets are formed within PrTNs, attached to places and transitions of the net. If $x \in P \cup T$ and the arity of x is k, then a token set of x can be defined as $C(x) = U^k$. A given token $c=(u_1,\ldots,u_k)$ in a place $p \in P$ denotes the fact that a predicate (of arity k) corresponding to the place is true for a given instantiation of the tuple of arguments contained in the token. Similarly, for a token c in a transition $t \in T$, the implication is that the variables within the annotations specified for the transition and its incident arcs are substituted by the corresponding constants appearing in the token.

A DSL module can be represented by a PrTN as shown in the general form in figure 2-8. The diagram gives the basic structure of the net that can be generated for a given module. At the top, an input place (shaded in black) is defined where tokens annotated with possible message names arrive. The assumption is that only one token may reside in the input place at a particular time. To guarantee that no new message is processed by the net, a busy place is explicitly indicated where a token can be grabbed by active message tokens and released upon completion of their behavior.

The initial arcs emanating from the input place are annotated with different possible message predicates, hence each token type is directed to a different sub-net corresponding to the particular DSL behavior. Any conditions demanded by the DSL model for a given behavior are annotated upon the first set of transitions. In the event that a condition is required from a resource, a bi-directional annotated arc is created to a place maintaining a

token for the given resource (within the set of resources block). The output arcs of these transitions connect to a data manipulation sub-net which models the data manipulation statements of DSL. Eventually arcs connect from this sub-net to a set of transitions that make assignments to tokens as necessary, modelling a DSL communication statement, to generate a token to an output place.

The individual DSL behavior statements can be translated into Predicate-Transition net elements as depicted in figure 2-9. As can be seen, the basic DSL elements can be represented by a sub-net. Each corresponding sub-net can connect with one another through intermediate places between them. The connections to the intermediate places are made by input and output arcs within each sub-net. These arcs are shown as partly connected to each transition in the sub-nets of figure 2-9. However there are two limitations for translation to be observed. The first is that delay statements are ignored, hence timing is not explicitly captured in the net representation. Another limitation is that the name of the resource must be given in the use of the create statement. In DSL it is possible to define behavior such as:

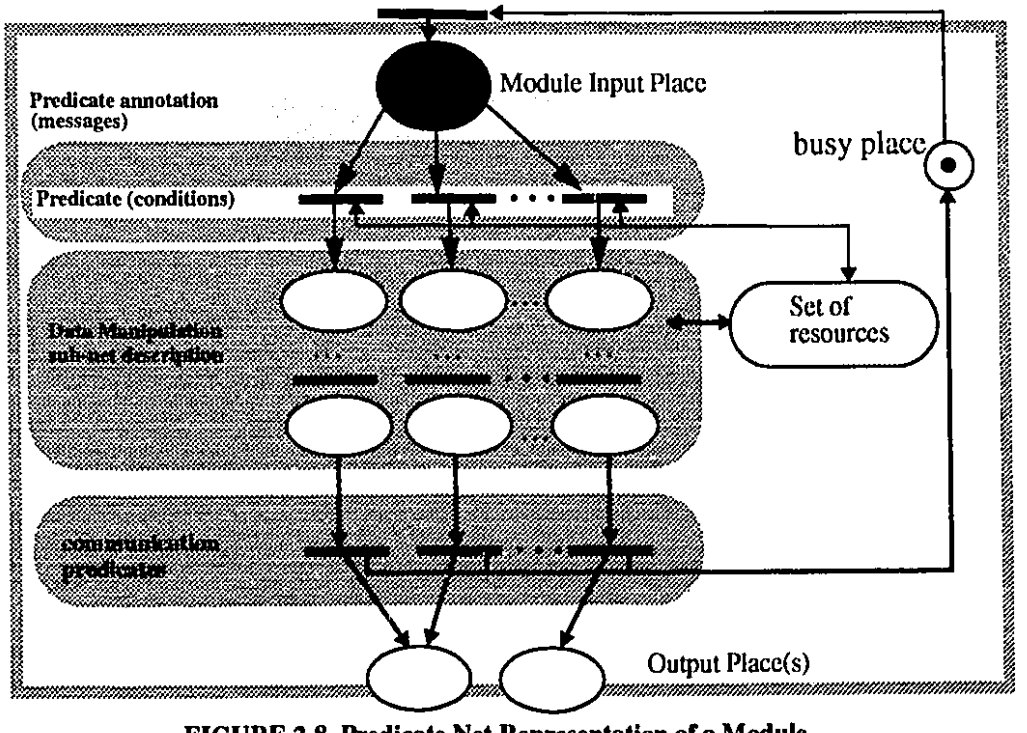


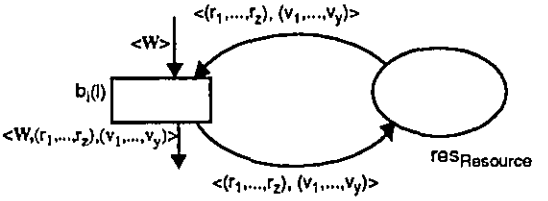
FIGURE 2-8. Predicate Net Representation of a Module

DSL Statement:

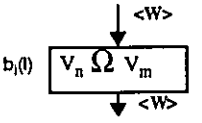
PrTN Representation:

Condition statements

check_res(Resource(r_1,...,r_z), (v_1,..,v_y))
i.e. check_res(count(1), Value).

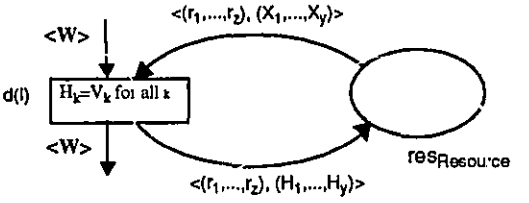


V_n Ω V_m
where Ω is a valid Prolog operator.
i.e. N < New.

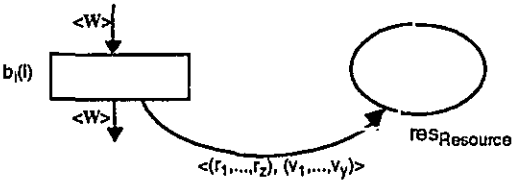


Data Manipulation statements:

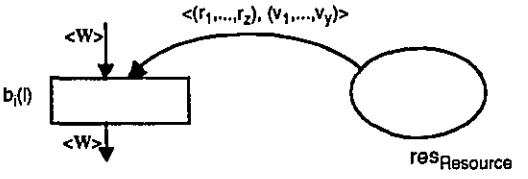
set_res(Resource(r_1,...,r_z), (v_1,..,v_y))
i.e. set_res(count(1), 0).



create(Resource(r_1,...,r_z), (v_1,..,v_y))
i.e. create(buffer(8), (1,0,0,1)).



remove(Resource(r_1,...,r_z), (v_1,..,v_y))
i.e. remove(pointer(1), VAL).



Communication statement:

send(Dest(d_1,...d_n), Port(o_1,...,o_m),
Message(a_1,..,a_z)).

note:
(execute(M) is simply an arc back to the input place)

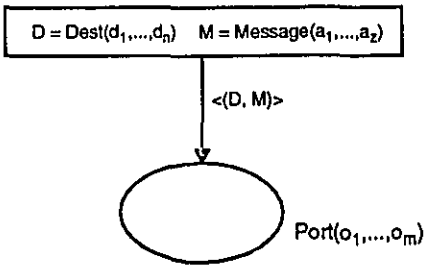


FIGURE 2-9. Predicate/Transition Net correspondence with DSL statements

b(Resource, Value):- create(Resource, Value).

Since this resource name depends upon an external type (supplied by another module), a corresponding place cannot be identified within the PrTN. The same problem also is attributed with the probe statement, since it requires a value from an external module.

Observing the limitations, an algorithm for translation from DSL behavior to a PrTN is given in figure 2-10. A small example is presented below to demonstrate the algorithm and clarify the use of the annotations. Larger nets are presented in the case studies described in chapter 4.

Assume a module *timer(Value)* written in DSL is to be translated to a PrTN, where the module is described as:

```
module(timer(Value),
[(clck:- check_res(count, M),
        Value > M,
        New is M+1,
        set_res(count, New)),
(clck:- send(Dest, out, alarm)),
        execute(reset)),
(reset:- set_res(count, 0))
]).
```

The DSL behavior describes a timer that counts the number of *clck* messages it receives and transmits an alarm message (through its *out* port) when the count has reached a limit *Value*. At this point the count is reset to 0 and the process repeats. An external reset message will also reset the count.

The behavior can be represented as a PrTN. The translation algorithm generates the PrTN net shown in figure 2-11(a). The figure omits the busy place which is part of a standard module's behavior. The input place accepts the incoming messages and one of three transitions (reset(1), *clck*(1), *clk*(2)) are fired depending upon the message name. A place is

```
Define a set T = {} and P={}.
For a given module name with arguments e1,..,en
  create an input place labeled in_name and add it to P.

For each behavior bi(a1,...ak) do:
{ create a transition node labeled as bi(l) where 0<l<n such that bi(l) ∉ T, and add it to T.
  current_transition = bi(l).
  current_label = (a1,...ak).
  create an arc with annotation <bi(a1,...ak)> from in_name to bi(l),
  For each DSL statement of the clause defined by behavior do:
  {
  If a condition statement is encountered then:
        if the condition = check_res(Resource(r1,...,rz), (v1,..,vy)) then:
        {       if a place res_Resource does not exist create it.
                create and annotate arcs as in figure 2-9.
        else
        {       annotate the transition bi(l) with the condition}
  if a data manipulation statement is encountered then:
        {create transition ti, place pi and arc(s) and annotate as required by figure 2-9.
         connect arc between current_transition to newly created place pi.
        annotate arc with current_label.
        current_transition = ti.
        current_label = {current_label appended with annotation of input arc to ti.}
        }
  if a communication statement:
        {create transition si and annotate as required by figure 2-9.
        create arc to appropriate output place
        }
  }
}
```

FIGURE 2-10. DSL - PrTN translation algorithm

required to hold a token relating a value to a resource (labeled res_{count}) "count". The two possible conditions for the behaviors for *clk* are represented by the *clk*(1) and *clk*(2) transitions. In the event of a *clk* token in the input place, *clk*(1) will first be tested and then, if needed, *clk*(2). The external message generated by the net is a result of following the *clk*(2) path, where an alarm message is sent to "Dest".

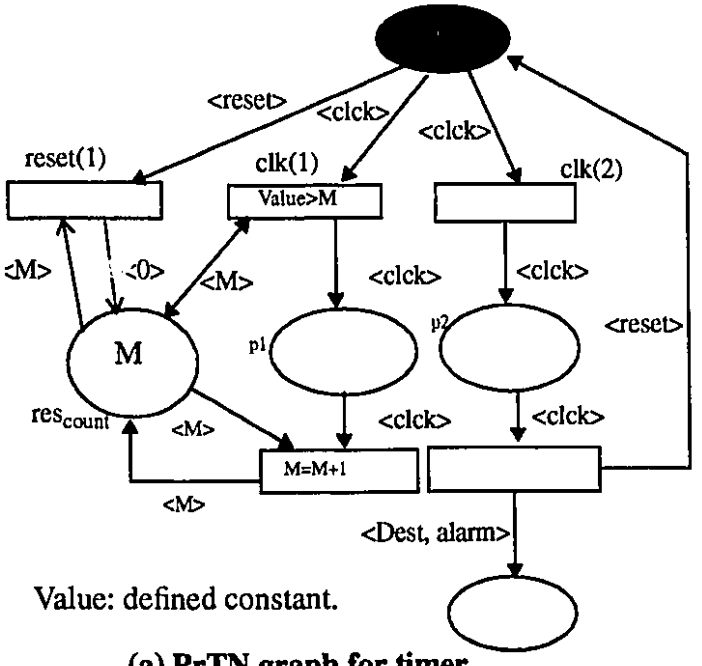
The graph can be depicted within the PROD language (developed at the Helsinki University of Technology [Gronberg 93]) or any other PrTN tool for further analysis. Figure 2-11(b) depicts the corresponding PROD representation of the PrTN graph. The syntax of the net description is straightforward where the corresponding places and

transitions are defined. Since module behavior is described as a bounded net, high limits are also defined at each place in the PROD description, ensuring that the boundedness property holds.

2.3 Design development support

As with any modelling language, DSL embodies a philosophy of its own to the design of systems. The language is suited to the top-down design approach whereby design detail is added and further refined by inherited modules. An initial design commences with a main model consisting of a higher order module. Sub-components are then systematically defined or utilized to accomplish a suitable level of design detail. This approach requires a reasonable model library to support design exploration and a structured approach to the design (which enforces a modelling discipline upon the designer) [Booch 91].

A modelling activity can be viewed as a composition of two models: a model of the environment and of the system under consideration. The relation of the two within DSL is given in figure 2-12. The model of the environment represents a conglomerate of the



(a) PrTN graph for timer

```
#enum reset, clk
#define value 3
#place input lo(<.0.>) mk(<.clk.>+<.reset.>)
#place res_count lo(<.0.>) hi(<.1.>) mk(<.1.>)
#place p_1 hi(<.1.>)
#place p_2 lo(<.0.>) hi(<.1.>)
#trans reset_1
        in  { input: <.reset.>; res_count: <.m.>; }
        out { res_count: <.0.>; }
#endtr
#trans clk_1
        in  { input: <.clk.>; res_count: <.m.>; }
        gate m<value;
        out { p_1: <.clk.>; res_count: <.m.>; }
#endtr
#trans clk_2
        in  { input: <.clk.>; res_count: <.m.>;}
        gate m==value;
        out { p_2: <.clk.>; res_count: <.m.>; }
#endtr
#trans t_1
        in  { p_1: <.clk.>; res_count: <.m.>; }
        out { res_count: <.m+1.>; input: <.clk.>; }
#endtr
#trans t_2
        in  { p_2: <.clk.>; }
        out { input: <.reset.>; }
#endtr
```

(b) PROD net representation

FIGURE 2-11. PrTN representation of timer example

external traffic and test cases to be applied to the model under consideration. Generally the applied load does not completely encapsulate all possible events that may occur upon the system, however it should at least cover the known scenarios of interest. State driven models can easily be described in DSL or the model of the environment can generate test cases through another language (such as Prolog or C). Hence there may be a mixture of DSL and other languages used. It should be noted that the model of the environment is used solely for experimentation and model support, hence it is never intended to be synthesized as part of the system.

Design development support, including simulation and synthesis constitutes the remainder of possible DSL commands in a DSL program. These commands can be grouped as:

dsl_experiment_constructs ::= *module_type_definition*
| *library_definition*
| *constraint_definition*

The different commands are defined below.

2.3.1 Co-design constructs

In designing models for the system under consideration DSL makes no distinction between software and hardware modules, however the language has provisions to allow the designer to identify these modules. This is of significance for synthesis since software modules will not be synthesized to the target hardware language: VHDL. A module can be

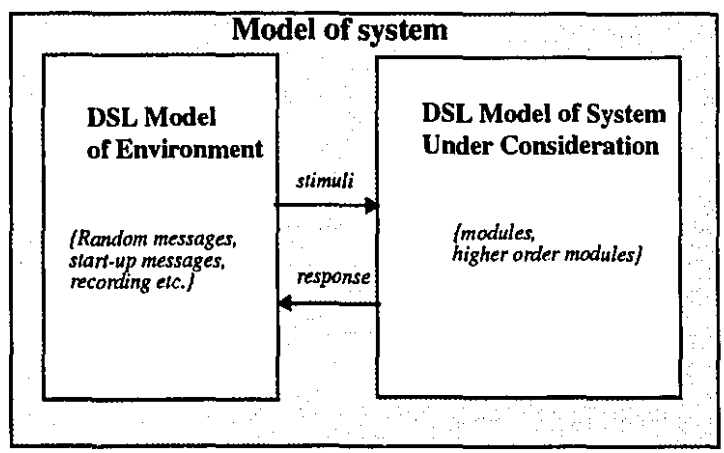


FIGURE 2-12. Modelling of a System

identified to represent a hardware or software entity simply within the predicate *module_type:*

module_type_definition ::= **module_type(** *dsl_name* **,** *synth_type* **).**
synth_type ::= **software**
| **hardware**
| **system**
| **protocol**

dsl_name is the name of a module. The *system* identifier implies that the module is a supporting module (such as a load generator) and is not synthesized. The *protocol* argument indicates that the module is used to implement a protocol between other modules and can be synthesized into a VHDL procedure or function.

If the *module_type* predicate is not defined by the user, the respective module is assumed (by default) to be a *system* type and is not considered for synthesis.

The representation mechanism provided by DSL facilitates the description of generic modules. These modules are defined entirely in terms of their potential behavior with respect to the environment and are free to bind to software or hardware constructs. Since there is no default notion of hardware and software, co-design modelling can be facilitated by DSL.

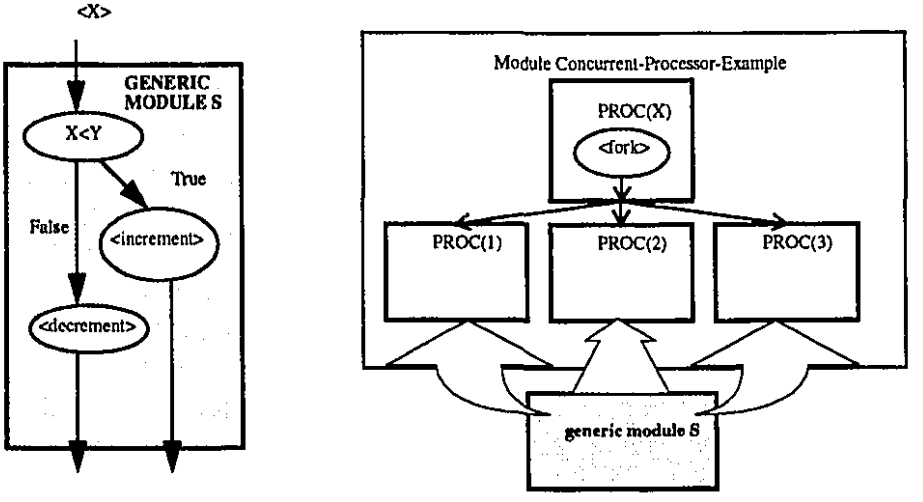
Co-design concepts within DSL are illustrated in figure 2-13. Modules can represent behavior and structure (HO-modules) so that many different abstraction levels of hardware can be described. Software behavior can similarly be captured by the behavior of a module. Since each module may execute one particular behavior at a time, it is straightforward to describe serial program execution in terms of delays and program control. This is depicted within figure 2-13(a) by the shaded generic module S (for software). The ellipses graphically represent a particular behavior, while the arrows indicate the program flow control. The names within "< >" indicate a name of a behavior defined within the module. In this simple example, module S will accept a message with parameter X and either

increment or decrement the value, depending on the condition X<Y (where Y is assumed to represent a local variable).

Concurrent software/hardware can also be described with a set of modules, confined within a higher-order module. An example of this is given in figure 2-13(b). Higher level program control is described by the module interconnects, while local module control flow is described by the behavior of each module.

In this example, a module *PROC(X)* spawns three identical messages indicated by the *fork* behavior. This behavior would typically be described in DSL in the form:
fork(X):- send([proc(1), proc(2), proc(3)],_, X).
The three concurrent processors (hardware or software) inherit a part of their behavior from the generic module *S*. Consequently each processor will process the message *X* concurrently. It should be noted that the processors *PROC(1-3)* can represent both hardware processors or software processes. Inheritance is the typical way in which a hardware module may absorb the behavior of a software module.

If a library is maintained with generic software modules representing specific functions such as branching, looping and sorting, then these modules can be inherited by other mod-



(a) (b)

FIGURE 2-13. CO-Design Construct Examples

ules as required. Parameterized generic modules allows for the dynamic definition of module resources and delays upon invocation. This facilitates the description of re-usable software modules such as "sorter" which may accept messages such as "quick-sort(SIZE)" and "binary-sort(SIZE)". The parameter SIZE may be used by the generic sorter module to determine the delay of the particular sort algorithm.

2.3.2 Model refinement

An initial DSL model can undergo several refinements by the environment during design exploration. Figure 2-14 depicts the interactions between environment components and the DSL model. The initial model is at a very close level of abstraction to the design requirements, hence if the requirements are not specific, the DSL model will leave room for significant refinement. In such circumstances, the user may intervene to make design decisions to help narrow the design space. Refinement occurs during the course of simulation as a consequence of interactions between the environment and the user.

The DASE environment provides the support (such as constraint checks and model library) for refinement. For example, whenever an ambiguous interconnection is detected between modules (no path exists between communicating modules), the environment will initiate a connection, or if a constraint is violated the environment can access its model base for replacement modules. This is all described in the next chapter, however DSL provides some basic library commands to facilitate these activities with the library support system.

The library support commands are defined as:

library_definition ::= **use_dsl_library**([*dsl_name**]).
| *configure_library_rules*
| *start_cond_rules*

configure_library_rule ::= **configure_library**(*dsl_name*):- {Prolog rules and DSL statements}

start_cond_rules ::= **start_cond:-** {Prolog rules and DSL statements}

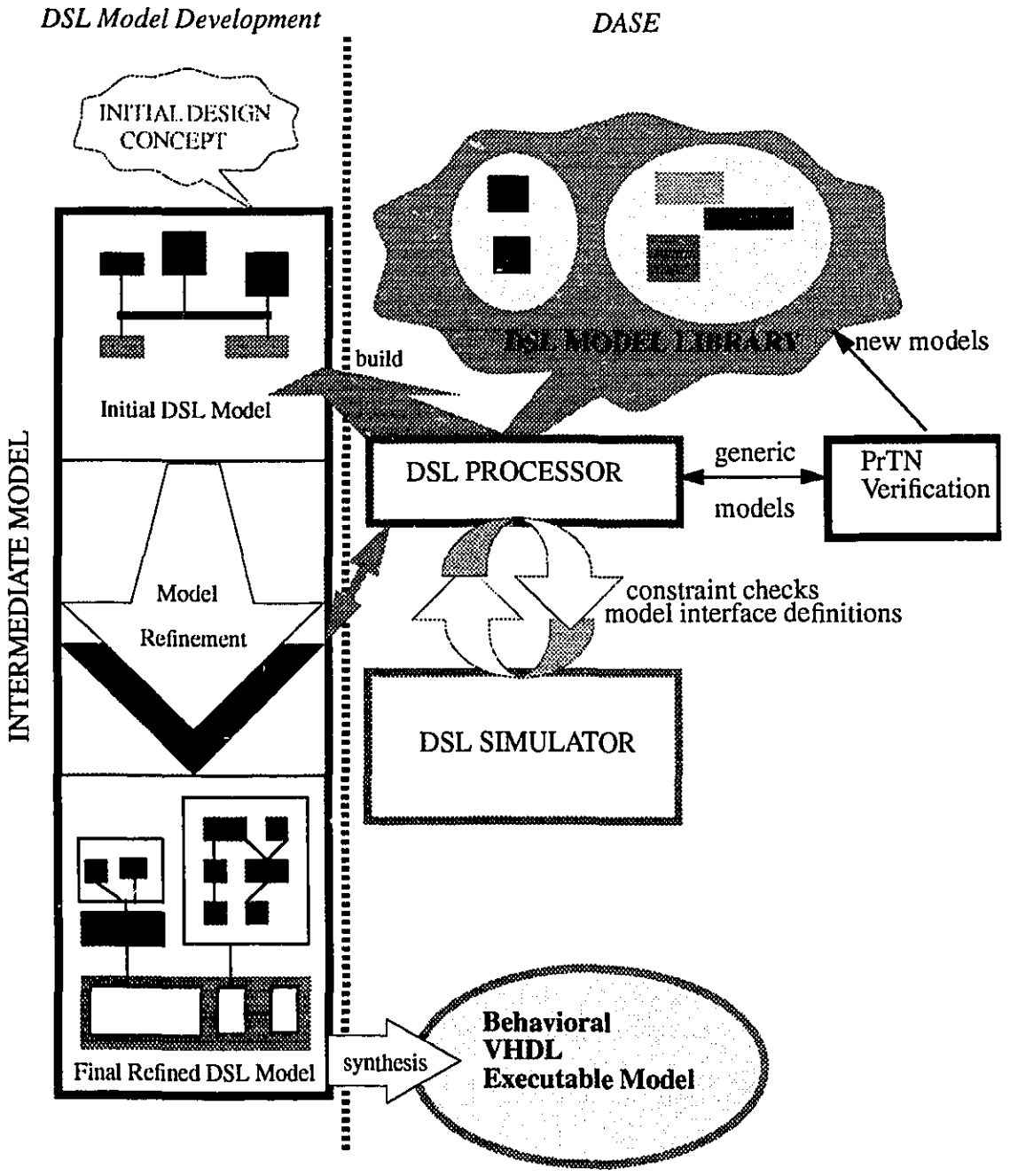


FIGURE 2-14. DSL Model Refinement

The *use_dsl_library* command references library module names (*dsl_name*) to be used within a given DSL program. A DSL program is free form, but adhering to basic Prolog syntactical guidelines. The next two commands are used to automate the set-up of DSL models. They are both headers for Prolog rules, hence any Prolog rule as well as DSL statements can be used to describe the actions to be taken by the respective commands. The *configure_library_rules* are rules associated to specific (library) modules. They describe the initial values the module may need upon initiation. These include creation of resources, providing initial values for free variables, and definition of default paths. The *start_cond_rules* refer to the starting conditions required for the module. This command is a mix of Prolog commands (generated by the environment) and DSL *send* statements. The use of the command is to send initial messages to modules so as to place them in a known state. The use of these commands will be described in detail in chapter 3.

An additional predicate is useful both during synthesis and simulation. This is the *constraint* statement which identifies a constraint to be imposed upon a module's resource or behavior. The form is:

constraint_definition::= **constraint(** *module_name* , *entity* , *const_type*, *values* **).**

module_name ::= *dsl_name*

entity ::= *dsl_name*

const_type ::= *dsl_name*

where, *module_name* is the name of the module which the constraint is applied to, *entity* defines what is being constrained (resource name or behavior name), *type* is the name of the type of constraint, and values (defined in section 2.2.1) is a constraint value applied to the *type*.

type is a user defined computation of a constraint (some pre-defined types are given within appendix A). For example, *constraint(clock, Clock_rate, upper_limit, 100)* describes that the maximum clock rate is 100 (nsecs) for the indicated module. The type upper_limit would correspond to Prolog code which will compute the current value of Clock_rate and ensure that it is below the upper limit.

2.4 DSL modelling example

This section introduces the design of a significantly complex telecommunication system to illustrate the hardware and software modelling capabilities provided by DSL, moving from specifications to architectural modelling. The example is intended to provide insight into the modelling approach taken to describe a desired system. The example will be revisited in the next chapter to emphasize the DASE environment's capabilities.

The example is the design of a traditional digital voice switch providing service for "Plain Old Telephone Service" (POTS). For this example it is useful to envision the design as that of a service provider - the switch will provide a specific service for the telephone. As part of the system requirements, the services that must be provided are tabulated below:

No.	Description of service	DSL behaviour name *
1.	- Detect an incoming call (offhook) from Source Agent - Provide dialtone	phone_off_hook(A) dialtone
2.	- Collect digits from Source Agent	dial_dest(A,B)
3.	- Translate digits	call_dialed(A,B)
4.	- Select Terminating Agent	status_response(B,STATE)
5.	- Establish connection between the two Agents	called_state(A,B)
6.	- Send Ring tone to Terminating Agent - Send Ring tone to Originating Agent - Send busy tone to Originating Agent	ringtone ringtone busytone
7.	- Detect Answer (offhook)	responded_call(B)
8.	- Detect a disconnect (onhook)	phone_on_hook(B)

Table 2-1: Services Provided to Telephone

* DSL Variables: A= Source Agent, B= Terminating Agent, STATE= {onhook, offhook, busy}.

These services will be addressed during the construction of the DSL modules within the remainder of the section. A digital time-space switch (DTSS) based upon conventional switch design will be introduced which can provide fast switching and flexible telephone service support.

A particular design of a DTSS can be viewed as composed of three major high level elements: a switch element, an interface element and a service provisioning element (all shown in figure 2-15). These are the initial elements the designer would specify as part of a top-down design approach. At this point an overview of the functions of the various elements are needed to proceed to the next level of design detail.

The switching element is the most real-time dependent component of the design. Its function is to take a time slotted incoming digital stream of voice samples (along incoming serial lines) and switch them to potentially different output lines (space switching) at different time slots (time switching).

The operation of the DTSS is as follows: An interface unit digitally samples (m) pulse code modulated (PCM) telephone calls and time multiplexes them towards the switching element at an allocated time slot (s). The switching element then transfers the time slotted data to different outgoing lines and slots. Generally the number of time slots (s) available between the interface units and switching element is such that $m \geq s$. This is because all telephones connected to an interface module are not active at once and do not require all s time slots. However if more than s calls occur at the same time, then blocking of some calls will occur (a fast busy signal is sent by the network to the telephone). The choice of the number of interface units (n) and time slots to use are design issues based upon assumed traffic load criteria of the network.

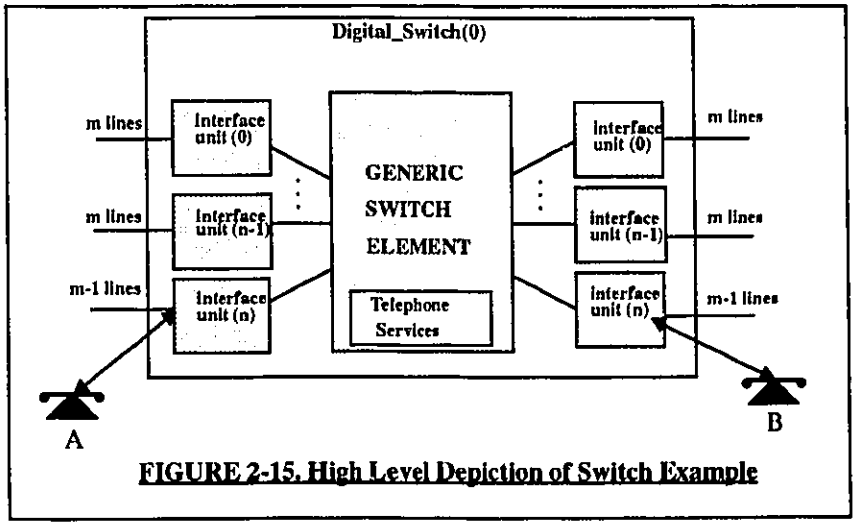


FIGURE 2-15. High Level Depiction of Switch Example

The service providing (Telephone Services) element can be located within the switching element. The function of this element is to provide the call setup support needed to establish a two-way telephone service. This element is assumed to be software. This is desired since different services (such as call waiting) can be added as needed via software rather than building custom hardware.

The serial time-multiplexed lines are given various specifications and conventions within the telecommunication domain. For example, the number of time slots available per line can vary depending upon the transmission rate. The standard for digital transmission in North America is DS1. DS1 carries PCM signals in a 193 bit frame composed of 24 (8 bit) channels (for voice samples) and a single framing bit. Transmitted at a rate of 1.54Mbits/ sec, this ensures that a frame is transmitted every 125 microseconds. For telephone grade quality, a 4khz voice needs to be sampled at 8000 times per second - which implies a sample (byte) every 125 microseconds. Hence a DS1 transmission facility can carry up to 24 separate voice samples across a single line.

The DS1 transmission standard is used between the interface units and the switch element, however the design will be flexible so as to allow different number of channels, always guaranteeing the minimum requirement of 125 microseconds per frame.

At the system level of design shown in the figure, some design considerations are issues such as traffic, the number of lines required for the switch or verification that a call will be set-up. For the designer to explore different trade-offs in a rapid and intuitive manner key re-usable or generic components must be available. For example the number of input lines is an important factor in the switch size. This design criteria will impact the number of interface units to use and the size of the switch element. Hence such a design parameter must be accessible to the designer so that the system may configure the internal of the switch accordingly. To satisfy high level design concerns, a digital time-space (*dts*) switch higher-order module can be defined as:

```
ho_module(dts_switch(NUMBER, INPUTS, CARDS, CHANNELS),
        [switch_element(NUMBER), interface_card(Y)]).
```

The parameters for dts_switch are relevant to the system designer. These are,

i. NUMBER: is an identifier for the switch. This is used to identify and configure different switches within a network of interconnected ones.
ii. INPUTS: defines the number of input (and consequently output) lines that a switch should have.
iii. CARDS: defines the number of interface units the designer may require.
iv. CHANNELS: defines the number of channels the serial lines should comply with (the default is 24 which is DS1).

Given these parameters for a DTSS, the environment should be able to generate the underlying model based upon the high level architecture shown in the figure and existing models in the model library. We will assume that no applicable library models are defined (the role of library support will be described further in chapter 3). As a result, the various details of the lower level elements must be described by the designer. The description of the structural details is presented below for the two main hardware components of the switch with some behavioral description. The description of the software element (telephone services) is given as an example of behavioral specification in DSL. The complete design description in DSL can be found in the appendices.

2.4.1 The Generic Switch element

The design presented in this section is one that is applicable to many large DTSS designs. The example concentrates upon the call-setup operations, so other operations that are important in the switch design (such as customer billing, error processing, maintenance and fault-tolerance) are not included in the example. However these are easily added as inherited (software) modules in a similar manner as that of the telephone services described later in the section.

The real-time design concerns for this element can be numerous and complex depending upon the size of the switch. The objective of the generic switch element is to capture the serial streams of data from input ports, identify the incoming time-slot and port, determine the destination port and time-slot and ensure the right incoming data gets transferred to the correct output stream. Since the number of input and output ports are variable, it is useful to separate their functionality into separate modules. The design will also require a control processor(s) to manage all the routing and call set-up functions. To avoid congestion on the control processor bus, a separate bus is desirable to move data from the input to the output.

Addressing these design issues, a typical generic switch element model is given in figure 2-16. The description is termed generic because the design is being constructed such that the components can be re-used within a model library framework.

The modules are described below:

line_in: This module is the interface for the incoming message stream. Incoming serial channels are captured as byte long words (or voice samples) and sent to the *buffer* module for temporary storage.

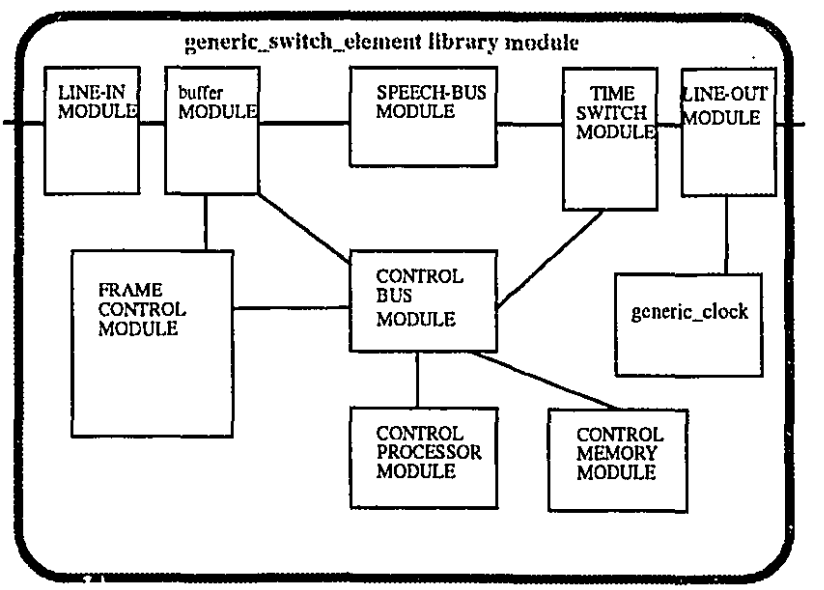


FIGURE 2-16. DSL Model of Generic Switch Element

buffer: The buffer is a temporary storage facility for all the line-in modules, which implies that the buffer size is dependent upon the number of *line-in* modules.

frame_control: The module provides the timing control for the speech_bus to transfer data from the input buffers to the output. Control is also provided for use by the *control_processor.*

speech_bus: This is a model of the bus between the input buffers and the output (time_switch).

time_switch: This module, under *control_processor* control, latches onto data from the *speech_bus* and transfers it to the appropriate line-out module.

line_out: This module depicts the output interface. The module contains a buffer for each channel it supports (i.e. for DS1 there are 24 buffers). New data from the *time_switch* overwrites relevant buffers and information from each buffer is transmitted in sequence under control of the *generic_clock* module.

generic_clock: This module provides the timing requirements for DS1 type transmission equipment. The behaviour of the module will be further elaborated in this section.

control_processor and control_memory: The routing and call-setup control is maintained by these modules. The memory contains the routing tables for call connections as well as the control software.

As a sample DSL behavior, the details of the *generic_clock* library module are presented below.

```
1.0   module(generic_clock(INDEX, NO_OF_CHANNEL), [
1.1       (clock_count(NO_OF_CHANNEL):- send(clock_count(0)),
1.2           send(_,clock_port,frame_cycle)),
1.3       (clock_count(NEW):- NEW<NO_OF_CHANNEL,
1.4           Clock_rate is (125/NO_OF_CHANNEL),
1.5           delay(Clock_rate),
1.6           COUNT is NEW+1,
1.7           send(_,clock_port,clock(NEW)),
1.8           send(clock_count(COUNT))) ]).
```

```
1.9     start_cond:- write('Setting up initial messages '),nl,
1.10        isa(X, generic_clock(_, CHANNELS)),
1.11        retract(current_module_processed(XXX)),
1.12        asserta(current_module_processed(X)),
1.13        send(clock_count(CHANNELS)).

1.14    configure_library(generic_clock).
```

This segment of code is typical of library modules. The first segment (lines 1.0 to 1.8) is the behavioral description of the module. The module processes only one message: *clock_count*(NEW), where NEW is an undefined parameter or NO_OF_CHANNEL (a parameter passed to the module). The parameters of the module are given in line 1.0. INDEX is an identification number given to the clock module and NO_OF_CHANNEL is the number of channels the clock is to support. The latter permits the module to produce timing control for a wide variety of transmission components.

Lines 1.3 to 1.8 demonstrate the typical actions that occur upon receipt of a *clock_count*(NEW) message (where NEW=<NO_OF_CHANNEL). Line 1.4 establishes a Clock_rate for each channel and in line 1.4 the clock module is delayed for this period of time. After the delay, lines 1.6 and 1.7 increment the value of NEW and send a *clock*(NEW) message to any module connected to its clock_port. In line 1.8 an internal clock_count message is transmitted with the increment value of NEW to start the cycle over again.

Lines 1.1 and 1.2 guarantee that the module provides a modulo-NO_OF_CHANNEL based timing. If the clock_count parameter is at the limit (NO_OF_CHANNEL) an internal message *clock_count*(0) resets the parameter. A frame_cycle message is also generated to modules connected to the clock's clock_port. This message advises the modules of the start of a new frame (or a sync bit in DS1).

The second segment of code (lines 1.9 to 1.13) define starting conditions for this module. The required start message in this case is a clock_count message with an integer value,

used to start the clock module. This value can be anything in the range 0 to NO_OF_CHANNELS. If a value is not provided by the user as in line 1.13, the environment obtains the value from the instantiation of the module (lines 1.10 to 1.12) if possible - otherwise an error message is given to the user. In this case the instantiation of the generic module would provide the NO_OF_CHANNELS value which is a valid parameter for the clock_count message. Lines 1.10 to 1.12 are simulator directed operations added by the system.

Line 1.14 completes the last segment of code in the library module. These are configuration rules. This module does not have any significant configuration requirements, hence there is nothing added by the environment.

2.4.2 The Interface Unit(s):

The design demands are less severe for the Interface Units. These units must provide the basic services that an individual telephone will require - services such as dialtone, ring-back tone and capture the dialed digits. These services can be provided in many ways. In this example they will be provided by a programmable *controller* module. The module's "program" and hence its behaviour is defined by another library module: *sub_server* (for subscriber server). Hence the direct interactions with the telephone can be altered by changes to the latter module.

The Interface Units also require a *line_interface* module which will provide the necessary transmission capabilities between the *switch_element*. The behaviour of this module is identical to the combined *line_in* and *line_out* modules within the generic_switch_element. Also a timer module is defined (an instantiation of the *generic_clock* module) to provide the appropriate timing control for the transmission components. The DSL representation of the Interface Unit is provided in figure 2-17. The details of the *sub_server* module (controller functionality) will not be elaborated here, however some aspects will be shown during the description of the telephone call-setup services in the following section.

2.4.3 Software constructs - Telephone Services

The example assumes that the telephone services provided by the switch are controlled through the *control_processor* modules in the *generic_switch_element*. This is a design choice. Alternatively, most of the services could be off-loaded to the interface_units, or to another processor. These alternative scenarios can easily be incorporated within this example by inheritance of different parts of telephone services through other modules.

Telephone services require communication between elements of the switch and possible other switches. For example a call originating on one switch could be trying to ring a party connected to another switch. To transmit these type of signalling information, the switches will reserve channel 0 (the first channel in a frame) as the default communication channel. Hence all telephone services messages passed from the *generic_switch_element* to the *interface_units* will be through this channel. Alternatively a separate line can be allocated specifically for signalling information - which is another design choice.

The telephone service routines are defined within a *pots_server* (for Plain Old Telephone Service) module. This module will eventually be synthesized to software, hence will not impact the DSL model in a structural manner - its behaviour being inherited by the *control_processor* within the switch.

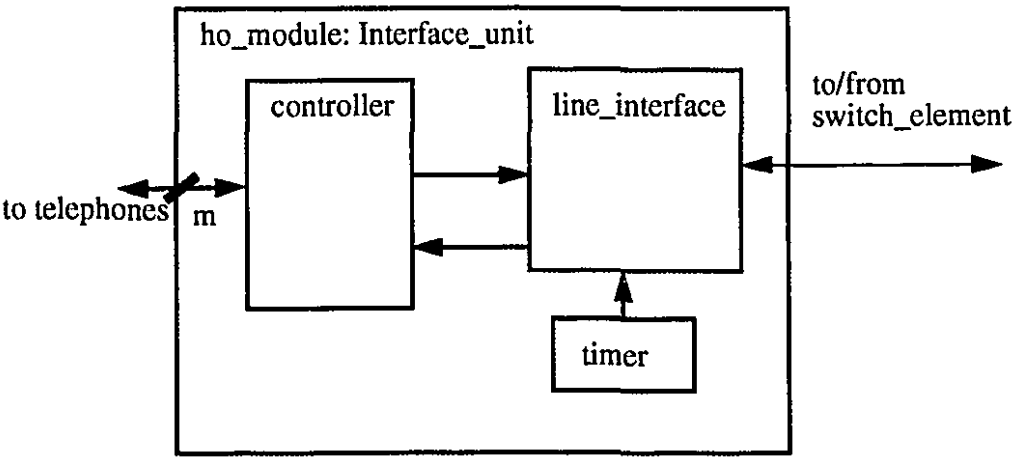


FIGURE 2-17. DSL Model of the Interface Unit

The basic operation of the pots_server is as follows:

When a phone call is placed, a *call_dialed* message is sent from the respective *interface_unit* to the *switch_element* - indicating the caller and destination phone numbers (the phone number in this example is identified as telno). The module creates three resources to accommodate the call. The first is the *connect* resource which identifies the two parties and also the status of the connection. The status can be *connect_request* (connection not yet established), ringing (the destination is ringing), and connected (connection is established). The other two resources are the agent and terminator which contain local data for the caller and destination phone respectively. Once complete, a message line_status is sent to the destination phone's interface_unit to determine if the telephone is occupied. The DSL code is:

```
(call_dialed(telno(ID, SDNO, STEL), telno(RSNO, RDNO, RTEL)):-
        create(connect(telno(ID, SDNO, STEL), telno(RSNO, RDNO, RTEL)), connect_request),
        create(agent(STEL), SDNO),
        create(terminator(RTEL), RDNO),
        send(_,line_port(RDNO), update_buffer(RDNO, line_status(RTEL))))
```

As a response to the last message, the respective *interface_unit* will eventually send a status_response message indicating the state of the destination phone. The phone can be *onhook* (not busy) or *busy* (talking to or dialing another line). In the former case, the *pots_server* will allocate a free pair of channels for the two telephones (this information is contained within a resource named *connect_table*) and also send ring-tones to the two. If no free lines are available, a fast-busy tone will be sent to the calling party and the data structures will be cleaned up. The DSL code is outlined below:

```
(status_response(B, STATE):-
    check_res(connect(A, B), connect_request),
    send(called_state(A, B, STATE))
),
(called_state(telno(SW,DS,A), telno(BSW,BDS,B), busy):-
```

```
    check_res(agent(A), Aport),
    remove(connect(SW, telno(DS,A), telno(BSW, BDS,B)), SS),
    remove(terminator(B), Bport),
    send(_,line_port(Aport), update_buffer(Aport, busy_tone(A)))
),
(called_state(telno(XX,DS,A), telno(YY, BDS,B), onhook):-
    check_res(agent(A), Aport),
    check_res(terminator(B), Bport),
    check_res(channel_table(Aport, ChA), unused),
    set_res(channel_table(Aport, ChA), A),
    check_res(channel_table(Bport, ChB), unused),
    set_res(channel_table(Bport, ChB), B),
    Index1 is ChA*20+Aport,
    Index2 is ChB*20+Bport,
    set_res(connect(telno(XX,DS,A), telno(YY, BDS,B)), ringing),
    send(_,mem_port(ID), mem_write(Index1, Index2)),
    send(_,mem_port(ID), mem_write(Index2, Index1)),
    send(_,line_port(Aport), update_buffer(Aport, channel_allocate(A, ChA))),
    send(_,line_port(Bport), update_buffer(Bport,channel_allocate(B, ChB))),
    send(_,line_port(Bport), update_buffer(Bport,change_state(B, ringing))),
    send(_,line_port(Aport), update_buffer(Index1, ring_tone)),
    send(_,line_port(Bport), update_buffer(Index2, ring_tone))
),
(called_state(telno(XX,DS,A), telno(YY, BDS,B), onhook):-
    write('No free lines available'),
    remove(connect(telno(XX,DS,A), BB), SS),
    remove(terminator(B), Bport),
    send(_,line_port(DS), update_buffer(DS, fast_busy(A)))
),
```

If the destination phone is picked up, a *responded_call* message is received by the pots_server. The module will consequently ensure the proper data structures are updated. This is shown below:

```
(responded_call(telno(X,DSB,B)):-
    check_res(connect(A, telno(X,DSB,B)), ringing),
    check_res(terminator(B), Bport),
    set_res(connect(A,telno(X,DSB,B)), connected),
```

```
    send(_,line_port(Bport), update_buffer(Bport,change_state(B, busy)))
).
```

The phone conversation will terminate when the originating party hangs up the telephone. The *phone_on_hook* message is generated by the respective *interface_unit* when a telephone has been detected to become on-hook. The *pots_server* determines if the telephone is the originator or the destination and accordingly cleans and updates the respective data structures as below:

```
(phone_on_hook(telno(X, DS,A)):-
    check_res(connect(telno(X, DS,A),telno(Y, DSB,B)), connected),
    remove(connect(telno(X, DS,A), telno(Y, DSB,B))),
    remove(agent(A), Aport),
    remove(terminator(B), Bport),
    set_res(channel_table(Aport, ChA), unused),
    set_res(channel_table(Bport, ChB), unused),
    send(_,line_port(Bport), update_buffer(Bport,phone_off_hook(B)))),
(phone_on_hook(B):-
    check_res(connect(A,B), connected)),
(phone_on_hook(telno(SW,DS,A)):-
    check_res(agent(A), Aport),
    remove(agent(A), Aport),
    send(_,line_port(Aport), update_buffer(Aport, change_state(A, onhook))) ),
(phone_on_hook(telno(SW,Aport,A)):-
    send(_,line_port(Aport), update_buffer(Aport, change_state(A, onhook))) )
```

This completes the behavioral description of the telephone services. At this point, design exploration and further model development can be undertaken by use of the modelling environment (DASE) - which is the topic of the following chapter.

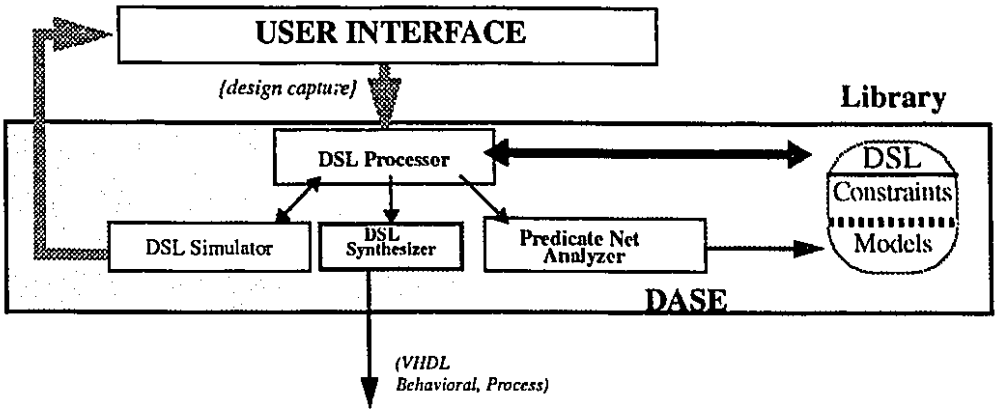
Chapter 3 - Design and Synthesis Environment (DASE)

3.1 Introduction

The Design And Synthesis Environment (DASE) is the supporting framework for DSL to provide the necessary facilities for design exploration and rapid prototyping at the architectural level of design. The environment accepts a DSL model as input and provides capabilities for model configuration, library support, module verification, design exploration via simulation and module synthesis to a subset of VHDL. The environment was motivated by a desire to support a top-down structured approach to design.

The major components of DASE are shown in figure 3-1. Specifications are entered through a user interface and captured by the environment through the use of the internal representation; DSL. The language is interpreted through a DSL processor which manages the library and DSL model configurations. The library support is a key component in permitting model re-use and facilitating rapid prototyping. The DSL processor also interacts with a DSL simulator and a predicate/transition net based analysis tool (PROD) [Gronberg 93] to provide verification of module properties. The DSL simulator interacts with the DSL processor during design exploration activities to refine the model as required. The final component is the synthesizer which translates the (refined) hardware DSL models into a behavioral VHDL representation.

This chapter will present the various components of DASE and describe their functionality.



LOWER LEVEL DESIGN AUTOMATION TOOLS

FIGURE 3-1. DASE Organization

These include; The DSL processor and library support, analysis of modules, simulation and synthesis.

The current implementation of DASE is invoked at the Prolog interpreter prompt with the command: "> [dsl].". The DSL processor and simulator are then automatically loaded within the system displaying an introductory message.

3.2 DSL Processor

The DSL processor is a central component which manages the DSL models and is the primary interface with the user. Its main functions are the management of library modules, constraint checking (with the simulator) and module communication support (such as determination of unspecified destination modules). The library support system of DASE provides a high level means for the modelling and management of models for telecommunication systems. The support entails the organized storage and retrieval of modules such that library modules are accessible to the environment as required. This implies that the DSL processor be capable of accessing modules at different levels of representation and utilize them as warranted. Design details which may be necessary, but not of interest to a particular user can be processed by the DSL processor transparently - freeing the designer to concentrate upon the pertinent details. For example, if the modeler is modelling an application layer of a protocol, all lower layer messages used within the

model can be transparently maintained and re-used through a library supporting the protocol. Alternatively, different levels of abstraction can also be modelled within the same model due to the ability of the environment to maintain a message hierarchy.

3.2.1 Library support

The top-down design approach of DSL is supported by an experimentation and model management mechanism where re-usable "generic" model components are made available to support simulation of different design options. A versatile library support system is managed by the DSL processor which enables creation, storage and retrieval of module libraries in an organized manner. Libraries maintain all module information, as well as added rules regarding any constraints to be imposed on the modules, any configuration rules to be applied to the components of the library, and the interface specification of the library module. The interface specification is created by the library system to define exactly what ports are available for communication with the library module.

Library modules are invoked with the *use_dsl_library([NAMES])* DSL statement defined in the previous chapter. Upon invocation, each library module is processed in an order identified through a model hierarchy tree. The tree is created by the DSL processor to establish a relationship between modules. An example of a model hierarchy is shown in figure 3-2. The top level module (module a1) utilizes two other library modules (modules

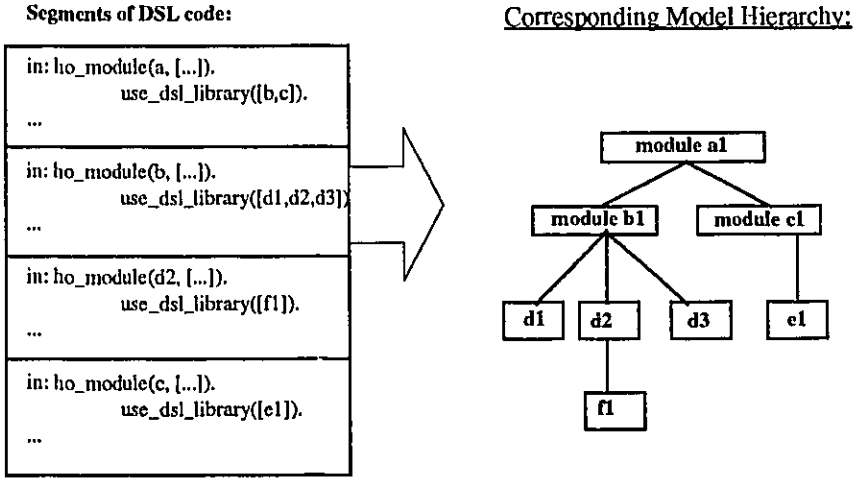


FIGURE 3-2. Model Hierarchy in DSL

b1 and c1). Hence a tree is created with module a1 identified as the root and modules b1 and c1 as subsequent nodes. The procedure is then repeated for the subsequent nodes to create the complete tree.

The library modules are processed breadth-first by the DSL processor. This ensures that any specific requirements (such as interconnections and instantiations) are identified by the immediate parent module. Hence library module configurations are affected by the environment into which they are invoked. For example, referring to the hierarchy example, any instantiations of module b1 are defined by module a1 - which can impact the way in which b1 instantiates its subcomponents d1, d2 and d3. This is further demonstrated within the library construction predicates explained below.

Library modules are created using the *build_dsl_library(NAME)* predicate interpreted by the DSL processor which interactively constructs the necessary data structures for the library module *NAME*. The *NAME* is a valid module name loaded within the environment or available as a file name in a model directory. The basic structure of a library module is depicted graphically in figure 3-3. The predicate proceeds through three phases of construction of a library module structure.

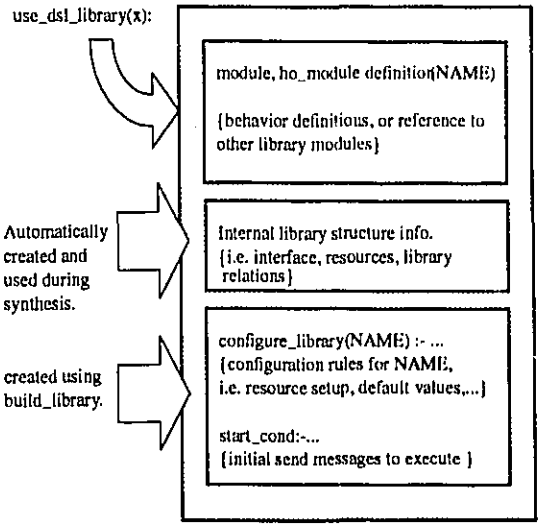


FIGURE 3-3. Library Module Structure

Phase 1. The first stage is to create the module and higher-order module definitions of the library. The system searches for the specified module and if successful, it recreates the behavior of the module (the predicate will fail in the case of an undefined module).

Phase 2. The second stage is the construction of the library configuration rules and starting conditions which are created interactively. The system provides a template which, when completed by the user, specifies the nature and use of the module. The DSL processor will examine the module behavior and identify (to the user) any resources, and path names used by the module requiring special treatment during configuration. The configuration rules are stored as Prolog rules with the predicate head defined as:

configure_library(NAME).

Configuration rules allow for the creation of module resources, interconnection definition (for higher-order modules), definition of constraints and instantiation of other modules. During library invocation the rules are executed by the DSL processor to create the required resources for the library module. The rules may be parameterized, allowing for different possible types of path definitions. The user may also define default values for parameters used by the library module which is maintained as a defaults(Module(D1,..., Dn) predicate where the parameters D1 to Dn are default values the system is to use. Defaults are only used if the argument value is not defined by the environment during model configuration. Conflicting or missing interconnections during configuration are identified by the DSL processor and reported back to the user.

The structure of a configuration rule is shown in figure 3-4. The regions outlined by bold text are inserted by the DSL processor as required. Four lines (with italic text) are shown that require user input. These are definitions of resources (local variables, data structures), paths (interconnections), other modules and user defined sub-rules. The figure indicates only one line per definition, however there

```
configure_library(NAME):-
        isa(INSTANCE, NAME),
        asserta(resource(INSTANCE, RESOURCE, VALUE)),
        asserta(path(INSTANCE, OTHER, [PORT, OTHER_PORT])),
        asserta(module(NAME2, [])),
        other_rules(...),
        fail.
configure_library(NAME):- write('Library module '),
        write(NAME), write(' configured\n').
```

List of Variables:
NAME: name of the library module.
INSTANCE: an instance name for NAME extracted from the environment.
RESOURCE: name of resources for INSTANCE.
VALUE: initial value of RESOURCE.
OTHER: name of module(s) to interconnect with INSTANCE.
PORT and OTHER_PORT: port names for INSTANCE and OTHER.
NAME2: any other module instantiations that are dependent upon NAME.
other_rules(...): are further user defined sub-rules that are defined. The functor and arity of this predicate is not significant and is used merely here as an example.

FIGURE 3-4. Sample Configuration Rule Structure

can be multiple lines for each definition. It should also be noted that the configuration rules are applied to all instances of **NAME**.

Start conditions are also identified in this phase of library construction. A rule beginning with the *start_cond* predicate identifies any messages that need to be sent to the module to place it in a valid state of operation. This allows some control over the initial starting state of modules. All *start_cond* rules are executed in turn during library invocation, scheduling any initial messages for the simulator.

Phase 3. The third stage is to capture the interface and resource information for the module. This is achieved by the DSL processor by grouping the required module resources, required ports and relations. In the latter case, a class relationship (if applicable) is associated with the module library and is established with the *isa* predicate as with modules. The class relationships for a model library are maintained within a file called "lib_def". The file consists of *isa* predicates identifying any relationships that may exist between library modules.

Figure 3-5 illustrates an example for building a library module (called x) that is of the same class as an existing module (y). The user, shown on the left hand side of

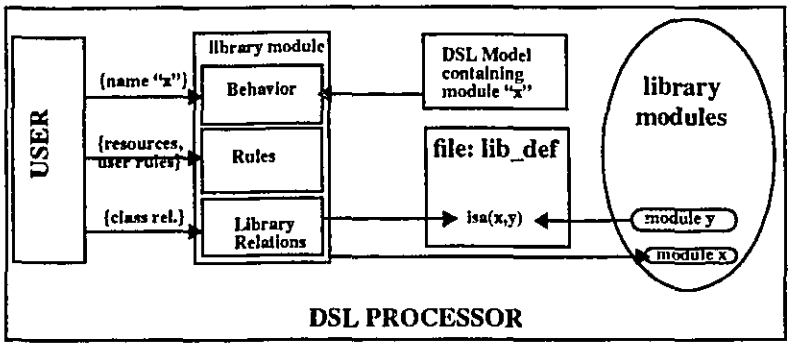


FIGURE 3-5. Library Construction of Module "x"

the figure, supplies the module's name, configuration rules and identifies a relationship to another module (y). The DSL processor will verify the existence of the library module and verify that all resources and paths used within the module's (x) behavior have been properly defined.

The class relation allows library components to be identified and utilized by the system during simulation, design exploration and synthesis.

Upon creation of a library module, DSL also verifies that constraints are not in conflict and there is no inconsistency with the library interface and its associated modules. Library configuration rules allow a library to configure itself depending upon different conditions during invocation. These rules allow a generic library module to be reused in different ways as the design requirements demand.

3.2.2 DTSS example revisited - library support

The DSL model for the DTSS that was introduced in chapter 2.4 can be structured as a re-usable library model. Library structure is maintained upon invocation as a tree of modules - where the root is the top-most level (level 0) module which has directly or indirectly instantiated the others. Each parent library module configures its sibling within the environment as required. This property is demonstrated in figure 3-6(a), which shows a segment of DSL and user written Prolog code which are part of the *dts_switch* library module's configuration rules.

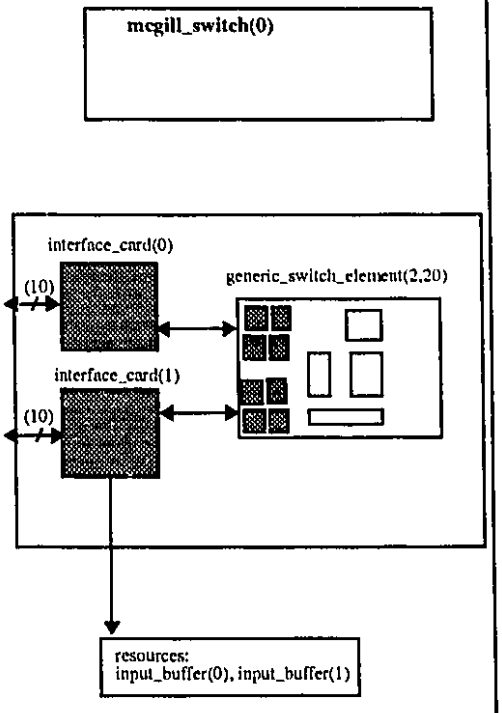
```
ho_module(dts_switch(NUMBER, INPUTS, CARDS,
  CHANNELS), [switch_net(NUMBER), interface_card(Y)]).

use_dsl_library([
    generic_switch_element,
    interface_unit
    ]).

configure_library(dts_switch):-
    isa(INSTANCE, dts_switch(N,I,C,CH)),
    II is I//C,
    asserta(isa(switch_net(N), generic_switch_element(C,CH))),
    asserta(isa(interface_card(Y), interface_unit(II, CH))),
    asserta(path(interface_card(Y), switch_net(N), [into_port(Y),
    trunk_in(Y)])),
    asserta(path(switch_net(N), interface_card(Y), [trunk_out(Y),
    intl_port(Y)])),
    asserta(ho_module(switch_net(N), [])),
    example_config(N, II, C).

example_config(N, IN, 1):-
    Index is N*16,
    asserta(ho_module(interface_card(Index), [])),
    tel_port_config(Index, IN, 0).
example_config(N, IN, CARD):-
    CARD1 is CARD-1,
    Index is N*16+CARD1,
    asserta(ho_module(interface_card(Index), [])),
    tel_port_config(Index, IN, CARD1),
    example_config(N, IN, CARD1).

tel_port_config(Index, 0, CARD1).
tel_port_config(Index, IN, CARD1):-
    isa(INSTANCE, dts_switch(N,I,C,CH)),
    IPRIME is I//C,
    INN is IN-1,
    Index2 is IPRIME*CARD1 + INN,
    asserta(path(interface_card(Index),          INSTANCE,
    [t_pair(Index2), tel_line(Index2)])),
    asserta(path(INSTANCE,          interface_card(Index),
    [tel_line(Index2), t_pair(Index2)])),
    tel_port_config(Index, INN, CARD1).
```



(a) DSL Description (b) Rule Illustration

FIGURE 3-6. Configuration Rule Example

The *dts_switch* module (the parent module in this case) identifies the sibling library modules that it requires using the *use_dsl_library* statement. The next statement *configure_library(dst_switch)* defines the configuration rules for resources, ports and module instantiation that are identified by the *dts_switch* module. In this case, the *generic_switch_element* and *interface_unit* modules are instantiated using configuration parameters (such as number input/output lines and channels) defined from the instance of the *dts_switch* module. A sub-rule, *example_config*, determines the number of *interface_units* to instantiate and also defines the port interconnections for each with the *dts_switch*.

The effects of the library configuration rules are demonstrated in figure 3-6(b). The figure shows an example of a mcgill_switch(0) module which is an instance of the library module dts_switch(0, 20, 2, 4). The parameters identify the mcgill_switch(0) to be configured as a switch with a switch number 0, 20 telephone lines supported, 2 interface units desired and 4 channels per multiplexed line. The parameters have been kept small so that their effects can be captured in the figure. There is also a significant impact upon the resources within the various modules. The modules that are affected are shaded.

The restriction of the design domain enables the use of library modules with different configuration rules in various interconnection schemes. This is useful to enhance model modularity and re-usability within the application domain.

3.2.3 Modelling support

A *constraint* statement was introduced in chapter 2 as part of the DSL support commands. This command is also used by the DSL processor to identify any design condition violations. Constraints are related to a module and are used to define limits upon parameters of a given module, resources or messages. These can be defined to be system wide constraints (affecting more than one module) or local constraints (restricted to individual modules). For example, the maximum size of a memory module, can be regarded as a local constraint. However, if multiple instances of the memory module is used by a main DSL model to define a larger memory sub-system, then this can be viewed as a system level constraint. System level constraints are defined within the main DSL model, whereas local ones can be established within library modules. After invocation of the library, DSL performs routine checks upon the constraints so that they are not violated during simulation.

The DSL processor also resolves the communication decisions for messages sent through undefined ports or destinations. A search algorithm is applied to deduce the final destination module. The approach utilizes the backtracking in Prolog to attempt to find a module that is both connected (through path statements) and is capable of understanding

the message. The algorithm for the destination search is described as the following:

Let Source be the sender of a given message through an unknown port or destination.
Let Port be a valid output port available to sender.
0. Assign sender = Source.
1. choose Dest_port and Dest such that there exists a path(sender, Dest, [Port, Dest_port]).
2. If Dest is a module then
If $message \in \{Behavior\}$
where {Behavior} is the set of all behaviors within module Dest,
then final_destination=Dest;
Exit algorithm.
else backtrack and repeat step 1 with a new Dest.
3. If Dest is a higher-order module then
sender = Dest, Port = Dest_port, repeat step 1.

The algorithm determines a path moving away from the source module to potential destination modules. This is accomplished in step 1. Step 2 is the stopping condition which is the discovery of a module (*final_destination*) that contains the intended message within its set of behaviors. On the other hand, if at this point the message is not within the module's behavior set, backtracking is utilized to search for alternative modules. Consequently the search is a depth-first search of destinations.

Step 3 addresses the case when the destination is a higher-order module. The sender and port is then assigned to that of the higher-order module name and port. It should be noted that there are also provisions to ensure that the algorithm always moves away from the source module (except when back-tracking) so that messages do not bounce back to the source.

The algorithm returns an error condition if there is no possible destination with a connection to the sender. The environment then will attempt to search for modules that are not connected to the sender, but contain the message within their behavior set. If a module is discovered, that can accept the message, then a pseudo path is defined by the environment. If no module is found, the message is trapped and an error message is

generated.

3.3 Petri-Net analysis

Section 2.3 introduced a relationship between module behavior and predicate/transition net (PrTN) representation. This section will illustrate possible analysis options available to the designer using the PrTN. The analysis of large models is a considerable effort requiring experienced modelers with a good understanding of the analysis methodology and the application domain. The techniques presented here are applications of the basic PrTN analysis methods to DSL behavior, but are intended to be an introduction to DSL analysis. The innovative use of these techniques along with the art of modelling (such as partitioning higher-order modules into sub-nets) will produce further analysis methods and benefits. A tool called PROD (developed at the Helsinki University of Technology) has been used to perform the PrTN analysis described in this section.

3.3.1 Module analysis

Analysis upon module representations can conceptually be performed at two levels: the individual module level, and the higher-order module level (sub-nets of modules). There is a difference in the two approaches. The first approach restricts the overall complexity of the PrTN since nets representing individual modules are strict nets (one message is processed at a time) and the state space is generally manageable. A module's PrTN representation can be analyzed using reachability or invariance techniques [Genrich 81]. This analysis can detect inconsistent behavior structures, unused DSL code and undefined message ports. Module level analysis is supported within DASE with the use of the PROD analysis tool. .

Some basic properties of modules can be verified at this level of analysis. These include, verification of safeness, liveness, sequence of firings and reachability to a particular state. To perform the traditional analysis of a net generated from module behavior, the desired initial markings must be specified. The net will have some tokens placed from the starting conditions, however, there is a need to identify the incoming token stream (representing the

possible messages). This involves playing the token game. The petri net can be initialized in many different ways depending upon the analysis objective. One possible scenario is illustrated in figure 3-7. The PrTN representation of a module is indicated by the shaded rectangle. Around this representation, a supporting net can be attached consisting of a place to hold all the messages that can be interpreted by the module, a place (busy) ensuring that only one message is processed at a time, and a place (message) identifying the message being processed by the PrTN. The generation of the supporting net is trivial since only three places and two transitions are required. The input place to hold the incoming messages can be populated with tokens typed as possible message names (extracted from the module behavior). Consequently a reachability tree can be generated with a tool such as PROD and analysis of properties pursued.

As an example, the reachability tree for the PrTN of the timer introduced in chapter 2.2.5 (figure 2-11) is shown in figure 3-8(a). The initial marking assumes either a *reset* or *clk* type token arriving to the *input* node (no other message type is acceptable). In the event that a *reset* token arrives, the resource value (within *res_count*) is initialized to 0. The *clk* token causes the value to increment or a reset to occur. The latter is when the resource is equal to the count limit defined by *Value*. The PROD language permits one to query the reachability

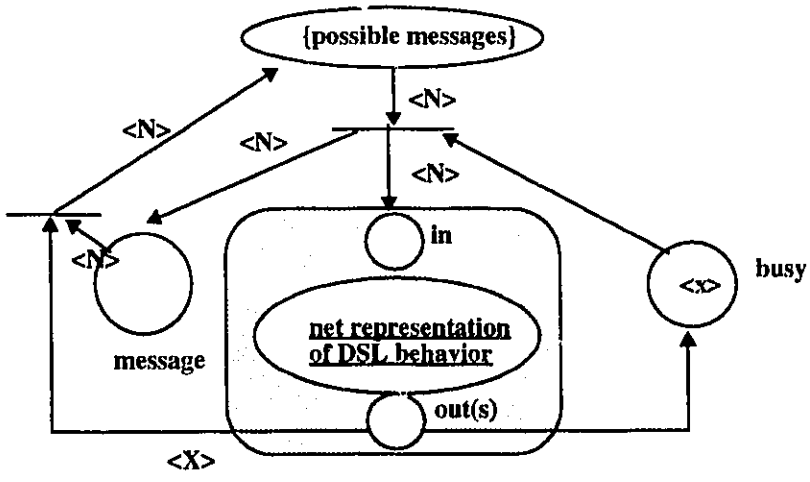


FIGURE 3-7. Analysis setup for Module Behavior

graph.

A typical session within PROD is presented in figure 3-8(b) and is shown here only as an example of some possible analysis that can be pursued using the tool. The example assumes a count limit of 3 for the timer (hence the behavior is that of a modulo-4 device). The listing is segmented into four sections. The first shows the statistics for the reachability graph, indicating the number of nodes and initial marking of the graph. In this case, the initial marking consists of a clk message in the input place. The second segment shows a query command in PROD. This command displays the sequence of firings to reach a terminal node in the reachability graph. The resource variable is indicated in bold, showing the successful increment of the variable at each clk message. Eventually when the resource value reaches the limit, a reset token is generated and placed into the input place. The

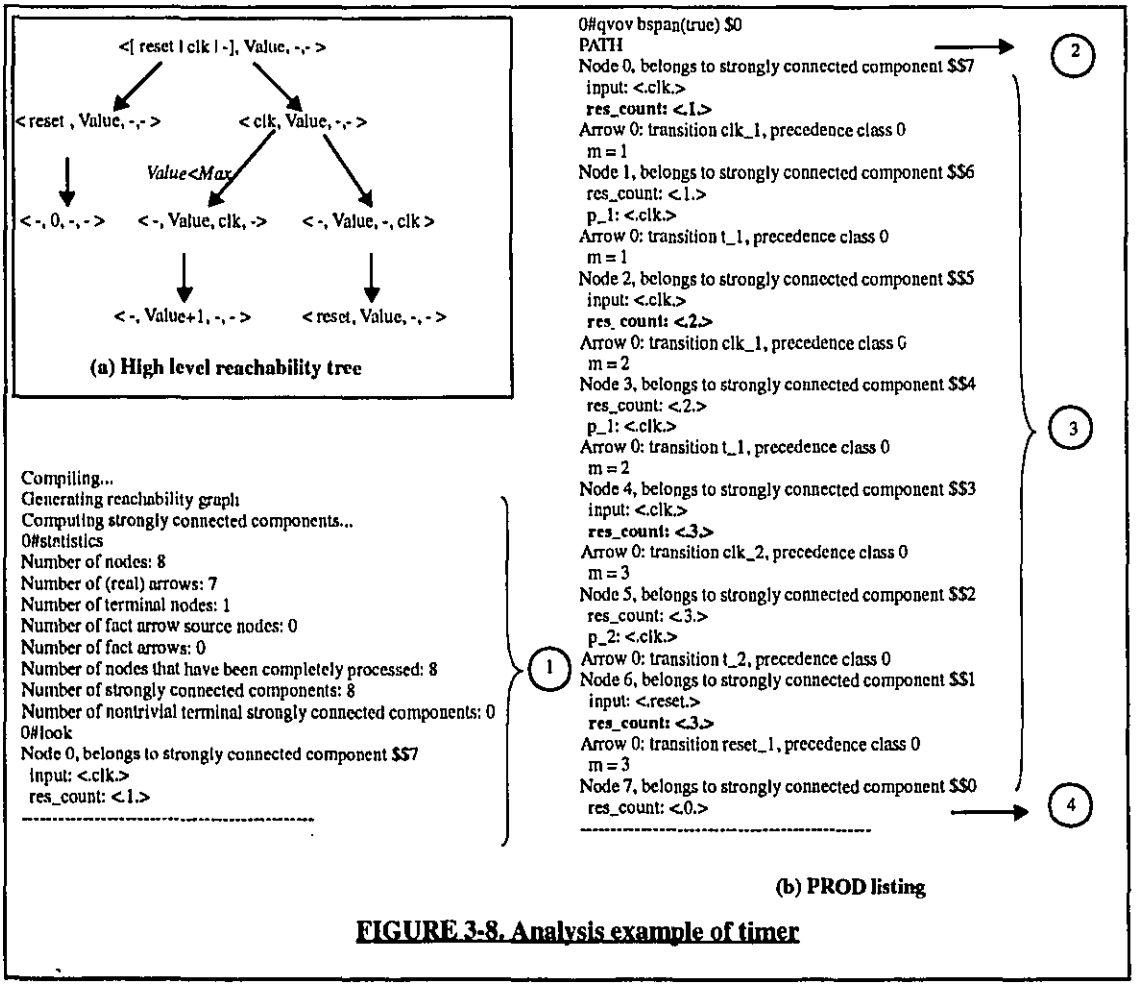


FIGURE 3-8. Analysis example of timer

analysis has then reached a terminal node, which was the initial marking of the graph. Hence, the behavior of the timer, corresponds to what one would expect.

3.3.2 Higher-order analysis

Analysis of higher-order modules is not as straightforward. Such models can contain unbounded places and very-large state spaces making analysis very difficult. DASE does not support explicit PrTN analysis at a higher-order module representation. Analysis at this level is executed informally with simulation (explained in the next section).

3.4 DSL simulator

The DSL simulator provides a means for experimentation and simulation of the DSL models. The simulator interacts with the DSL processor and library, and manages the scheduling and exchange of messages between modules, the update of a virtual simulation clock as well as ensure no constraints are violated.

The simulator utilizes several data structures to schedule messages (shown in figure 3-9). Messages are scheduled and maintained in message_queues created by the environment for each module. The queue is a list of tuples consisting of the message to be executed at the module and its creation time (to be used for statistics and constraint checks).

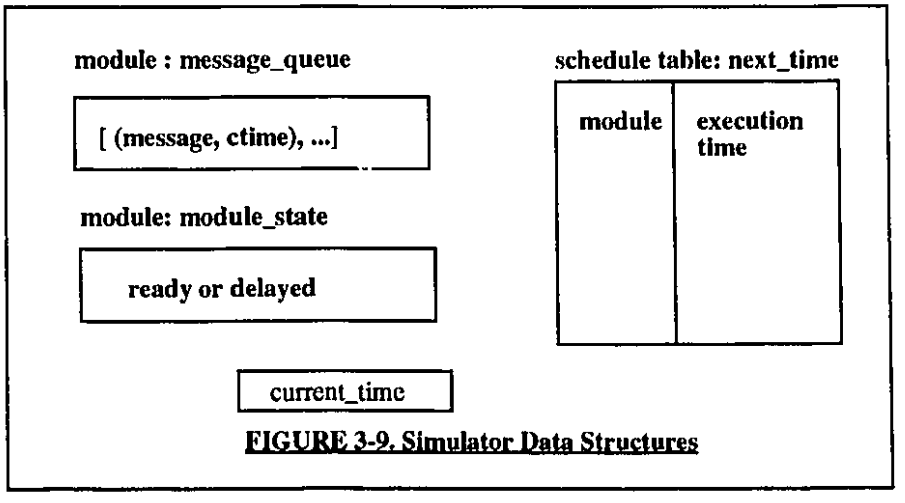


FIGURE 3-9. Simulator Data Structures

The simulator also defines modules to be in a "ready" or "delayed" state which is maintained in a variable module_state. In the ready state, the next message in the module's queue can be processed, whereas the delayed state indicates that the module is delayed until a specified time period. This time period is maintained in a schedule table named next_time. The simulator schedules the messages based upon this time value.

The simulator proceeds through several steps to schedule a message:

i. When a send statement is executed as part of a module's behavior, the simulator automatically checks the next_time table to determine if any messages are scheduled for the destination module. If no messages are scheduled (a negative time entry in the next_time table), the execution time for the module is updated to the current_time. In either case, the message and its creation time (current_time) is appended to the message_queue for the destination module.

ii. The simulator will then process all messages with execution time equal to the current_time. The processing involves removing the message from the respective message_queue and executing it through the DSL processor. If the message was in a delayed state, then the delay statement within the behavior is bypassed allowing for the continuation of execution of the module's behavior and the module_state is set to ready. If the module was in a ready state and a delay statement is encountered, then the simulator places the module in a delayed state and updates the respective time in the next_time entry.

iii. During each behavior execution, any constraints associated with the module is checked. If a constraint is violated, the simulator will identify the violation to the user. Included in the message sent to the user is a list of possible alternative library modules of the same class. This is retrieved through the lib_def file maintained by the DSL processor. The designer can then choose to possibly alter the design with the replacement, relax the constraint or ignore the constraint.

iv. The simulator determines the next highest time possible in the next_time table and sets the simulator clock (current_time) to it. The previous step is then repeated.

The model hierarchy created during library invocation is also utilized by the simulator to aid the designer through design exploration. An observation level is defined by the simulator which represents the level of detail different modules define. For example, the root of the model hierarchy is labelled as level 0 - which represents the most abstract module. The next level down would consist of modules that are referenced and inherited by the topmost module - and similarly for subsequent levels.

Abstracting hierarchy has often been employed to facilitate simulation and modelling at various levels of detail [Zeigler 84]. For example, a system level analyst may not be interested in the lower level details of a given model, whereas a hardware designer would be more concerned with the lower timing details of the simulation. The DSL simulator provides for dynamic alteration of the level of abstraction viewed by defining an observation level. The level can be defined to be limited to a given level or at a particular level and all levels below it.

The observation level can be set at any time in the simulator with the *set_level(LEVEL)* command. By default, the simulator is initiated to display all levels of information. The simulator also utilizes a list command to provide snapshots of the state of the system. This command is a menu driven form which permits viewing of resources, modules and higher-order modules, states of the simulator data structures, and the interconnections of modules. All of the views can be constrained to a particular level or range of levels of observation. The typical output of the command is shown in figure 3-10. The level of the observed module or data structure is displayed at the first column. The sample shows four particular views of the state of the simulation. The first is a display of the hierarchy. As an example, a module *delivery_q(1)* is defined to inherit behavior from another module (*delivery_queue(1)*). The second and third views show the state of the message queues and resources for each module. A module *clock(1,3)* is indicated as ready (a delay time of *0.0* is shown in the right-most column) to process a new message (*clock_count(44)*) from its

```
Hierarchy listing
delivery_q(1)        --> delivery_queue(1)
priority_q(1)        --> priority_queue(1)
aclock(3)            --> generic_clock(3,2,44)

Listing state of the module queues:
LEVEL   Module:          Messages queued:          Delayed until time:
2       clock(1,3)       [(clock_count(44), 0.0)]  0.0
2       clock(1,2)       [(clock_count(44), 0.0)]  0.0

Listing Resources Created:
LEVEL   Module:          Resource:        Variable(s):
4       priority_q(3)    p_queue(1)       []
4       gbcast(2)        label_count      1

Listing Path interconnections:
2   output(4,1) is connected to:
      --> output(4,1)    Port pair: [in,out]
      --> switch(4)      Port pair: [out_port(4,1),outline(4,1)]
      --> buffer(4,1)    Port pair: [queue_port(4,1),out_port(4,1)]
      --> atm(4,_764)    Port pair: [out_port(4,1),outline(4,1)]
```

FIGURE 3-10. Sample Output of List Command

message queue. The final view is a description of the interconnections between the modules.

3.5 Synthesis

When the designer is satisfied with the simulation results, the system's synthesis may be attempted. A translator within the DASE environment translates the DSL constructs into concurrent entities in VHDL under user guidance. The synthesis process within DASE can be partitioned into pre-synthesis support, DSL model code parsing and finally code translation.

3.5.1 Pre-synthesis support

During the course of design exploration and simulation, the abstract DSL model may be refined to a more detailed structure. For example, ports may be identified by the environment and created on demand (these ports are labeled as *dport(N)* where *N* is an increasing integer value). To synthesize the DSL model to VHDL, the following pre-synthesis requirements must be met by the DSL model:

Requirement 1: all undefined communications paths within an initial DSL model must be resolved. This implies that all undefined ports within send statements have been instantiated by the environment.

Requirement 2: all message types have been resolved for ports. This essentially indicates that all messages transmitted by send commands have been identified to respective ports. The synthesis process requires this information to correlate the port types to VHDL signal types.

Requirement 3: the environment has defined all destinations for DSL send statements. The environment can deduce most connections from the path statements

Requirement 4: multiple connections from output ports must be resolved. The implication is that DSL send statements with multiple destinations of the form send([b,c,d],aout,message) must be handled by the environment. The issue during synthesis with such communication is that the output port is used to transmit directed messages (messages where the destination is given). This implies that a separate signal must be used in VHDL or a bus protocol (utilizing the destination as part of the address resolution) to realize the communication.

The chosen solution within DASE is to identify a unique port name for each destination sharing the same output port of the source module. This is shown in figure 3-11. The illustration on the left depicts three modules connected to the same output port of module 'a'. Such a connection is interpreted by the DASE synthesizer as the diagram shown on the right within figure 3-11. The output port name is appended with the respective destination names and instantiated as three separate ports. This operation is performed during synthesis whenever a multiple destination send statement is encountered (hence the importance of requirement 3).

Requirement 5: multiple connections to input ports must be resolved. This requirement

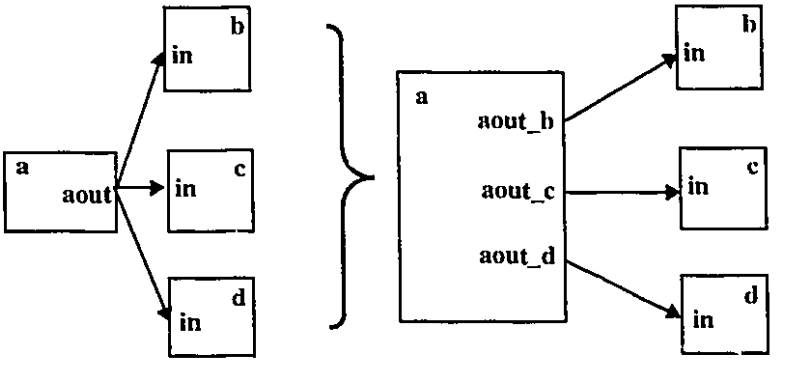


FIGURE 3-11. Output Port Preparation for Synthesis

can be satisfied in two ways. The first is to ensure a unique input port name for each path statement. The second is to introduce a *protocol module*. This module is conceptually different from normal modules. It does not get synthesized to a VHDL entity or process, but defines communication protocols between modules. This is a natural step when modelling system utilizing protocols for bus arbitration or shared mediums. A module is tagged as a protocol module simply within the predicate *module_type(Module_Name, protocol).*

The synthesis process is invoked through the *synthesis* command within DASE. The environment utilizes various data structures to effectively synthesize a design to VHDL. The relevant support information required by the environment to proceed through a successful synthesis is shown in figure 3-12.

The environment maintains a list of module names that are to be synthesized. The list is extracted from the *module_type(NAME, hardware)* predicates (see chapter 2.3). The predicate identifies that the respective module will be implemented as a hardware component. DASE will then extract all pertinent predicates regarding the module(s) into an intermediate file (file.dsl) as guidance to the parser. The predicates are created by the *synthesis* command and can consist of information such as:

i. The complete module behavior which is obtained directly from the module definition and the *isa* statements.

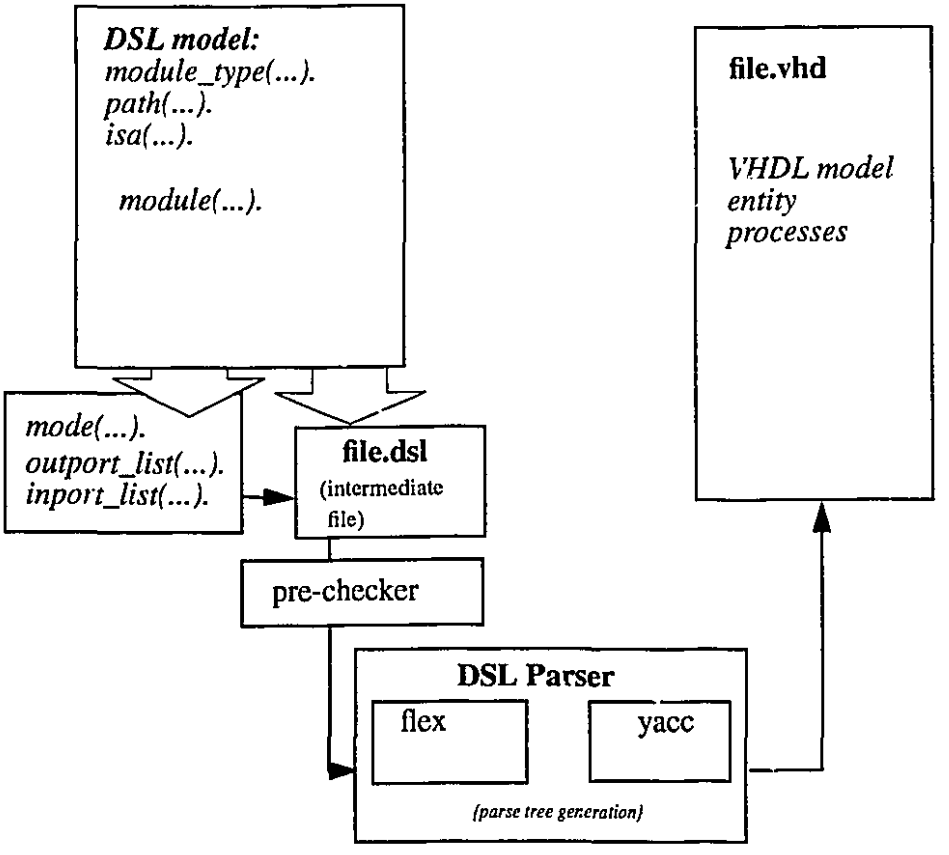


FIGURE 3-12. The Synthesis Process

ii. The *outport_list(Module, OUTPort, [Behavior])* which identifies all messages that the *Module*'s output port *OUTPort* will support. The pertinent information is extracted from the *send* statements within the module and added to the *behavior* list.

iii. The *inport_list(Module, INPort, [Behavior])* is used to identify all the messages that can be received (*Behavior*) through *Module*'s input port *INPort*. This information is obtained through a comparison of the possible sources of messages to a module (found by an examination of the path statement) and the corresponding messages that can potentially be delivered to the respective port (this information is already available from the *outport_list*).

iv. All *path* statements related to a module.

A pre-check predicate verifies to ensure that all the requirements for synthesis to proceed are satisfied.

3.5.2 DSL to VHDL Parser

The file.dsl is input to a DSL parser. The parser is built using the UNIX flex and yacc utilities. As the input file is parsed, a parse tree (represented in a linked list structure) is created within the parser. The data structure is shown in figure 3-13. Each branch of the tree identifies the VHDL equivalent of DSL actions, variables, branching conditions and other model information. Not all fields are filled during the initial parse of the input. A second pass of the tree entries is required to establish the VHDL counter-parts to the DSL code. The parse tree is then traversed to generate the corresponding VHDL model.

The nodes of the parse tree contain three information fields. The first identifies the type of node. This can be an entity (module name), a command (such as set_res, create), signal (identified VHDL signal names), type (identified types), statement (any branching information) and condition node (any conditions before attempting commands). The other

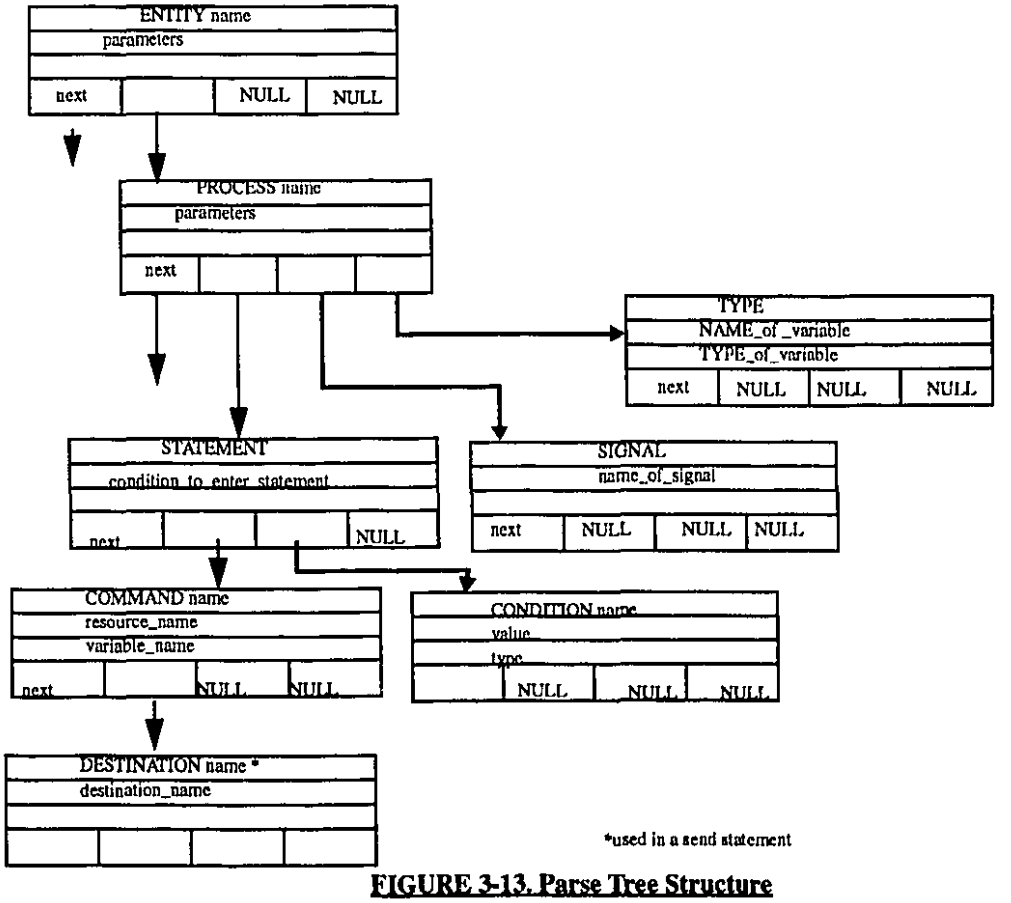


FIGURE 3-13. Parse Tree Structure

two information fields are used to store the relevant data with the corresponding node. Each node also possesses four pointers. One of these is dedicated to point to the next node, however others are used as warranted to indicate other nodes that hold information regarding the particular DSL statement's parameters.

During the course of the construction of the parse tree, the data within nodes is modified as the complete DSL code is parsed. The tree is then traversed applying translation rules (explained below) to complete the tree and produce VHDL code.

3.5.3 DSL to VHDL translation

The parser maintains a set of default variables and conventions during synthesis. It also synthesizes to a subset of VHDL, since commercial VHDL synthesizers cannot handle the full VHDL language. The applicable subset of VHDL will vary with each synthesis tool. The corresponding VHDL statements presented in this section is generally acceptable in most cases, however some tools may not support a particular form. In this case, additional attention (such as further translations) may be needed before use with the particular tools.

The translation of DSL to VHDL representations is not straightforward and can proceed in two different modes depending upon the intended description.The particular mode of synthesis is maintained within the *mode(Module, Type)* predicate where the *Type* variable is either a 1 or 2, depending on the desired synthesis mode for *Module* as described below:

Mode 1: By default, the environment will assume that causality of messages arriving to modules are satisfied by the DSL design. Most synchronous DSL designs will satisfy this assumption. In such a case, translation from DSL to VHDL follows direct rules where DSL ports and messages translate to VHDL signals and values respectively. Each DSL module corresponds to a VHDL process and DSL behaviors map to VHDL statements within a process. State or variable values and types are also defined through the DSL environment. The basic algorithm for this mode of translation is given below:

step 1. For each DSL higher-order module:

begin h-o-module,

create a VHDL entity header and port definition,

step 2. For each DSL module:

begin module,

step 2.1. create a VHDL process (named after the DSL module),

step 2.2. create the VHDL process sensitivity list,

step 2.3. translate each DSL behavior to corresponding VHDL behavior,

end module,

end h-o-module

The details of the algorithm are presented below for each of the steps:

Step 1:

The first step establishes the entity declaration. The higher-order module name encapsulating the module(s) is used as the entity name. VHDL port declarations are defined through the DSL path statements.

Step 2:

This step constitutes most of the behavior translation activities. The first substep (2.1) creates a VHDL process with a label corresponding to the DSL module name. This is a straightforward procedure shown within the translation table (table 3-1). Step 2.2 establishes a sensitivity list for the corresponding VHDL process. The list is defined through the *inport_list* predicate. Every input port of a module corresponds to a signal within VHDL

The final step (2.3) is a translation of DSL actions to VHDL statements. Table 3-1 summarizes all the corresponding translation rules employed by the parser.

Mode 2: For asynchronous design components, the user may direct the environment by setting a predicate within the simulator called *mode(MODE). MODE* establishes

the mode to use during parsing. This can be "1" or "2". The latter indicates the desire to synthesize an asynchronous design. This has the effect of indicating that the implied causality and ordering of DSL messages should be maintained in the VHDL model. In such a case, the corresponding VHDL model will include an input buffer and control for each entity so that the order of arriving signal values (which correspond to DSL messages) are maintained. Each signal value within the buffer in turn are processed by the VHDL entity. Such designs may be costly in terms of hardware, but VHDL post synthesizers can optimize it further.

Id	DSL Statement	VHDL Statement
declaration	outport_list(Module, OPort(o1,...,om), [Behavior(a11,...,a1n),..., Behavior(aj1,...,a1ij)]). inport_list(Module, IPort, [Behavior(a11,...,a1n),..., Behavior(aj1,...,a1ij)])	/* within declaration of Module: OPort_o1..._om_type is ('Behavior(a11,...,a1n)',..., 'Behavior(aj1,...,a1ij)'); IPort_o1..._om_type is ('Behavior(a11,...,a1n)',..., 'Behavior(aj1,...,a1ij)'); signal OPort_o1..._om: OPort_o1..._om_type; signal IPort_o1..._om: IPort_o1..._om_type;
	module(Module(p1,...,pl), [{behavior}]).	Module_p1..._pl: process(
behavior	(Behavior(a1,...,an) :- {conditionals and action})	if in1 = behavior(a1,...,an) then {conditionals and actions} endif;
conditional	check_res(Resource(r1,...,rz),(v1,...,vy)), {actions}.	type Resource_type is record v1 : VALUE1; v2 : VALUE2; ... vy : VALUEy; end record; variable Resource_r1..._rz : Resource_type; if Resource_r1..._rz = (v1,...,vy) then {actions} endif;
action	delay(DELAY)	used during translation of DSL send statement below
action	set_res(Resource(r1,...,rz),(v1,...,vy))	Resource_r1..._rz := (v1,...,vy);
action	create(Resource(s1,...,sz),(v1,...,vy))	not permitted
action	remove(Resource(s1,...,sz),(v1,...,vy))	not permitted
action	send(Dest(d1,...,dn), OPort(o1,...,om), Behavior(aj1,...,aij))	OPort_o1..._om <= Message(aj1,...,aij) after DELAY;
action	send([Dest(d1), ...,Dest(dn)], OPort(o1,...,om), Behavior(aj1,...,aij))	OPort_o1..._om_Dest_d1<=Message(aj1,...,aij) after DELAY; ... OPort_o1..._om_Dest_dn<=Message(aj1,...,aij) after DELAY;

Table 3-1: DSL to VHDL Translation Rules

The algorithm for the second mode of translation is slightly different, so as to ensure the desired timing requirements. The first step of the algorithm is identical to that described for mode 1 type synthesis. The differences are within step 2. The algorithm for step 2 is:

step 2. For each DSL module:

begin module,

step 2.1. create a VHDL process "MODULE" named after the DSL module,

step 2.2 create a VHDL process "queue_MODULE"

step 2.3. create the VHDL process sensitivity list for "queue_MODULE",

step 2.4. sensitivity list for process MODULE is: signal_MODULE,

step 2.5. create FIFO queue code for queue_MODULE, and MODULE.

step 2.6. translate each DSL behavior to corresponding VHDL behavior for MODULE,

end module.

Essentially what is obtained by the algorithm are two VHDL processes per DSL module (shown in figure 3-14). One process is sensitive to incoming signals (as in the case for mode 1 processes), however its behavior represents a first-in-first-out (FIFO) queue. Hence any new signal values is stored within the queue structure represented within the process. A signal named *signal_MODULE* is used to communicate with the second process. Another signal (*ready*) is an input from the *MODULE* process. When the *ready* signal is asserted, the *queue_MODULE* places the next value of *signal_MODULE* from its FIFO.

The *MODULE* process is sensitive only to the *signal_MODULE* signal. When a change on this signal is detected, the value is obtained and treated within the body of the process as in the case for mode 1 translation. At the same time the ready signal is not asserted indicating that the process *MODULE* is busy processing a "message". The signal is asserted again to obtain the next message (if any).

The combination of the two signals ensures that incoming messages are correctly ordered for the process *MODULE*. Figure 3-14(b) presents the typical timing waveform encountered for the two processes. The first two waveforms represent the *ready* and *signal_MODULE* signals (in this example the *MODULE* post-fix is

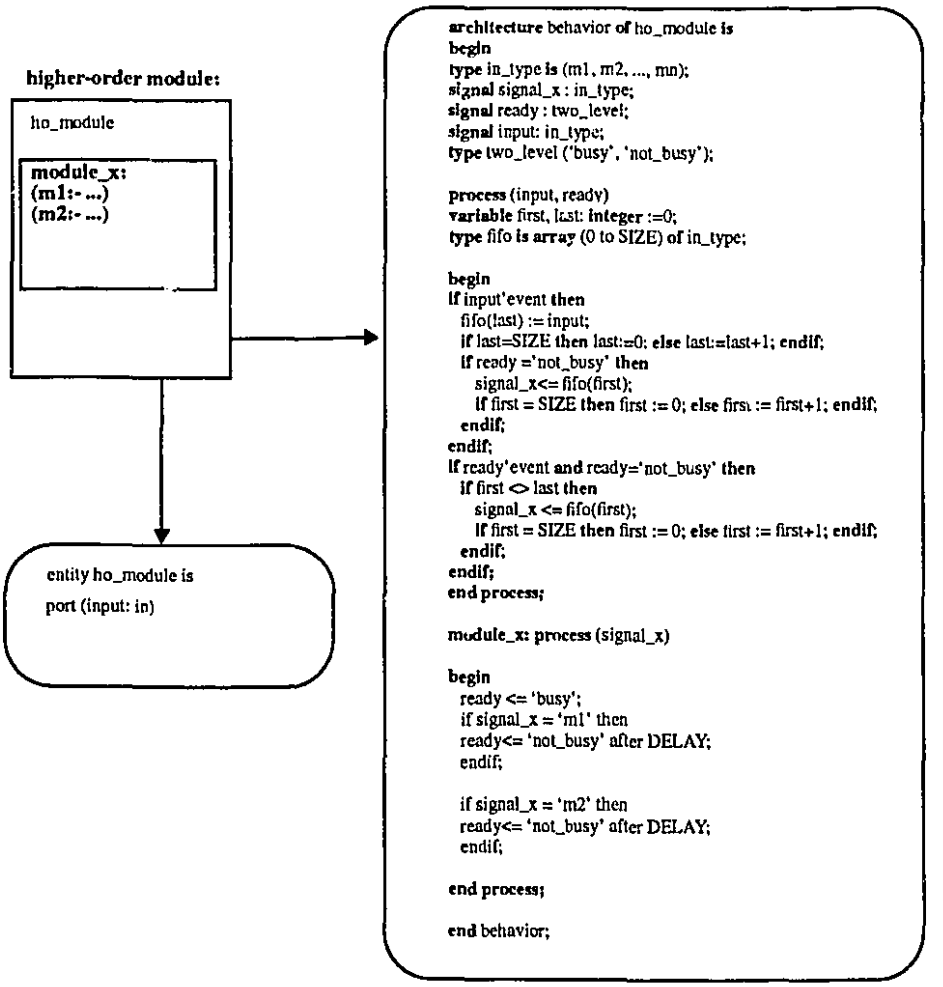


FIGURE 3-14. Mode 2 Translation

a). The *signal_a* is an enumerated type (m1, m2, m3, m4 in this example). The third waveform indicates the arrival of new signal values detected at the *queue_a* process. The ready waveform indicates the order in which the incoming signal values (or DSL messages) have been processed.

The amount of attention given to timing considerations within DSL can significantly impact the eventual VHDL code. If possible the designer should incorporate as much of the

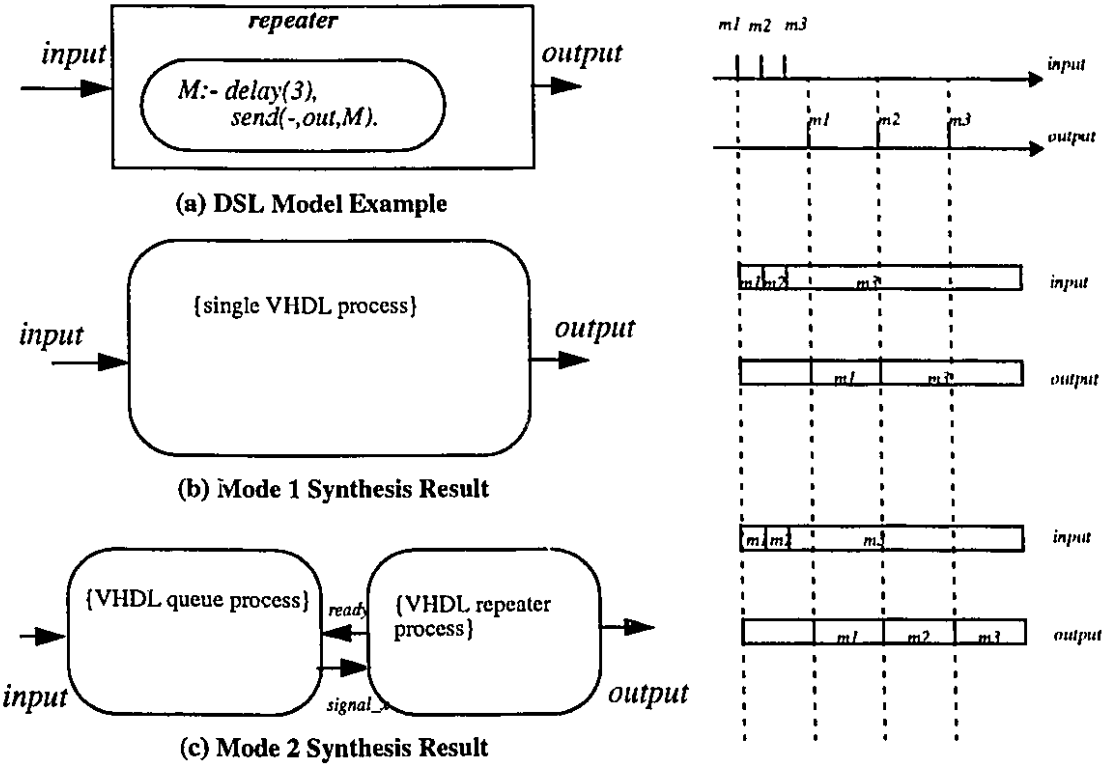


FIGURE 3-15. Synthesis Timing Example

timing relationship within the DSL model (especially for re-usable generic modules). This will tend to produce less overhead within the synthesized VHDL code. An example is given below to help clarify the modelling issues.

Consider the example presented in figure 3-15. A DSL module *repeater* is presented with one input port *input* and one output port *out* specified. The behavior of the module is simple, it sends any message arriving at its input port onto the output port after a delay of 3 time units. A stream of incoming messages are assumed to be sent to the module as shown in figure 3-15(a).

The first example shows a synthesis of the DSL module in mode 1. The resultant VHDL code does not produce the exact same timing characteristics for the input message stream. The problem is that the m3 value overwrites the m2 value before the VHDL process can detect it. Figure 3-15(c) depicts the second mode of synthesis, ensuring the correct ordering of messages. However it can be observed that the overhead within the VHDL model is

significant compared to the first mode. Further optimization could be attempted at this point with lower level synthesis tools. The other alternative to guaranteeing the same timing characteristics between the two representations is to enforce the timing discipline within the DSL model. In this example, a possible solution is to guarantee that the message m2 or m3 does not arrive before m1 is finished processing.

3.5.4 DTSS example revisited - experimentation

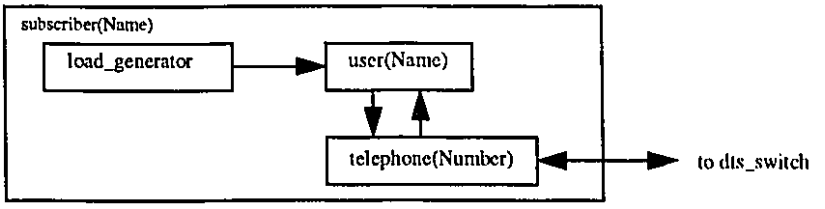
A design model is as good as the experimentation it supports. Different operational scenarios or configurations help refine and study the overall behavior of the modelled system. Design experimentation requires the definition of two models. The first is a model of the system under consideration. Such a model is studied and different sections are eventually synthesized to hardware. The DSL modules presented all fall under this category. The second type of models that are required are those that emulate some key aspects of the environment that the system under consideration will work in. Such models can be traffic generators, failure/fault injectors, interrupt generators etc. These provide the stimuli to test and validate the models - utilized for experimentation support.

This subsection will use the DTSS example to demonstrate and observe a typical telephone call connection within a certain amount of time as a test of basic DTSS functionality. Hence the environment will require a model of a typical telephone and user as well as a traffic generator for background telephone traffic. The user module will posses a simple behavior that: lifts up the handset, waits for dial tone, dials a number, talks for a specified random time period (if the connection is made) and hangs up.

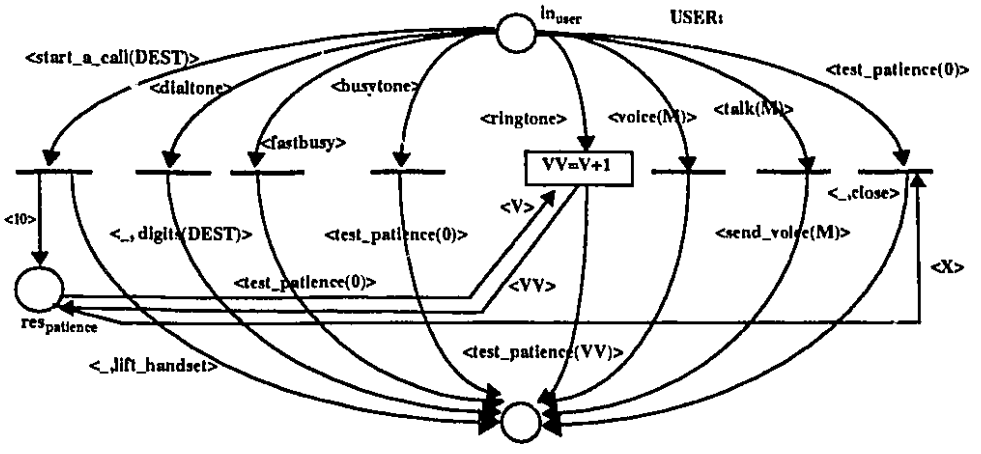
The telephone module will act as the interface between the user and switch by performing operations such as; sending the dialed digits to the switch, providing the different tones to the handset (user feedback), and allowing voice messages to pass to and from the handset. A load_generator module is also used to initiate a request for a call to a user module at random intervals to random destinations. The structural representation of the environment model is represented in figure 3-16 (a). Figure 3-16 (b) provides a predicate/transition-net

representation of the behavior for the telephone module and user modules.

DASE provides simulation support for design exploration. Once the main DSL program is loaded into the environment, all library modules are also loaded, configured and



(a): DSL Model



(b) PrTN Model for Telephone Module

FIGURE 3-16. Representation of Environment Elements

instantiated. The DSL simulator automatically creates an internal record of the hierarchy tree. This tree is used to define the observation level for the model during simulation. The hierarchy tree for the DTSS is shown in figure 3-17.

By default the observation level is set to depict the maximum detail - which corresponds to level 4 in this example. However if the telephone call-setup is of interest, which is a relatively high level view of the system, then the internal nuances of the switch should be hidden from the user so that there is not an overload of information. This implies setting the observation level to the first level with the statement *set_level(1)*. Based upon the hierarchy tree, this level will restrict the simulator output to only messages passing between the switch and subscribers.

The observational level can be changed at any time during a simulation - allowing the user to zoom in or out as required. The library modules may also contain associated internal constraints. For example some telephony practices dictate a maximum call delay (time from when a call is dialed till ringtone is heard) of 450 microseconds. This time is incorporated as a constraint (upper bound) for the ringtone message to be delivered to the

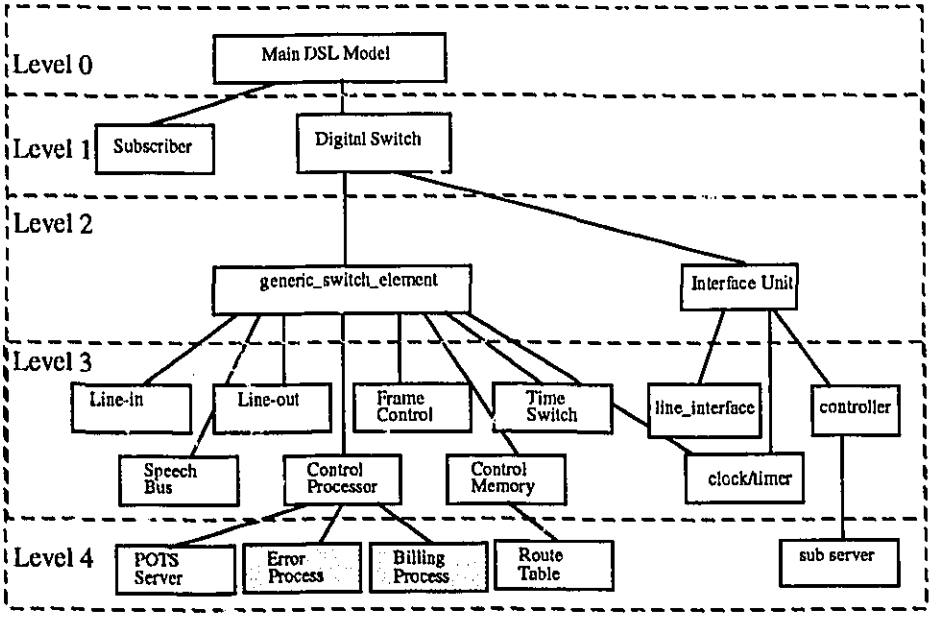


FIGURE 3-17. Hierarchy Tree of DTSS Example

destination. A sample snapshot of a simulation run is shown in figure 3-18.

The listing traces the messages for subscriber(0) who initiates a call to subscriber(5). Part 1 of the listing shows the level 1 messages observed by the simulator. These include lifting of the handset, dialtone, dialling the destination number and the related messages to the

PART 1

```
DSL SIMULATOR: Observable level set to 1

----> CURRENT SIMULATOR TIME is 0.0 microseconds.
==========================================================
----------------------------------------------------
DSL SIMULATOR: Scheduling message from subscriber(0) via subscriber(0) to telephone(0)
Source Port: hand(0) Message is: lift_handset
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(0) via mcgill_switch(0) to controller(0)
Source Port: tel_line(0) Message is: phone_off_hook(0)
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(0) via telephone(0) to subscriber(0)
Source Port: headset(0) Message is: dialtone
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from subscriber(0) via subscriber(0) to telephone(0)
Source Port: hand(0) Message is: digits(telno(0,1,5))
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(0) via mcgill_switch(0) to controller(0)
Source Port: tel_line(0) Message is: dial_dest(0,telno(0,1,5))
```

PART 2

```
----> CURRENT SIMULATOR TIME is 400.0 microseconds.
==========================================================
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(0) via telephone(0) to subscriber(0)
Source Port: headset(0) Message is: ringtone
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling INTERNAL message from subscriber(0)
Message is: test_patience(9)
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(5) via telephone(5) to subscriber(5)
Source Port: headset(5) Message is: ringtone
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling INTERNAL message from subscriber(5)
Message is: talk(hello)
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from subscriber(5) via subscriber(5) to telephone(5)
Source Port: hand(5) Message is: lift_handset
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from subscriber(5) via subscriber(5) to telephone(5)
Source Port: hand(5) Message is: send_voice(hello)
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(5) via mcgill_switch(0) to controller(1)
Source Port: tel_line(5) Message is: phone_off_hook(5)
----------------------------------------------------
----------------------------------------------------
DSL SIMULATOR: Scheduling message from telephone(5) via mcgill_switch(0) to controller(1)
Source Port: tel_line(5) Message is: voice_message(5,hello)
----------------------------------------------------
```

FIGURE 3-18. Sample Simulation Output for Call Set-up

switch from the telephone. No other level 1 message is observed in the simulation run until 400 microseconds later. The output for this time period is shown in part 2. A ringtone is heard on subscriber(5)'s telephone as well as the caller's, the phone is answered and a "hello" voice message is sent to the originator. Hence the phone call is established.

In this example, the call delay is 400 microseconds, hence it is within the defined system constraint. This constraint is used to understand and set some performance bounds for a system level parameter (or metric). Lower design related parameters are also a part of the model. A significant design constraint is placed upon the control processor in the generic switch element. It must be able to process new incoming data before they are overwritten. Hence there is a constraint imposed upon the control processor (in particular the frame_select message must be processed before a certain upper bound). If the number of channels per line were increased to 32 and the number of interfaces units attached to the switch to 20, then the demands upon the control processor are greater. The defined control processor will not be able to keep up with the incoming traffic stream and a message will be issued from the simulator such as:

DSL SIMULATOR: WARNING: Upper_limit Constraint Violation!
Module: control_processor(0) Message: frame_select(15)
Constrained Variable 0.1 exceeds upper limit of 0.051

Accordingly, the user can decide upon the course of the design exploration. Options include the traditional design decisions such as replacing the control processor with a faster alternative or adding multiple processors. Alternatively, the design constraint can be relaxed or ignored to view the impact upon the overall system behavior.

As in all modelling environments, the ultimate design decisions are directed by the user. Hence the designer's modelling approach and style will effectively dictate the size of the resultant VHDL code. Some of the different modelling approaches are further illustrated in the next chapter as detailed case studies - providing system design and modelling solutions using DASE.

Chapter 4 - Case Studies

This section will present two case studies dealing with the application of DSL to different problems in design. The examples are complementary to the DTSS example developed in chapters 2 and 3. The studies are intended to present the generality and flexibility of the DASE environment within the restricted domain of telecommunication systems. Each study focuses on different capabilities of the environment. The studies provide different levels of detail and elaboration with respect to the DSL code - presenting detailed examples of behavioral representation of design entities.

The first study will elucidate the advantage of DASE in the design of an upcoming technology in switch design: ATM. The study is intended to provide insight to the DSL modelling process as it presents a model of an ATM switch and applies it to an ATM network. The model reusability is also illustrated with the incorporation of some modules from the first case study in the ATM design. A network simulation is presented in detail within the second case study.

The second study will detail the DSL modelling of three broadcast protocols used in the support of reliable distributed computing systems. This example provides the most detail of behavioral descriptions within DSL. After a detailed walk-through the design of the protocols, an example node model utilizing the protocols will be introduced. The section will conclude with a simulation of nodes communicating with the broadcast protocols through an ATM switching network.

4.1 ATM Switch Design

4.1.1 Introduction

The technology push in telecommunications has blurred the distinction between voice and data services. While most services today address voice and low speed data, future demands will require the transparent use of these and more demanding services such as high definition TV and high speed data across a common medium. Apart from placing significant performance requirements on a system, these services have significantly different characteristics. 50% of traditional voice traffic contains silent periods which translates to a burstiness factor of 2 (the ratio between the maximum and average information rate). This factor helps to determine the design requirements for a switch. Designing for the peak information rates implies wasted bandwidth at lower rates, while design for the average rates results in loss of quality in service during peak periods.The burstiness and information rates of a variety services are tabulated in table 4-1 [Prycker 91]. It can be observed that the demands upon the system to support the different services will vary from one service to the next.

Service	Burstiness	Information Rate (bits/sec)
Basic Voice	2	$6.4x10^3$
Interactive Data	10	10^5-10^7
Bulk Data	1-10	10^5-10^6
Standard Video	2-3	10^7
High Definition TV	1-2	10^8-10^9

Table 4-1: Typical Service Characterizations

Asynchronous Transfer Mode (ATM) has been proposed as a potential solution for incorporating constant and variable bit rate services such as voice and data under one transmission and switching solution. ATM is essentially a connection oriented data-link level protocol with its main characteristics being summarized as [ATM 93]:

1. No error control on a link basis. Hence some information may be lost. The assumption is that the transmission medium quality is good enough to keep such losses within acceptable

limits.

2. The information field in ATM is small. Information is transferred in small fixed length cells of 53 byte lengths of which 48 are reserved for information and 5 are header information.

3. No flow control on a link basis. Information may be lost due to overflowing queues within a switch. This risk is minimized by statistical allocation of resources. The information fields are also relatively small in ATM so that a momentary loss of information may not be a major impact on the service.

4. ATM is connection oriented. Hence a call setup phase exists between the service requester (an ATM terminal) and a service provider (the ATM switch).

5. Header functionality and size are minimal in ATM. This allows for fast hardware switching of information once a connection has been established.

An ATM cell for an ATM switch is shown in figure 4-1. The five byte header comprises of a VPI (Virtual Path Identifier),VCI (Virtual Channel Identifier), PTI (Payload Type Indicator),CLP (Cell Loss Priority) and HEC (Header Error Control). The VPI/VCI fields are the virtual path and channel identifiers that represent a virtual circuit the cell is

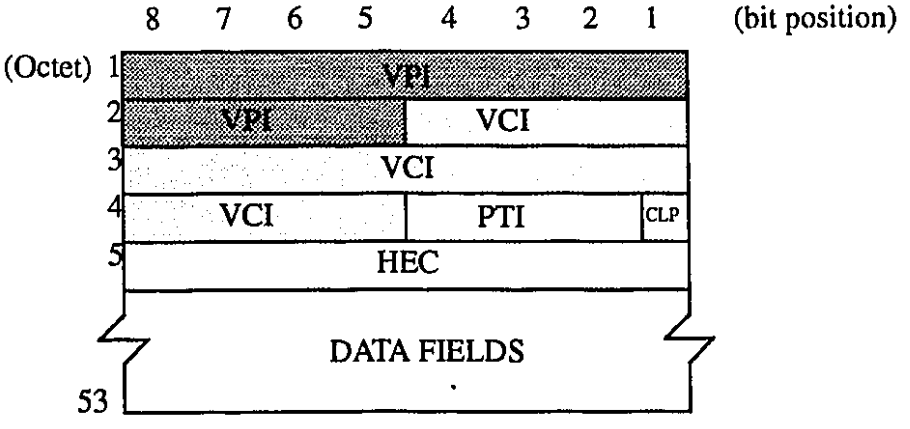


FIGURE 4-1. Basic ATM Cell Structure

traversing. A VCI is composed of a 16 bit address, hence each VPI can contain up to 64K VCIs. These addresses are local to the switch, hence a connection table within switches identify unique translations between incoming and outgoing VPI/VCI pairs. The PTI and CLP are used by the network for congestion control and identifying cell priorities. If the PTI indicates a significant increase in congestion on a line, variable bit rate cells are given a lower priority (with respect to constant bit rate cells). In the event of buffer overloading in the switches, the lower rate cells are discarded. The last header field is the HEC which provides for error control across the 4 bytes of header information.

It should be noted that for smaller ATM devices such as ATM hubs which multiplex a set of users across a single access point to an ATM switch, the 4 bits of the first octet are intended to be used for generic flow control between the hub and users. At present, there is no consensus within the industry with regard to the way in which these bits are treated, hence they are omitted in this example. ATM is initially expected to operate on a permanent virtual circuit mode. This implies that virtual paths and circuits are allocated ahead of time by the service provider and call-setup operations are not performed on a per call basis as for traditional voice switching. As the signalling aspects for ATM become clearer, ATM services will likely incorporate on-demand circuits [Wernik 91].

The basic operation of an ATM switch operating in a permanent connection mode is shown in figure 4-2. The switch contains three input and output ports. Definition of the permanent circuits are contained within a translation table in the switch (these would typically be provisioned by the service provider or administrator). The addresses within the table are local to the switch. The VPI/VCI address field of each incoming cell is examined by the routing control in the switch and used as a reference address to the translation table - where a new VPI/VCI address is identified along with the destination output port for the cell in the switch. The figure presents an example of a cell address translation (incoming address 40/ 75 to outgoing address 23/92). The cell is then routed (through a routing network) to the respective output port with the new VPI/VCI address until it finally reaches the destination.

There are many different design alternatives and decisions that can be considered in the

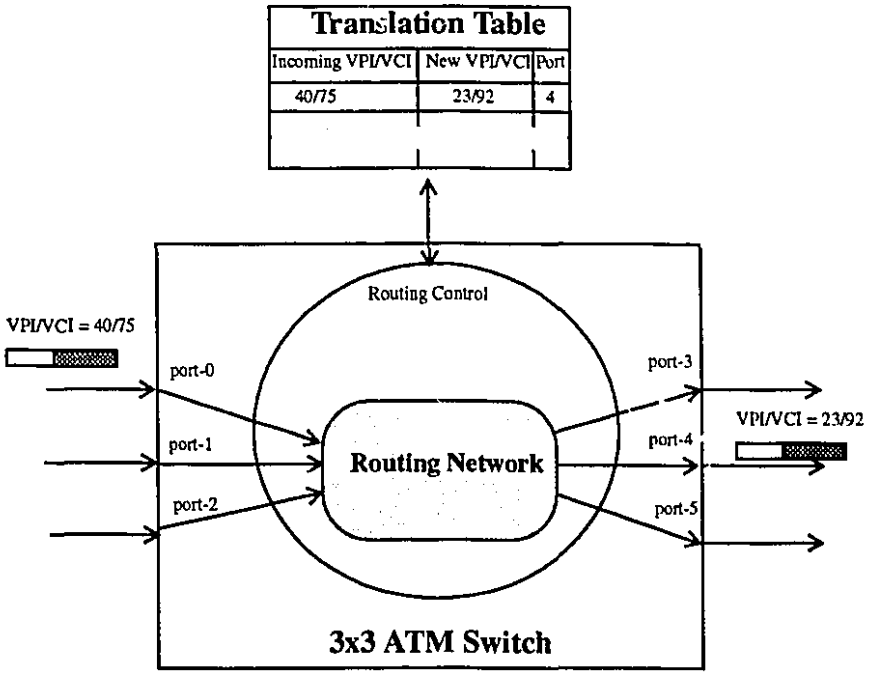


FIGURE 4-2. ATM Switch Operation Example

design of an ATM system suitable for fast switching of ATM cells [Zegura 93] [Bac 91]. This section will present an example of a permanent virtual connection oriented central queue based design with output buffering. The example is intended to demonstrate the decreased level of modeling complexity (as compared to the digital time-space switch) as well as the applicability of DSL in futuristic telecommunications applications.

4.1.2 The DSL model:

The ATM switch design will be introduced starting from the most abstract view - increasing in model granularity as the lower modeling levels are revealed (as in the case of the DTSS example). The delay parameters are omitted so as not to distract attention from the behavior. The evolution of the model is depicted within figure 4-3. At the top most level within a *main* model there is an instantiation of an ATM switch:

isa(atm(1), atm_switch(1,5)).

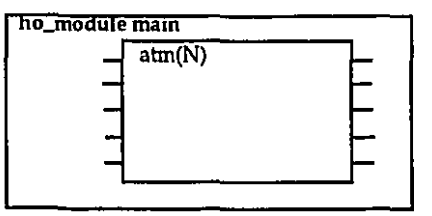
The above DSL statement indicates that *atm(1)* is an *atm_switch* with a parameter "5". The parameter in this case is the size (number of input/output lines) of the switch that is modeled.

The next stage depicted in figure 4-3(b) illustrates the different functionalities within the switch. The rational is similar to the DTSS design. There is a need for an input and output interface modules that will accept/transmit cells across a serial line. Since cells will be moving at relatively high rates, the design of these units should be simple and modular so that they can be synthesized readily into fast hardware. The output modules will consist of a buffer module, a clock module (which is re-used from the DTS design) and a transmission module. A control unit is needed to move incoming cells from the input modules to temporary buffers and a processor/memory module is required to manipulate the cell headers and route the cells to the appropriate output modules.

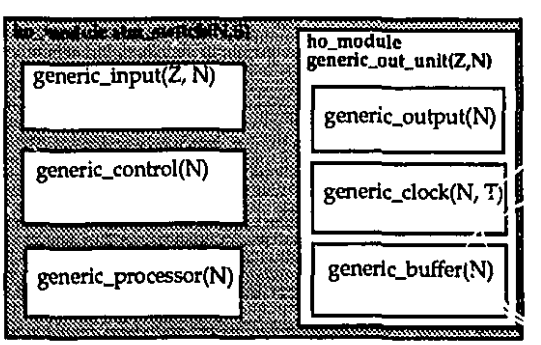
4.1.3 The Input module:

The input modules need to capture incoming cells into a single cell latch, signal the control module (*new_cell*) and send the cell to the module when signalled back (*read_rqst*) from the control module. Intuitively, the module should have an input port (*inline*) to receive the cells, and a port to interact with the control module (*control_port*). A typical description of

i.e. : isa(atm(1), atm_switch(5)).



(a) Instantiation of top level design



where Z = 1... S.

(b) Lower level library modules

FIGURE 4-3. ATM Modules

the module in DSL is:

```
module(generic_input(INDEX,NODE),
[
(new_cell(VPI, VCI, DATA):-
check_res(cell_buffer, (VPX, VCX, DX, unprocessed)),
     print("Lost an incoming cell at address (VPI/VCI) "),
     print(VPX), print("/"), print(VCX),
     set_res(cell_buffer, (VPI,VCI,DATA, unprocessed)),
     send(-,control_port(INDEX), new_cell(INDEX))
),
(new_cell(VPI, VCI, DATA):-
     set_res(cell_buffer, (VPI,VCI,unprocessed)),
     send(-,control_port(INDEX), new_cell(INDEX))
)
(read_rqst:-
check_res(cell_buffer, (VPX,VCX,DX,XX)),
     set_res(cell_buffer, (VPX,VCX,DX, processed),
send(-,control_port(INDEX), cell_info(VPX,VCX,DX))
```

Module Variables:
0=< INDEX < SIZE,
NODE = switch number -
the above two variables define a unique input module number

Module Resources:
name: cell_buffer.
Values: (V1, V2, DATA, FLAG)
where
V1 is the VPI address,
V2 is the VCI address,
DATA is the cell payload,
FLAG = (processed, unprocessed) indicates if data from the buffer has been successfully processed.

```
]]).
```

The first message that the module is capable of interpreting is the incoming cells *new_cell*. Two possible sets of behavior for *new_cell* are defined above. A condition is used to check if the cell in the buffer has been read, and if not, it is discarded and a message is printed to this effect. In the *new_cell* description, a new cell is stored in *cell_buffer* and a *new_cell(INDEX)* message is sent to the module's *control_port* (INDEX simply identifies the input module that is signaling).

The second message that is applicable to the module is *read_rqst*. This message initiates a message *cell_info* to the *control_port*. The intent is to model the reading of the buffer data. The *FLAG* variable of the *cell_buffer* is also set to *processed* (the cell data has been read).

4.1.4 The Control module:

The control module will require an input_port to interact with the input module(s) and also a port (proc_port) can be defined to communicate with the processor module. The behavioral description for the control module is relatively straightforward:

```
module(generic_control(I).
[
(new_cell(INPUT):-
        send(-, input_port(INPUT), read_rqst)
),
(cell_info(VPI,VCI,DATA):-
        send(-, proc_port(I), new_message(VPI, VCI, DATA))
)]).
```

The module accepts a *new_cell* message (from the input modules) and replies with a *read_rqst* message to the sender. It will then accept a cell_info message carrying the cell information. The cell now must be sent to the processor module for correct address translation and routing. This is accomplished with the *new_message(VPI,VCI,DATA)* message.

4.1.5 The Processor module:

The address translation and routing function is accomplished within the processor module. Traditionally the routing table would reside in a memory module accessible by the processor and other modules. In this example the memory component is embedded into the processor module to emphasize that the memory / processor interface is a fast one. This example assumes the permanent virtual connection oriented ATM, hence call-setup is assumed to be predefined. The routing information will reside within a *connect_table* resource in the processor module. This resource contains 5 fields: fields 1 and 2 define the incoming VPI/VCI pair, fields 3 and 4 define the new (outgoing) VPI/VCI pair to use, and field 5 defines the address of the physical output_unit to send the cell. The behavior of the processor module is represented below as:

```
module(generic_processor(I),
[
(new_message(VPI, VCI, DATA):-
        check_res(connect_table, (VPI, VCI, NVPI, NVCI, OUTLINE)),
        send(-, out_port(OUTLINE), atm_cell(NVPI, NVCI, DATA))
),
(new_message(V,VC,D):-
        print("ERROR: undefined circuit connection "),print(V),print(VC))
]).
```

Consequently the processor will forward an *atm_cell* message to the appropriate

output_unit module.

4.1.6 The Output_unit module:

The *output_unit* module is a higher order module consisting of three other modules: the *generic_buffer*, *generic_output* and *generic_clock*. The *generic_buffer* provides a simple FIFO queueing function for messages directed to a given *output_unit*. Consequently the module must accept *atm_cell* messages from the processor module and maintain them in a queue for processing. The DSL behavioral code is given below:

```
module(generic_buffer(UNIT,INDEX),
[
(next_cell:-
        check_res(queue, [(null, null, null) | Q])
),
(next_cell:-
        check_res(queue, [(VPI, VCI, DATA) | Q]),
        set_res(queue, Q),
        send(-, output_port(INDEX), next_info(VPI,VCI,DATA))
),
(atm_cell(VPI,VCI,DATA):-
        check_res(queue, Q),
        append([(VPI,VCI,DATA)],Q, NQ),
        set_res(queue, NQ))
]).
```

The *queue* can be depicted in terms of a resource. The top element of the *queue* is automatically removed through a *next_cell* message from the control module. This information is passed through the use of a *next_info* message from the buffer module. To reduce communication overhead, this message is only sent if there is a message waiting in the queue (no information is represented by the null entry in the queue).

The *generic_output* transmits ATM cells using cell data provided either by the *generic_buffer* module, or a "NULL" cell contained within an internal single cell buffer. All this is synchronized through the *generic_clock* module to generate the appropriate bit rates for cell transmission. The *generic_clock* introduced in the DTSS example in chapter2 is re-used here. The code for the *generic_output* module is listed below:

```
module(generic_output(UNIT,INDEX),
[
(clock(X):-
```

```
        check_res(out_buff, (VPI,VCI,DATA)),
        set_res(out_buff, (null,null,null)),
        send(-, out_port(INDEX), new_cell(VPI,VCI,DATA)),
        send(-, queue_port(INDEX), next_cell)
),
(next_info(VPI,VCI,DATA):-
        set_res(out_buff, (VPI, VCI, DATA)))
]).
```

The timing is such that the *generic_output* module is ready to transmit a message (*cell*) when given the *clock(X)* message from the *generic_clock* module. The typical high level ordering of key messages are depicted in figure 4-4. Intuitively, an ATM cell is being transmitted between each clock message. The clock messages are provided by the same *generic_clock* module described in the time-space switch example, however, since the timing of ATM does not require frame definitions, the *frame_sync* message generated by the *generic_clock* is not used. The transmission period (*T*) is dependent upon the transmission rate of the line. The CCITT recommends two rates for ATM: 155Mb/s and 622Mb/s for medium and backbone traffic respectively. Considering a medium traffic rate, this implies that each ATM cell takes 2.74 microseconds of transmission time (*T*). This time is expressed within the *generic_clock* module as *Clock_rate*.

4.1.7 Network model - module reuse

After definition of the specific modules, the higher order definitions provide the structural description of the underlying modules. The structural description of the switch is shown in figure 4-5. The figure shows only one incoming and outgoing line. For a switch of size N, the generic_input and generic_out_unit modules would be replicated N times. The depicted switch is capable of processing messages adhering to the ATM protocol.

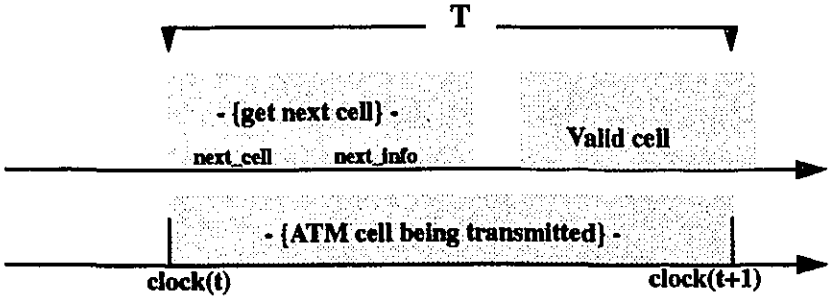


FIGURE 4-4. Key Message Ordering in generic_out_unit Module

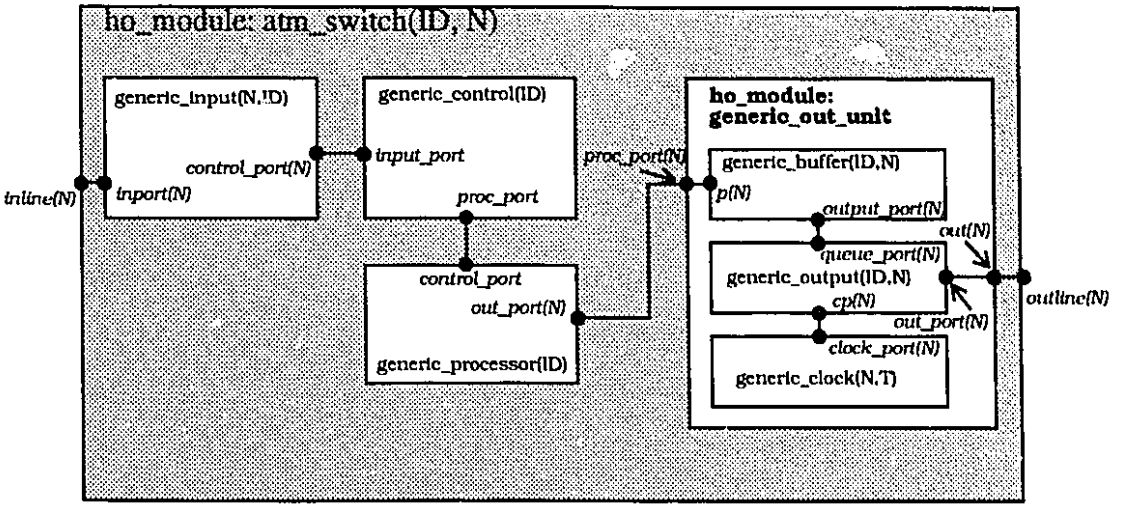
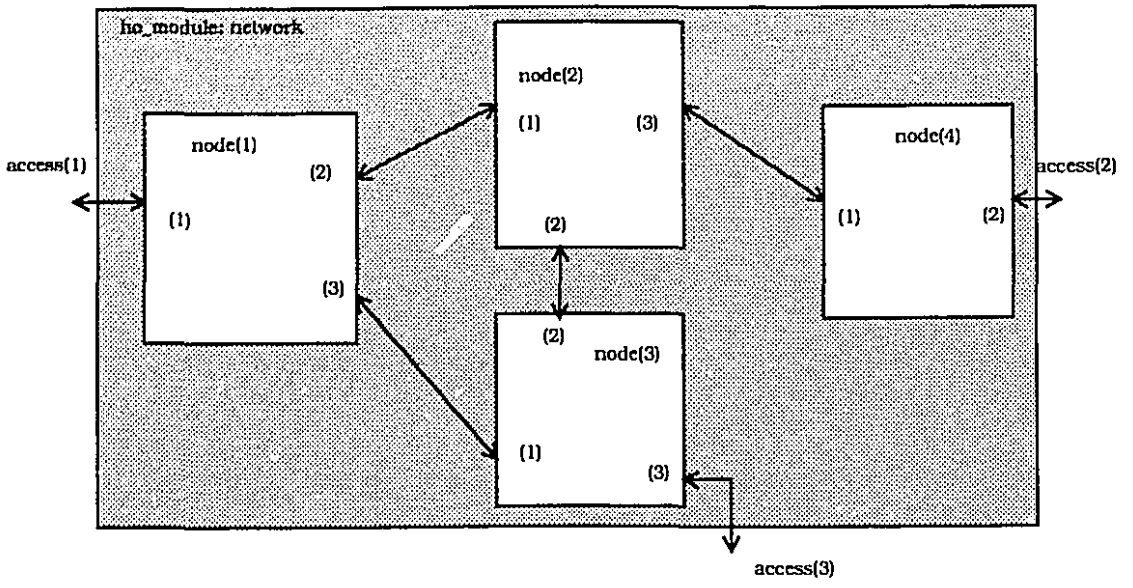


FIGURE 4-5. DSL Model of ATM Switch

The defined ATM model can be re-used to create a network of ATM switches capable of supporting routing and control of permanent circuits. The ATM model has already made use of the *generic_clock* module described in chapter 2 for the DTSS example. The generic_input modules are also a particular instance of the DTSS's line_in module. Further module re-use can be achieved if voice circuit emulation was desired over an ATM network. The ATM switch would then functionally be similar to the *generic_switch_element* of the DTSS example. The ATM switch would require an additional understanding of how to establish a demanded voice circuit. The desired behavior can simply be inherited by the ATM switch's *generic_processor* from the call processing software module (*pots_server*) described in the DTSS description in chapter 2. In this case, incoming ATM cells containing voice information will be handled by the *post_server* as in the DTSS example. Considering the lines of DSL code in the behavior of the ATM switch, approximately sixty-five percent of the ATM switch's behavior is comprised of re-used code available from the library.

A network composed of ATM switches can now be created for modelling different traffic scenarios. For example, figure 4-6(a) shows a network of 4 switches in a particular configuration and figure 4-6(b) lists the respective code for defining the network.



(a) DSL Network diagram

```
isa(node(1), atm_switch(1,3))
isa(node(2), atm_switch(2,3))
isa(node(3), atm_switch(3,3))
isa(node(4), atm_switch(4,2))

path(node(1), node(2), [outline(2), inline(1)])
path(node(2), node(1), [outline(1), inline(2)])
path(node(1), node(3), [outline(3), inline(1)])
path(node(3), node(1), [outline(1), inline(3)])
path(node(2), node(3), [outline(2), inline(2)])
path(node(3), node(2), [outline(2), inline(2)])
path(node(2), node(4), [outline(3), inline(1)])
path(node(4), node(2), [outline(1), inline(3)])

path(node(1), network, [outline(1), access(1)])
path(node(3), network, [outline(3), access(3)])
path(node(4), network, [outline(2), access(2)])
path(network, node(1), [access(1), inline(1)])
path(network, node(3), [access(3), inline(3)])
path(network, node(4), [access(2), inline(2)])
```

(b) Interconnection description of network

FIGURE 4-6. ATM Sample Network Configuration

The underlying network model is relatively complete. The only thing that is required is the routing (permanent virtual path connections) definitions between nodes. This is established by defining the respective *connect_table* resource in each nodes's processor module. Depending upon the assertions upon these resources, all possible combinations of virtual paths and circuits for the given network can be described.

A sample routing setup is described in table 4-2 where the entries for each node's connect_table is given. The defined set-up describes virtual connections between the nodes as shown in figure 4-7. Every VPI/VCI address of incoming cells are translated to defined outgoing ones at each node.The sample connection description defines three virtual connections. More connections can be defined by the user beforehand (using library configuration rules) or at run time using DSL *set_res* or *create* commands.

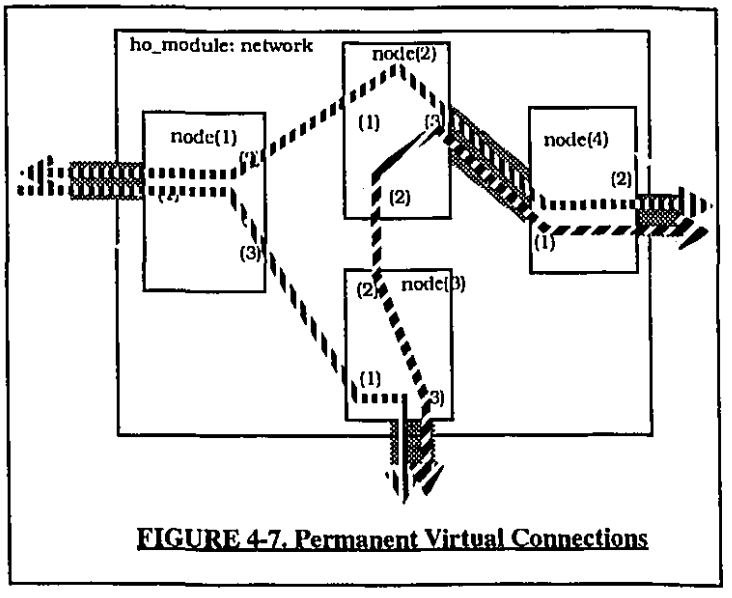


FIGURE 4-7. Permanent Virtual Connections

Node #	Incoming VPI/VCI	Outgoing VPI/VCI	Output Port #
1	1/1	2/1	2
1	1/2	3/1	3
1	4/1	1/1	1
1	5/1	1/1	1
2	1/X	4/X	1
2	2/X	6/X	3
2	8/X	7/X	3
3	3/1	4/1	3
3	1/1	8/1	2
4	6/1	2/1	2
4	2/1	1/1	1
4	7/1	1/2	2

Table 4-2: The Routing Table

As in the digital time space example, this model can be exercised using random traffic or traffic generation modules (such as a user or telephone). The ATM network model will be incorporated with the protocol models described in the next section to demonstrate a complex model of broadcast protocol executing over an ATM network.

4.2 Reliable distributed broadcast protocols

Protocols provide the basic formalism to support communication between computing agents in a distributed computing environment - hence protocol modelling is essential to the study of computer communication systems. This section presents a case study of the modelling of a set of broadcast protocols for reliable communications within a distributed environment. Typically an application will utilize several layers of protocol to communicate, hence it is generally desirable to identify different layers of functionality in models as well [Bochman 90]. The protocols to be modelled in this section (ABCAST, CBCAST and GBCAST) [Birman 93], are assumed to reside between higher and lower level services. Although these services are not modelled in detail, they provide and request the necessary services from the three protocols.The structure of a processing node using these protocols are shown in figure 4-8. The generic_protocol server directs the upper and lower layer requests to the respective protocols. The upper layer send requests for

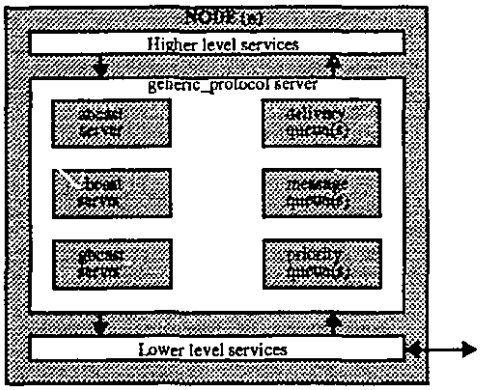


FIGURE 4-8. Structural View of Node

communication using one of the three broadcast protocols, whereas the lower layer provides the actual transmission level facilities such as send request and acknowledgments.

The section will first describe DSL models of the three protocols used to ensure reliable distributed broadcasts. The protocols ABCAST, CBCAST and GBCAST are described in further detail in [Birman 87]. The model of a node using the structure presented in the above figure will be described with the lower level services tailored to interface with the ATM protocol presented in the previous section.

4.2.1 The ABCAST protocol

The ABCAST protocol is one of three broadcast process primitives (within a distributed environment) and ensures the order of a broadcast message received at multiple destinations from a source process is the same even though the order is not pre-determined. The DSL model describes node modules which represent distributed sites. A high level program (*high_level_services*) will generate processes that request ABCAST communication with other nodes through a *low_level_services* module. To support the ABCAST primitive, each node will have an *abcast_server*, a *priority_queue*, and a *delivery_queue*. The structural representation of this model is given in figure 4-9. The algorithm presented in [Birman 87] for ABCAST is presented below with the DSL representation.

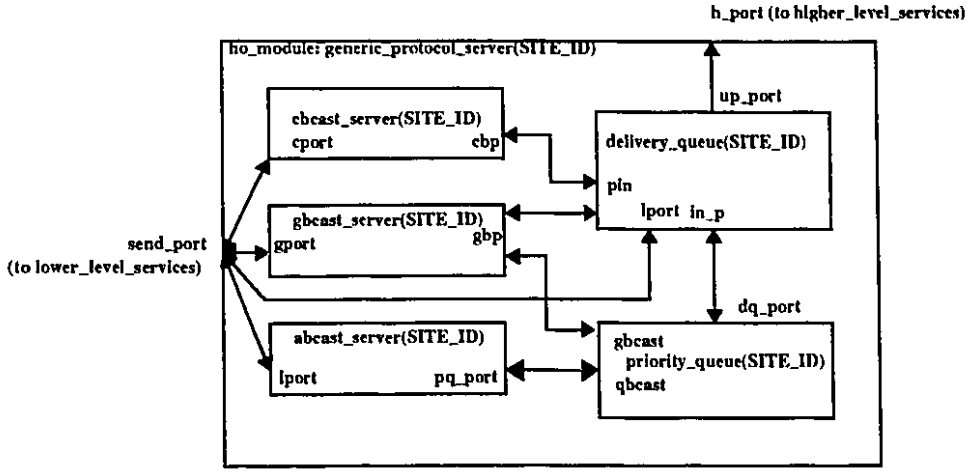


FIGURE 4-9. DSL Representation of generic_protocol_server

The algorithm is composed of four steps detailed below:

Step 1. The sender transmits a message (Msg) with a unique label to its destinations.

This is represented through a *form(abcast, PROC_ID, Dest, Msg)* message sent to the *abcast_server* (from the upper level protocols). *PROC_ID* is the source process identifier, *Dest* is a list of destinations and *Msg* is the message to be transmitted. This segment of DSL code is:

```
(form(abcast, PROC_ID, Dest, Msg):-
        check_res(label_count, COUNT),
        NCount is COUNT + 1,
        list_length(Dest, No),
        delay(X),
        set_res(label_count, NCount),
        create(NCount, (Dest, No, 0)),
        write(' Sending an abcast message '), write(Msg), nl,
        send(_, lprot(SITE_ID), msg_rqst(Dest, abcast(Dest, Msg, NCount, PROC_ID, SITE_ID))))
```

Two resources are utilized above: *label_count* and *NCount*. The first is used to generate unique labels for the messages. The latter is utilized to keep track of how many destinations within a process group have responded during step 2 of the protocol. The last line will send

the abcast message to a lower level protocol handler (connected through port *lprot(SITE_ID)*).

Step 2. Recipients add Msg to a priority queue associated with label, marking it as undeliverable. A priority is assigned to the message (NPri) larger than the largest in the queue, with a process identification (PROC_ID) added as a suffix. This suggested priority is then transmitted to the sender.

The DSL code given below depicts the behavior for this step. However an interaction with the *priority_queue* module (connected through the *pq_port(SITE_ID)*) is not shown here, but will be presented later in the section. The *add_new* message sent to the *priority_queue* module requests that the module add the new ABCAST message to the priority queue.

```
(abcast(Msg, Label, PROC_ID, Sender_ID):-
        send(_, pq_port(SITE_ID), add_new(abcast, Sender_ID, Msg, Label, PROC_ID)))
```

The priority_queue module will consequently send an updated_priority message (shown below) that provides the suggested priority (*NPri*). This value is sent back to the sender as shown below:

```
(updated_priority(abcast, SITE, Label, NPri):-
        send(_, lprot(SITE_ID), msg_rqst(SITE, abcast(SITE, sug_priority(NPri), Label))))
```

Step 3. The sender waits for all the suggested priorities from the destinations, computes the maximum of all the values and sends this value to all the destinations - thus guaranteeing correct order of delivery.

The DSL representation given below is used to compute the maximum priorities as well as a check if all responses have been collected from the destinations. The largest priority value is maintained within the resource *Label* (*NCount* in step 1) as well as a flag (*State*) indicating if all replies have been received.

```
(abcast(sug_priority(Value), Label):-
        check_res(Label, (Dest, State, Prior)),
        NState is State -1,
        set_res(Label, (Dest, NState, Prior)),
        Value> Prior,
        set_res(Label, (Dest, NState , Value )),
        send(check(Label))),
(abcast(sug_priority(Value), Label):-
        send(check(Label)),
        write('Priority received was not high enough'))

(check(Label):-
        check_res(Label, (Dest, State, Prior)),
        State>0,
        write('Have not received all replies yet'),nl),
(check(Label):-
        check_res(Label, (Dest, State, Prior)),
        delay(1),
        remove(Label, X),
        send(_, lport(SITE_ID), msg_rqst(DEST, abcast(set_priority(Prior), Label))))
```

Step 4. The destinations update the priority for Msg to the new value and mark the message as deliverable and re-sort the priority queues. The destinations then move messages in order of increasing priority from the priority queues to a delivery queue. This continues as long as the priority queue remains non-empty and there is a deliverable message at the top of the queue.

```
(abcast(set_priority(Value), Label, Proc_ID):-
        send(_, pq_port(ID), change_priority(abcast, Value, Label, Proc_ID)))
```

The above DSL code describes the final action that is required by the abcast_server. A message is sent to the *priority_queue* to change the priority of the message identified by *Label*. The *priority_queue* module requires some behavioral code to communicate with the *abcast_server* and the *delivery_queue*. Its behavior is given as follows:

The *add_new* behavior is used to add a new message to a priority queue

(*p_queue(PROC_ID)*). There are two cases defined below. The first case assumes that the priority queue for a process exists. Hence a new priority is assigned to the new message and another data structure (*waiting_message(PROC_ID)*) is created to keep a copy of the message and relevant information such as state (*deliverable* or *undeliverable*), message label, data and priority value.The second case takes into account the situation where a queue does not exist yet for a process (PROC_ID). Hence a queue is created to hold ordered priorities and all other operations are performed as in the first case.

```
(add_new(Protocol, Sender, Msg, Label, PROC_ID):-
        check_res(p_queue(PROC_ID), [Pri| Rest] ),
        NPri is Pri+1,
        create(waiting_message(PROC_ID), (NPri, Msg, Label, undeliverable)),
        check_res(p_queue(PROC_ID), QUEUE ),
        set_res(p_queue(PROC_ID), [Npri| QUEUE]),
        send(_, proc_port(SITE_ID), updated_priority(Protocol, Sender, Label, NPri))),
(add_new(Protocol, Sender, Msg, Label, PROC_ID):-
        NEW is SITE_ID/100,
        delay(1),
        create(p_queue(PROC_ID), [NEW]),
        create(waiting_message(PROC_ID), (NEW, Msg, Label, undeliverable)),
        send(_, proc_port(SITE_ID), updated_priority(Protocol, Sender, Label, New))),
```

The second behavior understanding required by the priority_queue is *change_priority*. Upon receipt of this message, the module updates the priority (*Value*) of the specified message (*Label*). An internal message *check_pq* is generated to clean up empty queues and test if the topmost element of the *p_queue* is deliverable. The latter is accomplished through another internal message (*delivery*) which removes deliverable messages from the top of the queues and sends them to the *delivery_queue* module.

```
(change_priority(abcast, Value, Label, PROC_ID):-
        check_res(waiting_message(PROC_ID), (Pri, M, Label, State)),
        check_res(p_queue(PROC_ID), QUEUE),
        delay(1),
        remove_element(QUEUE, Pri, NQ),
        append([Value], NQ, Result),
```

```
        iqsort( Result, Sorted),
        set_res(p_queue(PROC_ID), Sorted),
        set_res(waiting_message(PROC_ID), (Value, M, Label, deliverable))
        send(check_pq(abcast, PROC_ID, Label)),
(check_pq(abcast, ID, Label):-
        check_res(p_queue(ID), Queue),
        last_list(Queue, TOP),
        check_res(waiting_message(ID), (TOP, M, Label, STATUS)),
        send(delivery(Label, STATUS))),
(check_pq(abcast, ID, Label):-
        remove(p_queue(ID), []),
        write('Priority Queues for ID '), write(ID), write(' freed'),nl),
(delivery(Label, deliverable):-
        check_res(waiting_message(ID), (Val, M, Label, deliverable)),
        remove(waiting_message(ID), (Val, M, Label, deliverable)),
        check_res(p_queue(ID), Q),
        remove_element(Q, Val, NQ),
        set_res(p_queue(ID), NQ),
        send(_,dq_port(SITE_ID),delivery_of_new_message(ID, M, Label)),
        send(check_pq(abcast, ID, Label))),
(delivery(L, undeliverable):-
        write('Undeliverable message still waiting for processing.'),nl)]).
```

The DSL description for the delivery_queue simply records the sequence of messages delivered to a node. The queue functionality is implicit by the manner in which modules process incoming messages. The DSL representation is:

```
module(delivery_queue(SITE_ID),[
(delivery_of_new_message(ID, M, LABEL):-
        write('Message processed at node: '),
        write(SITE_ID),nl, write(' ID= '), write(ID),nl,
        write(' label= '), write(LABEL),nl)]).
```

This is a very simple view of the delivery queue, however the model will be refined later within the GBCAST protocol section to reflect the needs of all three protocols.

The behavior of the *abcast_server* module is described in terms of its equivalent predicate/

transition net given in figure 4-10. The protocol functionality is separated into two sets for clarity; the sender and receiver components. Each handles the respective functionality of the protocol. Input places (identified by the darkened circles) will allow their respective token types (such as *form_abcast(D,M,P)*). These tokens arrive from other sub-nets (representing the other modules). A controlling net (omitted here for sake of clarity) ensures that the specified incoming tokens are directed to the appropriate input places as well as ensuring safeness within each sub-net. Operations on transitions are indicated between a "<>" while predicates are labelled on arcs. By convention, labels beginning with a capitalized letter is an unassigned variable.

Even at this level of representation, some design partitioning is being assumed. For example, the protocol and priority queue management aspects are separated into different modules which implies that the eventual design would also favor such a distinction.

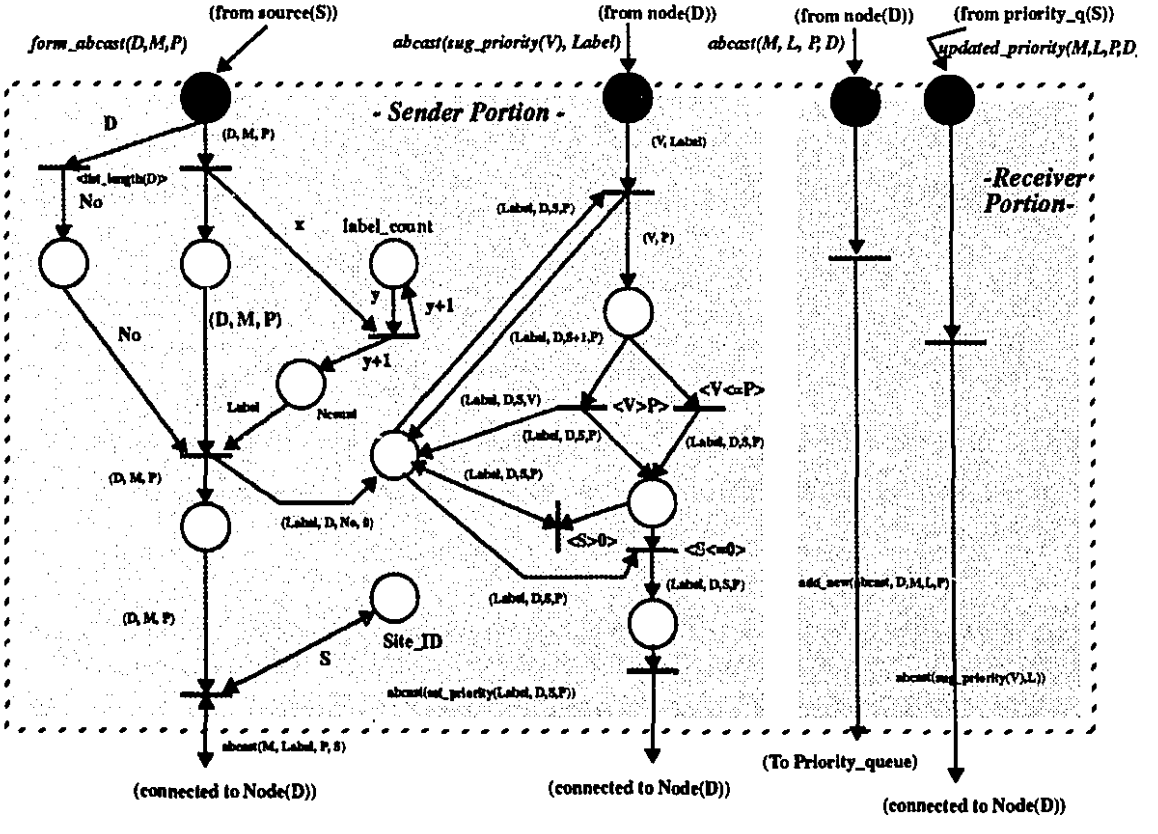


FIGURE 4-10. PrTN Representation of abcast_server Module

4.2.2 The CBCAST protocol

The CBCAST protocol is less restrictive than ABCAST, while preserving the causality of messages. It enforces a minimally synchronized delivery order. In this case, if a and b are two concurrent processes, CBCAST may deliver the processes in different orders whereas with ABCAST an order would be agreed upon by the destinations. The perceived advantage over ABCAST is the performance of the protocol. CBCAST does not delay transmission of messages and can therefore perform significantly faster. The algorithm for an implementation of the CBCAST protocol as presented by Birman is as follows:

Each process p has a message buffer BUFp which contains copies of messages sent to and from it. A message B is transmitted from BUFp at site s to BUFq at site t as follows:

1. When a CBCAST message B is initiated by p, an ID(B) is associated with the message and the list of destinations are added to REM_DESTS(B) to keep a record of destinations that are destined to receive the message.

2. A transfer packet <B1, B2, ...> is created which includes all messages B' in BUFp such that B' must precede B and REM_DESTS(B') is non-empty. The messages are maintained in an order of precedence to preserve causality (B1 must precede B2).

3. The transfer packet is sent from site s to t.

4. For each Bi, the destination q is deleted from REM_DESTS(Bi).

5. On the receiving site t, for each message in preceding order:
5.1 The ID(Bi) is checked. If it is associated with a message in BUFq then Bi is a duplicate and is discarded.

5.2 If q is a member of REM_DESTS(Bi), q is removed from REM_DEST(Bi), Bi is placed on the delivery queue for q and a copy of Bi is placed in BUFq.

5.3 If none of the above two cases apply, Bi is a message in transit to another process and is simply placed in BUFq.

The CBCAST algorithm is modelled within DSL using the following resources:

buff(PROC_ID) represents the BUFFp (where p=*PROC_ID*) in the algorithm above, and the variables within this resource are *(ID, M, REM_DESTS). ID* is a unique identification of the message *M* and *REM_DESTS* - as above, represents the (remaining) destinations that the message is to be sent to.

The DSL model follows the same steps outlined above. There are two possible messages that can be generated to the *cbcast_server* from a higher level protocol: 1) a message to transmit any outstanding messages in the message buffer *buff(PROC_ID)* and 2) a request for the transmission of a new CBCAST message. Since the former is a special case within the latter, the DSL code deals with the latter and is given below. A message from a higher level protocol is given as *form(cbcast, PROC_ID, DESTS, MESSAGE)* to create a new entry in the buffer for the process (PROC_ID). The former is represented by a *form_cbcast* message. This message looks at any entries in the buffer(s) where there are messages awaiting transmission, creates a transfer packet (PACKET) and send the result to a lower layer protocol for transmission. The DSL code to handle the CBCAST transmission is given as:

```
(form(cbcast, PROC_ID, DESTS, Message):-
        check_res(message_label(PROC_ID), Label),
        ID is Label+1,
        set_res(message_label(PROC_ID), ID),
        create(buff(PROC_ID), (ID, [DESTS], Message)),
        send(form_cbcast)),
(form_cbcast:-
        check_res(buff(PROC_ID), (ID, [DEST|REST], M)),
        set_res(buff(PROC_ID), (ID, [REST], M)),
```

```
        send(create_packet(PROC_ID, DEST, (ID, [REST], M) ))),

(create_packet(PROC_ID, DEST, PACKET):-
        check_res(buff[PROC_ID), (ID, REM, M)),
        not(member((ID,REM,M), PACKET))
        send(create_packet(PROC_ID, DEST, [PACKET | (ID,REM,M)]))),
(create_packet(PROC_ID, DEST, PACKET):-
        send(-, proto_port, msg_rqst(DEST, cbcast_packet(DEST, PACKET))),
```

The DSL model also requires knowledge about how an incoming CBCAST message is to be handled (step 5 of the algorithm for the protocol). A lower layer protocol is assumed to deliver a CBCAST message as *cbcast_message(PROC_ID, PACKET)*, where PACKET is an ordered precedence list of messages sent to this site and PROC_ID is the destined process identification. The DSL behavioral code along with the equivalent steps of Birman's algorithm is shown below. Step 5.0 is an additional case that is the stopping condition for the algorithm.

```
(cbcast_message(PROC_ID, [ ]):-                                          (step 5.0)
        write('CBCAST messages all processed at '),
        write(PROC_ID),nl),
(cbcast_message(PROC_ID, [(ID, REM, M) | Rest]):-                        (step 5.1)
        check_res(buff[PROC_ID), (ID, _,_) ),
        write('CBCAST message id '),write(ID),
        write(' already in buffer.'),nl,
        send(cbcast_message(PROC_ID, Rest))),
(cbast_message(PROC_ID, [(ID, REM, M) | Rest]):-                         (step 5.2)
        member(PROC_ID, REM),
        remove_element(REM, PROC_ID, NEW_REM),
        create(buff[PROC_ID), (ID, NEW_REM, M)),
        send(delivery_queue(PROC_ID),_,delivery_of_new_message(ID, M)),
        send(cbcast_message(PROC_ID, Rest))),
(cbcast_packet(PROC_ID, [(ID, REM, M) | Rest]):-                         (step 5.3)
        not(member(PROC_ID, REM)),
        create(buff[PROC_ID), (ID, REM, M)),
        send(cbcast_message(PROC_ID, Rest)))]).
```

4.2.3 The GBCAST protocol

The last of the three protocols to be described provides a mean of communication between changing process groups. The previous two protocols assume that all the destinations are known, however GBCAST permits communication to a group of processes whose membership may change during the course of time. To manage the group view, the protocol provides a totally ordered delivery scheme with respect to the other two protocols. The GBCAST protocol presented by Birman is outlined below:

1. The first steps of the protocol order GBCASTs with respect to ABCAST messages. A GBCAST message is composed of a destination group G and an action to be performed by the group members (action). A process p distributes the message to the member processes of group G.

The *form* message is interpreted by the gbcast_server as a request to send a GBCAST message (*ACTION*) to a *GROUP*. A local view of groups is maintained within a *site_group_table* (records which groups a node belongs to) and a *proc_group_table* (which tracks the processes membership in groups). A resource *label_count* is also used to generate increasing unique label values for messages. The originator of the GBCAST also tracks the replies from destination members using the resource *LABEL* (line 1.9).

```
1.1   (form(gbcast, PROC_ID, GROUP, ACTION):-
1.2           check_res(label_count, COUNT),
1.3           check_res(proc_group_table, (GROUP, PROCS)),
1.4           check_res(site_group_table, (GROUP, SITE)),
1.5           LABEL is COUNT + 1,
1.6           list_length(GROUP, No),
1.7           delay(DELAY1),
1.8           set_res(label_count, LABEL),
1.9           create(LABEL, (GROUP, No, 0)),
1.10          write(' Sending a gbcast message '), write(ACTION), nl,
1.11          send(_, gport, msg_rqst(SITE, gbcast(SITE_ID, ACTION, LABEL, GROUP)))),
```

2. Following the steps of the ABCAST protocol, the recipient q places the message (tagged as undeliverable) on all ABCAST priority queues. A priority value is assigned to all the messages based on the next highest value of any message on the ABCAST queues and the value is sent back to process p.

```
(gbcast(Sender, ACTION, LABEL, G):-
        check_res(proc_group_table, (G, PROC)),
        create(label_member, (LABEL, PROC, Sender)),
        send(gbcast_m(Sender, ACTION, LABEL, PROC))),

(gbcast_m(Sender, Msg, Label, []):-
        write('All gbcast priorities suggested...'),nl),
(gbcast_m(Sender, Msg, Label, [PROC_ID | REST]):-
        send(_,pq_port(SITE_ID),add_new(gbcast, Sender, Msg, Label, PROC_ID),
        send(gbcast_m(Sender, Msg, Label, REST))),

(updated_priority(gbcast, Sender, Label, NPri):-
        send(_, gport,msg_rqst(Sender, gbcast(sug_priority(NPri), Label))))
```

3. After obtaining all values from group members, p sends the maximum value of all the values to the group G. The group members accordingly assign the new value and re-sort the priority queues. However, the messages are not tagged as deliverable as in the ABCAST case before. When the GBCAST messages arrive to the head of their respective priority queues further delivery from the queues will be suspended. When the GBCAST message reaches the head of all the ABCAST priority queues, the next steps are undertaken to order the GBCASTs relative to CBCASTs.

```
(gbcast(sug_priority(Value), Label):-
        check_res(Label, (Dest, State, Prior)),
        NState is State -1,
        set_res(Label, (Dest, NState, Prior)),
        Value> Prior,
        set_res(Label, (Dest, NState , Value )),
        send(check(Label))),
```

```
(gbcast(sug_priority(Value), Label):-
        send(check(Label)),
        write('Priority that was received was not high enough'),nl),
(check(Label):-
        check_res(Label, (Dest, State, Prior)),
        State>0,
        write('Have not received all replies yet'),nl),
(check(Label):-
        check_res(Label, (Dest, State, Prior)),
        delay(1),
        remove(Label, X),
        check_res(site_group_table, (GROUP, SITE)),
        send(_, gport, msg_rqst(SITE, gbcast(set_priority(Prior), Label,GROUP)))),
```

The DSL description presented above is virtually identical to step 3 for ABCAST and is presented for sake of completeness. The difference is in the way in which the destinations are specified. In ABCAST the destinations were given directly as a part of the protocol. For GBCAST the site and process group tables are used to identify the destinations within a group relevant to a particular site.

Since GBCAST does not completely follow the ABCAST step 4, there are some additional differences to be considered. Unlike ABCAST, messages are not tagged deliverable at this stage. The protocol defined below will tag them as gbcast_wait, indicating that the message is to be delivered according to GBCAST. An additional resource (gb_ab_order) is maintained in the gbacst_server which keeps track of how many GBCAST members within the node still have not reached the top of their respective priority queues. The ABCAST step 4 represented in DSL as applied to GBCAST would then be:

```
(gbcast(set_priority(Value), Label, G):-
        check_res(proc_group_table, (G, PROCS)),
        create(gb_ab_order, (G, Label, PROCS)),
        send(gbcast(set_priority(Value), Label, G, PROCS)
(gbcast(set_priority(Value), Label, G, []):-
        check_res(proc_group_table, (G, PROCS)),
```

```
        send(_,pq_port(SITE_ID),check_pq(gbcast, PROCS, Label))),
(gbcast(set_priority(Value), Label, G, [ID | REST]):-
        send(_,pq_port(SITE_ID), change_priority(gbcast, Value, Label, ID)),
        send(gbcast(set_priority(Value), Label, G, REST))
```

The priority_queue will require some additions to add the GBCAST message. This is conveniently accomplished through the *change_priority* message. The first parameter of the message shown below identifies it as a GBCAST message, hence it is tagged as being in a *gbcast_wait* state.

```
(change_priority(gbcast, Value, Label, PROC_ID):-
        check_res(waiting_message(PROC_ID), (Prl, M, Label, State)),
        check_res(p_queue(PROC_ID), QUEUE),
        delay(1),
        remove_element(QUEUE, Prl, NQ),
        append([Value], NQ, Result),
        iqsort( Result, Sorted),
        set_res(p_queue(PROC_ID), Sorted),
        set_res(waiting_message(PROC_ID), (Value, M, Label, gbcast_wait))),
        send(check_pq(abcast, PROC_ID, Label))
```

In addition, the priority queue will inform the *gbcast_server* when a message with a state of *gbcast_wait* is detected at the head of a *priority_queue*. This is described through the delivery message below where a message *gbcast_wait* is sent to the *gbcast_server.*

```
(delivery(Label, gbcast_wait):-
        check_res(waiting_message(PROC_ID),(V,M,Label,gbcast_wait)),
        send(_, proc_port(SITE_ID), gbcast_wait(PROC_ID, Label))).
```

The *gbcast_server* module must process the *gbcast_wait(PROC_ID, Label)* message generated above. The module identifies removes the process identification (*PROC_ID*) from the appropriate *gb_ab_order* resource. When the third variable within the resource is an empty list, this implies that all members of the group are at the top of their respective

priority queues (these tests are performed within the additional *test_ab* behaviors given below). At this point a *set_wait_queue* message is sent to each process' delivery queue through the *instant_queue* behaviors. The additional code for the *gbcast_server* is given below.

```
gbcast_wait(PROC_ID, Label):-
        check_res(gb_ab_order, (G, Label, PROCS)),
        remove_element(PROCS, PROC_ID, NEW_PROCS),
        set_res(gb_ab_order, (G,Label, NEW_PROCS)),
        send(test_ab(G, Label, NEW_PROCS))

test_ab(G,Label,[]):-
        check_res(proc_group_table, (G, PROCS)),
        check_res(label_member, (Label, P, Sender)),
        send(instant_queue(Sender, PROCS, PROCS)),
test_ab(G,Label,[]):- print(' Not all members are at head of queues'),

instant_queue(S, G, []):-print('All wait queues generated'),
instant_queue(Sender, PROC, [MEMBER | REST]):-
        send(_,_,set_wait_queue(MEMBER)),
        send(_,_,flush_cbcast(Sender, PROC, MEMBER))
        send(instant_queue(REST))
```

4. A wait queue is used by the recipient processes to temporarily hold messages that would have been placed on the delivery queue by the CBCAST protocol. A list (IDLIST) is maintained which contains IDs of all CBCAST messages that were placed on a delivery queue of the recipient processes.

The requirements for this step of the algorithm dictate the need for a more sophisticated *delivery_queue* module. The module will contain resources that define the delivery queue (*d_queue*) and the wait queue function (*wait_queue*), as well as *idlist(ID)* to store the message labels within a *d_queue(ID)*. The DSL description below also identifies the behaviour of the *set_wait_queue* message sent from the gbcast_server. The

delivery_of_new_message will place the message onto a wait_queue if it exists or directly onto the *d_queue*. In the latter case a new message notification is sent to the upper layer protocol.

```
(set_wait_queue(ID):-
        create(wait_queue(ID), []) )

(delivery_of_new_message(ID, M, LABEL):-
        check_res(wait_queue(ID), REST),
        set_res(wait_queue(ID), [(M, LABEL)],
(delivery_of_new_message(ID, M, LABEL):-
        check_res(d_queue(ID), CONTENTS),
        set_res(d_queue(ID), [(M,LABEL) | CONTENTS]),
        check_res(idlist(ID), REST),
        set_res(idlist(ID), [LABEL | REST]),
        send(_,up_port(SITE_ID), new_msg)),
```

5. For a process q, if any message B in IDLIST is in BUFFq and B includes destinations that are a part of the group G, these messages are scheduled for transmission to the destinations. When all such messages are sent, q sends its IDLIST to the originating process p.

This step requires the interaction of the *gbcast_server*, *delivery_queue* and *cbcast_server* modules. First, the *gbcast_server* must generate a *flush_cbcast* message (from step 3) to the *delivery_queue*. This signals that GBCAST ordering with respect to ABCASTs are done and GBCAST with respect to the CBCAST ordering must now be performed. The *delivery_queue* module will use this message to transfer the *idlist* resource data to the cbcast_server with the *gb_request* message. The *cbcast_server* will then send any messages residing in its buffer belonging to a destination who is a member of a given group.

The DSL code below is a part of the *delivery_queue* behavioral description. As a response to the *gbcast_server's flush_cbcast* message, the module transmits a *gb_request* message

to the *cbcast_server* which is a request to process any messages that are to be transmitted to group members. A *cbcast_list* message is sent to the *gbcast_server* which is used within step 6 of the protocol.

```
flush_cbast(Sender, PROCS, PROC_ID):-
        check_res(idlist(PROC_ID), IDLIST),
        send(cbcast_server(SITE_ID),_, gb_request(PROCS, IDLIST)),
        send(_, lport, msg_rqst(Sender, cbcast_list(IDLIST, PROCS, PROC_ID)))
```

The code given below is located within the *cbcast_server* and is a description of the actions to be undertaken upon receiving the *gb_request* from the *delivery_queue*. Essentially each member of the REM destination list within the *buff* resource is compared with each member of the IDLIST within the *test_membership* behaviours and a *create_packet* message generated for CBCASTs to be sent to group members (this was described in the CBCAST protocol section).

```
gb_request(P, []):- print('All outstanding CBCASTs sent'),

gb_request(PROCS, [B| IDLIST] ):-
        check_res(buff(PROC_ID), (B, REM, M)),
        send(test_membership(REM, PROCS, B, PROC_ID),
        send(gb_request(PROCS, [IDLIST] ))
(test_membership([], G,ID,P):-
        print('Group members flushed from buffer')),
(test_membership([R|REST], GROUP,ID, PROC_ID):-
        member(R, GROUP),
        check_res(buff(PROC_ID), (ID, LIST, M)),
        remove_element(R, LIST, NEWLIST),
        set_res(buff(PROC_ID), (ID, NEWLIST, M)),
        send(create_packet(PROC_ID, DEST, (ID, NEWLIST, M)))
        send(test_membership(REST, GROUP, ID, PROC_ID))),
(test_membership([R| REST], GROUP,ID, PROC_ID):-
        send(test_membership(REST, GROUP, ID, PROC_ID)))
```

6. After receiving all the IDLISTs, p merges the lists into a before list which it sends to G.

The *cbcast_list* message that was sent from all the participating group members is processed within the GBCAST originator's *gbcast_server* module. A resource *before_list(Group)* is maintained for the group which keeps the merged version of the IDLIST as well as a list of members that have sent their lists. Upon receipt of an IDLIST from a group member, the member is removed from the *before_list(Group)*. A *test_before* behaviour is used to determine if all the members have sent their lists (this would be an empty list for the second variable of the before list). If so, the before list is sent to members as the message *before_list(IDlist, Group)*.

```
(cbcast_list(IDLIST, Group, PROC_ID):-
        check_res(before_list(Group), (LIST, PROCS)),
        remove_element(PROCS, PROC_ID, NEW_PROC),
        set_res(before_list(Group), ([IDLIST|LIST], NEW_PROC)),
        send(test_before(Group, NEW_PROC))),
(test_before(GROUP, []):-
        check_res(proc_group_table, (GROUP, PROCS)),
        check_res(site_group_table, (GROUP, SITE)),
        check_res(before_list(GROUP), (IDlist, _)),
        send(_, _, msg_rqst(SITE, before_list(IDlist, GROUP))))
(test_before(G,P):-print('Have not received all idlists yet'))
```

7. Any messages in the wait queue that are also in the before list are transferred to the delivery queue, maintaining their order. The GBCAST message is also placed on the delivery queue.

The gbcast_server requires an additional description of the before_list message. Receipt of this by member modules initiates a transfer of the new (before) List to the respective delivery_queue modules (sent as new_idlist).

```
(before_list(List, GROUP):-
        check_res(proc_group_table, (GROUP, PROCS)),
```

```
        send(send_list(List, PROCS))),
(send_list(List, []):-print(' Updated IDLIST sent')),
(send_list(List, [M | Rest]):-
        send(delivery_queue(M),_, new_idlist(List)),
        send(send_list(List,Rest))),
```

The *delivery_queue* modules perform the necessary comparisons below to move entries from the *wait_queue* to the *d_queue*. Finally a *get_gbcast* message is sent to the respective *priority_queue* modules to obtain a copy of the GBCAST message.

```
(new_idlist(List):-
        check_res(wait_queue(PROC_ID), WAIT),
        check_res(d_queue(PROC_ID), DELIVERY),
        send(compare_queues(WAIT, List,[],WAIT))),
(compare_queues([], BeforeList, TEMP, WAIT):-
        set_res(wait_queue(PROC_ID), WAIT),
        check_res(d_queue(PROC_ID), DELIVERY),
        set_res(d_queue(PROC_ID), [TEMP | DELIVERY])),
        send(priority_queue(PROC_ID),_,get_gbcast)),
(compare_queues([Top | Rest], BeforeList, N, NewWait)):-
        member(Top, BeforeList),
        append(Top, N, TEMP),
        remove_element(Top, NewWait, WAIT),
        send(compare_queues(Rest, BeforeList, TEMP, WAIT))),
```

The *priority_queues* react to the *get_gbcast* message by removing the GBCAST message from the *waiting_message* resource and transferring it to the *delivery_queue*.

```
get_gbcast:-
        check_res(waiting_message(PROC_ID), (TOP, M, Label, gbcast_wait)),
          remove(waiting_message(PROC_ID), (Val, M, Label, gbcast_wait)),
          send(_,dq_port(PROC_ID),add_gbcast_message(M, Label)),
```

8. The contents of the wait queue are appended to the delivery queue and the wait queue is deleted.

The *delivery_queue* modules perform clean up operations after receiving the GBCAST message from the *priority_queue* modules. The message (*MESSAGE*) is received through the *add_gbcast_message* sent within the previous step. The message is added to the delivery queue resource *d_queue* as well as remaining contents of the *wait_queue*, after which the *wait_queue* is removed. A *clean_gbcast* message is sent to the *priority_queue* modules to remove the GBCAST message from the head of the *p_queues*.

```
add_gbcast_message(MESSAGE):-
        check_res(d_queue(PROC_ID), DELIVERY),
        set_res(d_queue(PROC_ID), [MESSAGE | DELIVERY]),
        check_res(wait_queue(PROC_ID), WAIT)),
        check_res(d_queue(PROC_ID), DELIVERY2),
        set_res(d_queue(PROC_ID), [WAIT | DELIVERY2]),
        remove(wait_queue(PROC_ID))
        send(priority_queue(PROC_ID),_,clean_gbcast)
```

9. The GBCAST messages are removed from the heads of the ABCAST queues.

This completes the final step of the protocol. The code shown below is within the *priority_queue* module to clean up the GBCAST message from the *p_queue*. A *check_pq* message is sent as outlined within the ABCAST protocol to commence delivery of any eligible ABCAST messages.

```
clean_gbcast:-
        check_res(p_queue(PROC_ID), [TOP | REST]),
        set_res(p_queue(PROC_ID), REST),
        check_res(p_queue(PROC_ID), [Pri | R]),
        check_res(waiting_message(PROC_ID), (Pri, M, Label, State)),
        send(check_pq(abcast, PROC_ID, Label)))
```

4.2.4 Node model

A node is defined to be composed of a *high_level_service* (describing the higher level services), a *generic_protocol_server* (describing the various supported broadcast protocols) and a *low_level_service* (describing the low level support for the node) module. The *generic_protocol_server* module details are presented in the previous section. This section will present the details for the high and low level services support for the modelled node. The DSL structure is shown in figure 4-11.

The *high_level_service* module is intended to generate the communication requests (typically from an application running on the node). In this example, a set of process modules residing within this higher order module are defined to generate various broadcast requests. Each process module possesses a unique *PROC_ID* and a *requests* resource. The latter stores communication requests to be initiated from the process. The resource contains three fields. The first identifies the protocol requested, the second a message to be executed by a destination process, and the third indicates the PROC_ID of the destination process. Each process follows a simple behaviour of cycling through each of its entries in the requests resource and sending a form message down to the *generic_protocol_server.* This behaviour is shown below.

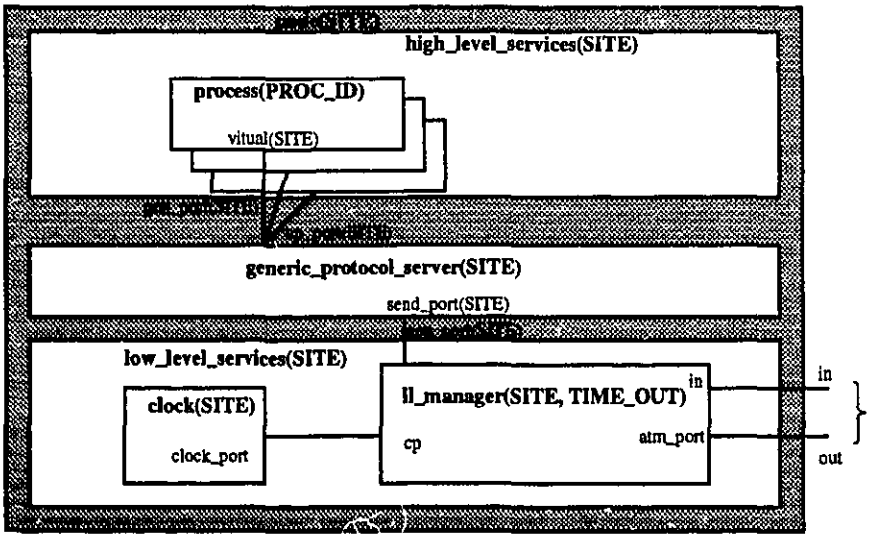


FIGURE 4-11. DSL Model of Broadcast Nodes

```
module(process(PROC_ID), [

start_process:-
        check_res(requests, (PRO, MSG, DEST)),
        set_res(requests, (PRO, MSG, DEST)),
        delay(PROCESS_DELAY),
        send(_, virtual(SITE), form(PRO, PROC_ID, DEST, MSG))
        send(start_process)]).
```

This description for a process will loop endlessly to provide a sequence of requests. As shown in the broadcast protocol models, the form message will initiate a given broadcast protocol which will eventually result in a *msg_rqst(DEST_SITE,MESSAGE)* message being sent from the *generic_protocol_server* to the *lower_level_services* module.

The msg_rqst message is a low level message to request a transmission of *MESSAGE* to a node address *DEST_SITE*. The *low_level_services* higher order module is composed of a *ll_manager(SITE,TIME_OUT)* and *clock(SITE)* module. The *ll_manager* provides the behaviour for actual communication and transmission to an ATM switch whereas the clock (as in the previous examples) provides the timing requirements for the communication. The *TIME_OUT* parameter identifies a time-out value for the low level communication protocol. The protocol modelled in this example is a simple send (*msg*) and acknowledge (*ack*) protocol. If an acknowledgment is not received within *TIME_OUT* number of ATM cells, then an error message is recorded.

To provide low level services, the *ll_manager* maintains a translation_table resource identifying the ATM (VPI,VCI) addresses for the destination. A message identification is also recorded for each message (from a *message_id* resource) to keep track of outstanding acknowledgments. A *time_out_buffer* for each message is used to record the waiting time for time-out purposes.Finally, a resource (*msg_store*) maintains a copy of messages waiting to be processed. The code for handling *msg_rqst* is given below.

```
msg_rqst([], _).
msg_rqst([DEST_SITE | REST], MESSAGE):-
        check_res(translation_table(DEST_SITE), (VPI,VCI)),
        check_res(message_id, VALUE),
        COUNT is VALUE+1,
        set_res(message_id, COUNT),
        create(msg_store(COUNT), (VPI, VCI, DEST_SITE, MESSAGE)),
        send(msg_rqst(REST, MESSAGE)
```

As defined in the ATM model, a *new_cell* message is sent to and from the switch. Hence the *ll_manager* must process such a message. This is defined by:

```
new_cell(VPI, VCI, MESSAGE):-
        send(MESSAGE).
```

The convention in this example is that *MESSAGE* can either be in the form of new message sent from another node - described as *hmsg(Sender, Destination, Message_ID, Message)* - or an acknowledgment to a past message (*ack(Sender, Destination, Message_ID)*). In the case of a *hmsg message*, an acknowledgment is sent back to the sender and the contents are forwarded to the *generic_protocol_server*. In the case of an ack message, the respective message is removed from the *time_out_buffer* - indicating a successful message transfer. The DSL descriptions are given below:

```
hmsg(Sender, _, MSGID, MESSAGE):-
        check_res(translation_table(Sender), (VPI,VCI)),
        send(_, atm_port(SITE), new_cell(VPI,VCI, ack(SITE, Sender, MSGID))),
        send(_, gen_port, MESSAGE)

ack(SITE, Sender, MSGID):-
        remove(time_out_buffer(MSGID), (TIME, DATA)),
        print('Message successfully received at destination').
```

There is also a *clock* message sent from the *clock* module to initiate the transmission of a cell. If a message is awaiting transmission it is sent as a *hmsg* type message to the

destination, otherwise a *null* ATM cell is sent to the switch. The *clock* message also is used to update any time-out values for waiting responses (this is performed within the *update_times* behaviour shown below). The description of the *clock* module is the same as the one given for the ATM switch design and will not be repeated here.

```
clock:-
        check_res(msg_store(MSGID), (VPI, VCI, DEST_SITE, MESSAGE)),
        remove(msg_store(MSGID), DATA),
        create(time_out_buffer(MSGID), (TIME_OUT, DATA)),
        send(_, atm_port(SITE), new_cell(VPI,VCI, hmsg(SITE, DEST_SITE, COUNT, MESSAGE)),
        send(update_times)
clock:-
        send(_, atm_port(SITE), new_cell(null,null,null),
        send(update_times)

update_times:-
        check_res(time_out_buffer(MSGID), (TIME, DATA)),
        NEW is TIME-1,
        set_res(time_out_buffer(MSGID), (NEW, DATA)),
        fail.
update_times:- .
        check_res(time_out_buffer(MSGID), (TIME, DATA)),
        TIME<0,
        print('Message '),print(MSGID),print(' timed out'),
        fail.
update_times:- print('Timeouts updated').
```

4.3 ATM - Broadcast system model

The DSL models presented in the ATM and this section provide the basic building blocks for defining an ATM based broadband network running the defined broadcast protocols. This section will demonstrate a design validation and simulation exercise that is typical for many protocol designs. A network model is shown in figure 4.12 depicting a configuration

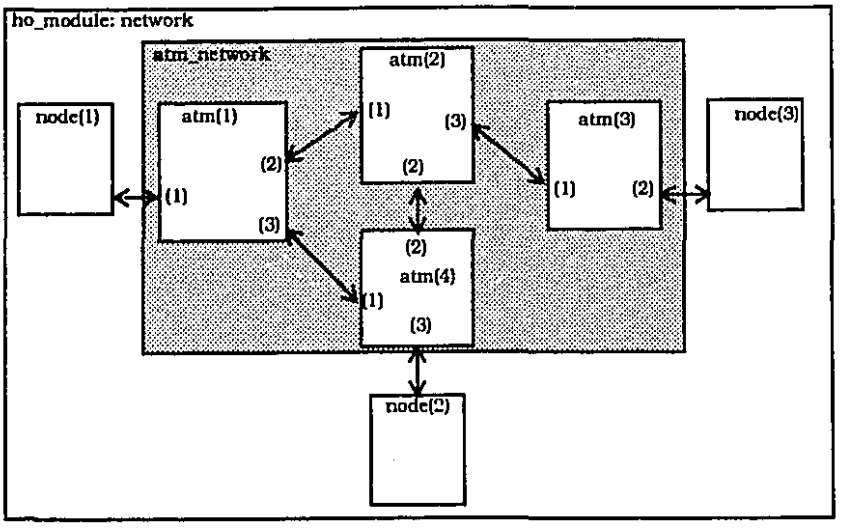


FIGURE 4-12. ATM-Broadcast Network Model

based upon the network introduced in the ATM section, and the nodes defined in this section. DSL model details for this level of the example will not be dwelt upon. The network is assumed to provide a bidirectional virtual path connection amongst the three nodes.

Each node is assumed to contain ten processes running within its high_level_services module. The processes are identified (PROC_ID) by a combination of the node number (SITE) and a unique integer n where $1 \leq n \leq 10$. The resulting PROC_ID is in the form: SITE.n. Five process groups are also defined within the model (G1...G5) and each process has 20 predefined broadcast messages consisting of a mix of the three broadcast protocols. Each process randomly chooses one of these messages, at random intervals, to broadcast to the various destinations. A constraint is also defined for the queue resource in the generic_buffer modules within each ATM switch. The constraint defines a maximum output queue size to detect overflowing message buffers (which is one of the major design issues in ATM switches).

When loaded within DASE, a hierarchy tree is generated by the system as shown in figure 4-13. The protocol modules are represented at the lowest level of the module hierarchy

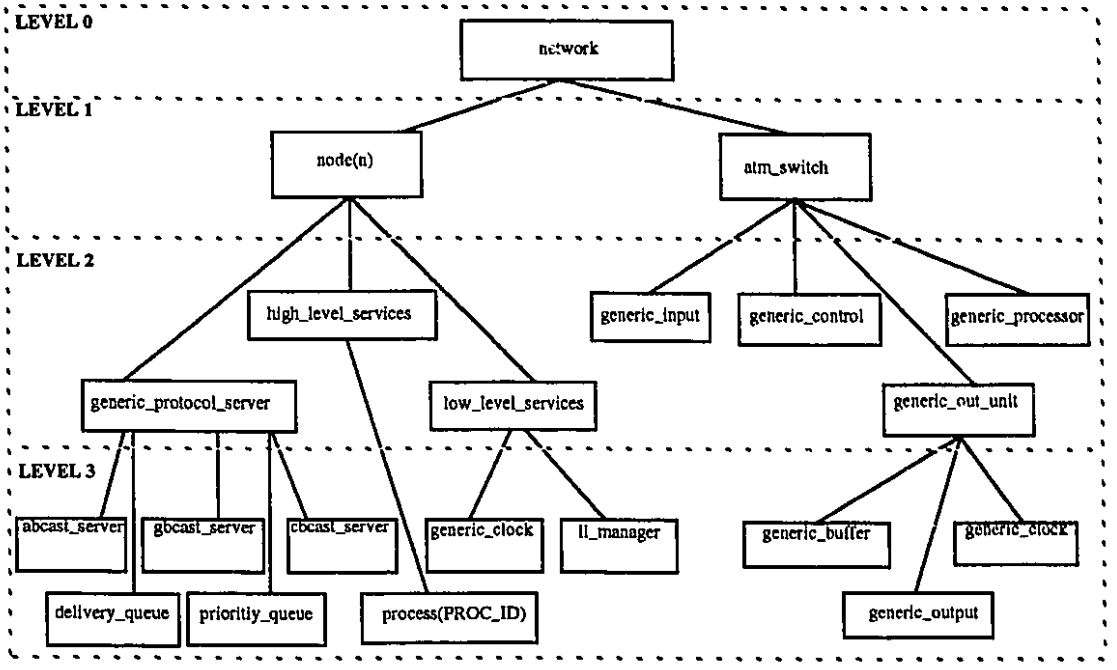


FIGURE 4-13. Hierarchy of Modules for ATM Broadcast

(level 3). Hence it may be desirable to implement these modules within software. All nodes are initiated with messages which randomly generate broadcasts, hence each simulation run can provide different outputs. A typical output is shown in figure 4-14. The effects of different observation levels can be seen. There are concurrent broadcast messages generated by different processes, though the output shows only those from process(1.1) residing on site 1. The ABCAST message is sent to processes at site2 and 3.

The first segment of code is set at an observation level of 1 which restricts the view to the higher level protocols being modelled. The sample code shows an ABCAST message request from the higher level protocol (l1), the receipt of the message at one of the destinations (l2), update of the priority queue (l3) and a suggestion of a priority value back to the destination (l4). This view avoids most of the low level details of the ATM protocol. However it provides a reasonable view of the message generation and delivery for the different broadcast protocols.

The second segment of the code shows the same simulation run with a different observation level. In this scenario, the switching function is of interest, hence higher level

protocol messages are not observed. The code segment shows the receipt of a cell from site 1 to a switch's input port (l5), translation of the cell headers (l6) and routing to the destination port on the switch (l7).

```
>set_level(1).

observation level set to 1

>run.

DSL SIMULATOR: Scheduling message from hls(1,200) via gps(1) to abcast(1)
Source Port: virtual(1,200) Message is: form(abcast,200,[3,2],abm3)
-==----------------------------------------------

<0.0> DSL SIMULATOR: new entry added to message queue of abcast(1)
<0.0> DSL PROCESSOR: Message start_process completed at module hls(1,200)
<0.0> DSL SIMULATOR: Module abcast(1) busy (delayed) until: 1.0
<1.0> DSL SIMULATOR: abcast(1) resource updated: label_count to value: 2
 Sending an abcast message abm3

-==----------------------------------------------
DSL SIMULATOR: Scheduling message from abcast(1) via gps(1) to lls(1,20)
Source Port: lport(1) Message is: msg_rqst([3,2],abcast(abm3,2,200,1))
-==----------------------------------------------

DSL SIMULATOR: Scheduling message from lls(2,20) via gps(2) to abcast(2)
Source Port: gen_port Message is: abcast(abm3,2,200,1)
-==----------------------------------------------

<10.0> DSL SIMULATOR: new entry added to message queue of abcast(2)
<10.0> DSL PROCESSOR: Message hmsg(1,2,2,abcast(abm3,2,200,1)) completed at module lls(2,20)

-==----------------------------------------------
DSL SIMULATOR: Scheduling message from abcast(2) via abcast(2) to priority_q(2)
Source Port: pq_port(2) Message is: add_new(abcast,1,abm3,2,200)
-==----------------------------------------------

<10.0> DSL SIMULATOR: new entry added to message queue of priority_q(2)
<10.0> DSL PROCESSOR: Message abcast(abm3,2,200,1) completed at module abcast(2)
<10.0> DSL SIMULATOR: Module priority_q(2) busy (delayed) until: 11.0
<10.0> DSL SIMULATOR: input(2,4) resource updated: cell_buffer to value: 1 , 2 , ack(2,1,2) , unprocessed

-==----------------------------------------------
DSL SIMULATOR: Scheduling message from input(2,4) via input(2,4) to control(4)
Source Port: control_port(4,2) Message is: new_cell(2)
-==----------------------------------------------

<10.0> DSL SIMULATOR: new entry added to message queue of control(4)
<10.0> DSL PROCESSOR: Message new_cell(1,2,ack(2,1,2)) completed at module input(2,4)

DSL SIMULATOR: Scheduling message from lls(3,20) via gps(3) to abcast(3)
Source Port: gen_port Message is: abcast(abm3,2,200,1)
-==----------------------------------------------

<10.0> DSL SIMULATOR: new entry added to message queue of abcast(3)
<10.0> DSL PROCESSOR: Message hmsg(1,3,1,abcast(abm3,2,200,1)) completed at module lls(3,20)

-==----------------------------------------------
DSL SIMULATOR: Scheduling message from abcast(3) via abcast(3) to priority_q(3)
Source Port: pq_port(3) Message is: add_new(abcast,1,abm3,2,200)
-==----------------------------------------------

DSL SIMULATOR: Scheduling message from priority_q(3) via priority_q(3) to abcast(3)
Source Port: abcast Message is: updated_priority(abcast,1,2,_1344)
-==----------------------------------------------

<11.0> DSL SIMULATOR: new entry added to message queue of abcast(3)
<11.0> DSL PROCESSOR: Message add_new(abcast,1,abm3,2,200) completed at module priority_q(3)


-==----------------------------------------------
DSL SIMULATOR: Scheduling message from priority_q(2) via priority_q(2) to abcast(2)
Source Port: abcast Message is: updated_priority(abcast,1,2,_7436)
-==----------------------------------------------
```

Figure 4-14 cont.

```
<11.0> DSL SIMULATOR: new entry added to message queue of abcast(2)
<11.0> DSL PROCESSOR: Message add_new(abcast,1,abm3,2,200) completed at module priority_q(2)
-==----------------------------------------
broadcast send completed for message msg_rqst(1,abcast(1,sug_priority(_11416),2))
<11.0> DSL PROCESSOR: Message updated_priority(abcast,1,2,_11416) completed at module abcast(2)

yes

>set_level(4).

observation level set to 4

>run.

DSL SIMULATOR: Scheduling message from lls(1,20) via site(1) to input(1,1)
Source Port: atm_port(1) Message is: new_cell(2,1,hmsg(1,2,2,abcast(abm3,2,200,1)))
-==----------------------------------------

<2.0> DSL SIMULATOR: new entry added to message queue of input(1,1)

----------------------------------------
DSL SIMULATOR: Scheduling INTERNAL message from lls(1,20)
Message is: update_times
----------------------------------------

<2.0> DSL PROCESSOR: Message clock completed at module lls(1,20)
<2.0> DSL SIMULATOR: input(1,1) resource updated: cell_buffer to value: 2 , 1 , hmsg(1,2,2,abcast(abm3,2,200,1)) , unprocessed

-==----------------------------------------
_24796 control_port(1,1) new_cell(1)
 Source Destin, and Sender input(1,1)control(1)input(1,1)
Module is control(1) message is new_cell(1)
Checking method in control(1)

Checking method in generic_control(1)
DSL SIMULATOR: Scheduling message from input(1,1) via input(1,1) to control(1)
Source Port: control_port(1,1) Message is: new_cell(1)
-==----------------------------------------

<2.0> DSL SIMULATOR: new entry added to message queue of control(1)
<2.0> DSL PROCESSOR: Message new_cell(2,1,hmsg(1,2,2,abcast(abm3,2,200,1))) completed at module input(1,1)

-==----------------------------------------
DSL SIMULATOR: Scheduling message from control(1) via control(1) to input(1,1)
Source Port: input_port(1,1) Message is: read_rqst
-==----------------------------------------

<2.0> DSL SIMULATOR: new entry added to message queue of input(1,1)
<2.0> DSL PROCESSOR: Message new_cell(1) completed at module control(1)
<2.0> DSL SIMULATOR: input(1,1) resource updated: cell_buffer to value: 2 , 1 , hmsg(1,2,2,abcast(abm3,2,200,1)) , processed

-==----------------------------------------
DSL SIMULATOR: Scheduling message from input(1,1) via input(1,1) to control(1)
Source Port: control_port(1,1) Message is: cell_info(2,1,hmsg(1,2,2,abcast(abm3,2,200,1)))
-==----------------------------------------

<2.0> DSL SIMULATOR: new entry added to message queue of control(1)
<2.0> DSL PROCESSOR: Message read_rqst completed at module input(1,1)
control unit: transferring a cell to output
-==----------------------------------------
DSL SIMULATOR: Scheduling message from control(1) via control(1) to processor(1)
Source Port: proc_port(1) Message is: new_message(2,1,hmsg(1,2,2,abcast(abm3,2,200,1)))
-==----------------------------------------

<2.0> DSL SIMULATOR: new entry added to message queue of processor(1)
<2.0> DSL PROCESSOR: Message cell_info(2,1,hmsg(1,2,2,abcast(abm3,2,200,1))) completed at module control(1)
ATM processor: routing a cell

-==----------------------------------------
DSL SIMULATOR: Scheduling message from processor(1) via processor(1) to buffer(1,3)
Source Port: out_port(3) Message is: atm_cell(2,3,hmsg(1,2,2,abcast(abm3,2,200,1)))
-==----------------------------------------


<2.0> DSL SIMULATOR: new entry added to message queue of buffer(1,3)
<2.0> DSL PROCESSOR: Message new_message(2,1,hmsg(1,2,2,abcast(abm3,2,200,1))) completed at module processor(1)
<2.0> DSL SIMULATOR: buffer(1,3) resource updated: queue to value: [(2 , 3 , hmsg(1,2,2,abcast(abm3,2,200,1))),(null , null , null)]
<2.0> DSL PROCESSOR: Message atm_cell(2,3,hmsg(1,2,2,abcast(abm3,2,200,1))) completed at module buffer(1,3)
```

Figure 4-14 cont.

```
-=----------------------------------------
DSL SIMULATOR: Scheduling message from buffer(1,3) via buffer(1,3) to output(1,3)
Source Port: output_port(1,3) Message is: next_info((2 , 3 , hmsg(1,2,2,abcast(abm3,2,200,1))))
-=----------------------------------------
-=----------------------------------------
DSL SIMULATOR: Scheduling message from output(1,3) via switch(3) to input(1,3)
Source Port: out_port(1,3) Message is: new_cell(2,3,hmsg(1,2,2,abcast(abm3,2,200,1)))
-=----------------------------------------
```

FIGURE 4-14. Sample Simulation Output

As the examples try to illustrate, different observation levels during simulation can help the user focus on the desired level of detail. This in turn, permits the user to zoom at different segments of the model during the course of the simulation, permitting observation of pertinent information. The real impact of these features are difficult to depict in words - the trial and use of the tool is the best way of observing the benefits.

Chapter 5 - Conclusions

This dissertation presents a rapid prototyping system for architectural level design of telecommunication systems. The basic notion introduced is the need for a small intermediate design language to refine conceptually abstract notions to more detailed executable designs. The Design Specification language has been introduced as a potential means for bridging the gap between the high level specification languages and behavioral VHDL for hardware description. This chapter summarizes the major findings and potential avenues for future research.

A major requirement to achieve an adequate level of design support is that the environment must be able to provide modular and re-usable library components. This is stressed within DSL with the use of generic modules and flexible interconnection schemes. The use of Prolog as the underlying implementation language was found to significantly ease the development of the DSL language. The implementation language did not significantly impact the simulator performance, as evident from some experimental results provided in appendix C.

There is a significant amount of design specification languages used by the community at the front end of the design process. To popularize the use of DASE within design teams, translators will be required from the specification language in use and DSL. For example, a DSL to SDL translator could be very favorable to a large amount of designers in the telecommunication protocol area - permitting SDL models to be refined through DSL and

synthesized to VHDL.

The dissertation has also provided a significant amount of language detail in terms of design examples and case studies. This is important to demonstrate the language elements and also the applicability of DASE within a reasonably realistic design project. Model development within DSL in the creation of the case studies was found to be quite intuitive. The relaxed use of port specifications greatly eased the construction of module behavior and generate generic components. A large portion of code introduced within a digital telephone switch was shown applicable to a broadband based ATM switch - demonstrating model re-use. The viewing capabilities of the simulator have also been observed to be very helpful in moving quickly through large design details.

Work on DASE is mainly focussed on hardware design, however the primary notions have been introduced for software modelling as well. Design exploration at the architectural level is possible in both domains and has been demonstrated to some extent within the broadcast protocol case study (chapter 4). In the example, protocol models can be implemented in hardware or software. Further work can involve in defining interfaces permitting the synthesis of software directed DSL modules to popular software design methodologies and tools.

As with any software engineering environment, DASE can be further enhanced to expedite the design modelling task.For example, future work remains in developing a more sophisticated and user friendly interface to communicate with the DSL processor. Research in the area of model verification is also required. A relationship between modules and predicate-transition nets was introduced, however other formal paradigms may also provide interesting analysis.

The environment described in this dissertation has been developed to provide the necessary hooks for design tools within different aspects of the design cycle. For example, synthesis directed predicates such as *module_type*, *output_list* and *mode* exist to facilitate translation to executable lower level representations. The *constraint* predicate can be utilized in

numerous ways within the design process. It can be used to define performance bounds during design exploration or it can be used to communicate manufacturing specific information back into the model.

It should also be restated that DASE is part of a larger concept - where a framework consisting of lower level design tools such as optimizers, structural synthesizers and simulators are considered as necessary ingredients in providing a complete solution for the designer. It is with this knowledge that many of the supporting predicates for DSL have been devised. With the development of the lower design tools, hopefully a seamless environment can be obtained permitting users to evolve abstract requirements into executable designs. The ultimate judge of the success of DASE will be its acceptance within different telecommunication system design groups.

References

[ATM 93] ATM Forum, User-Network Interface Specification (Version 3.0), Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[Bae 91] Jaime Jungok Bae and T. Suda, "Survey of Traffic Control Schemes and Protocols in ATM Networks", Proceedings of the IEEE, Volume 79, Number 2, Februaury 1991, Pages 170-189.

[Barbacci 81] M.R. Barbacci, "Instruction Set Processor Specifications (ISPS): The notation and applications, "IEEE Transactions on Computers, vol. C-30, no.1, pp.24-40, Jan. 1981.

[Bell 71] C.G. Bell and A.Newell, Computer Structures:Readings and examples. New York, NY, McGraw Hill, 1971.

[Bergamaschi 93] Reinaldo A. Bergamaschi, "High-Level Synthesis in a Production Environment: Methodology and Algorithms", Fundamentals and Standards in Hardware Description Languages - NATO ASI series, Kluwer Academic Publishers, 1993, Netherlands, Pages 195-230.

[Birman 87] Kenneth P. Birman and T.A. Joseph, "Reliable Communication in the Presence of Failures", ACM Transactions on Computer Systems, Vol. 5, No. 1, February 1987, Pages 47-76.

[Birman 93] Kenneth P. Birman, "The Process Group Approach to Reliable Distributed Computing", Communications of the ACM, December 1993, Vol. 36, Number 12, Pages 36-53.

[Bochman 90] Gregor V. Bochmann, "Specification of a Simplified Transport Protocol Using Different Formal Description Techniques", Computer Networks and ISDN Systems 18, Elsevier Science Pub., 1990, pp. 335-377.

[Boehm 81] Barry W. Boehm, Software Engineering Economics, Prentice Hall, N.Y., New York, 1981.

[Booch 91] Grady Booch, Object Oriented Design with Applications, Benjamin/Cummings Pub. Co., Redwood City, California, 1991.

[Borrione 93] Dominique Borrione, "CASCADE", Fundamentals and Standards in Hardware Description Languages - NATO ASI series, Kluwer Academic Publishers, 1993, Netherlands, Pages 411-430.

[CACI 87] CACI inc., SIMSCRIPT II.5 Programming Language, CACI inc., Los Angeles, CA, 1987.

[CCITT 88] CCITT Recommendation Z.100: Functional Specification and Description Language SDL, AP IX-35, Geneva, 1988.

[Cox 91] Brad J. Cox and A.J. Novobilski, Object Oriented programming: an evolutionary approach, Addison Wesley, Reading, Mass., 1991.

[Gajski 92] Daniel Gajski et al., High-Level Synthesis - Introduction to Chip and System Design, Kluwer Academic Publishers, Norwell Massachusetts, 1992.

[Ganesh 89] Ganesh C. Gopalakrishnan, "Formalization of an Operational Approach to Hardware Specification and Validation", Proceedings of the 3rd Banff Higher Order Wokshop, Banff, Alberta, September 24-27, 1989, pp. 1-29.

[Genrich 81] H.J. Genrich and K. Lautenbach, "System Modelling with High-Level Petri-Nets", Theoretical Computer Science 13, North-Holland Pub. Co., 1981, Pages 109-136.

[Gronberg 93] P. Gronberg et al, PROD - A Pr/T-Net Reachability Analysis Tool, Helsinki University of Technology, Digital Systems Laboratory, Series B: Technical reports, No. 11, June 1993.

[Gupta 92] Aarti Gupta, "Formal hardware verification methods: A survey", International Jouranl on Formal Methods in System Design, Kluwer Academic Pub., Vol. 1, Numbers 2/3, October 1992, pp. 151-238.

[Harel 87] David Harel, "Statecharts: A visual formalism for complex systems", Science of Computer Programming, 8, Elsevier Science Pub., 1987, Pages 231-274.

[Hayes 88] Robert H. Hayes, et al, Dynamic Manufacturing - Creating the learning organization, Free Press, NY, NY, 1988.

[Hoare 85] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[ISO 88] International Organization for Standardization. Information Processing Ssytems. Open Systems Interconnection. LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (ISO International Standard 8807), 1988.

[Jacobson 92] Ivar Jacobson et al, Object Oriented Software Engineering: a use case driven approach, Addison Wesley Pub., Reading, Mass., 1992.

[Jefferson 83] D.R. Jefferson, "Virtual Time", Proceedings of the 1983 International Conference on Parallel Processing, August 1983, Pages 384-394.

[Jensen 81] Kurt Jensen, "Coloured Petri-Nets and the Invariant Method", Theoretical Computer Science 14, pp 317-336. (1981).

[Jensen 89] Kurt Jensen, "Coloured Petri-Nets", Lecture Notes in Computer Science, Vol 266, Springer-Verlag, 1989.

[Jerraya 91] Ahmed Jerraya, P.G. Paulin and S. Curry, "Meta VHDL for Higher Level Controller modeling and synthesis", Proc. of VLSI-91, Edinburgh, August 1991.

[Kleinrock 76] Leonard Kleinrock, Queueing Systems, Vol. I and II, John Wiley and Sons, New York, 1976.

[Klick 91] Vickie B. Klick, J. Patti and M. L. Todd, "Experiences in the use of SDL/GR in the software Development Process", SDL '91: Evolving Methods, O. Faergemand and R.Reed (Editors), Elsevier Science Pub., 1991, Pages 449-457.

[Koomen 91] C.J. Koomen, The Design of Communicating Systems: A System Engineering Approach, Kluwer Academic Publishers, Boston, 1991.

[Kumar 89] A. Kumar, V.Kashyap, S.D.Sherlekar, G.Venkatesh, S.Biswas, P.C.P. Bhatt and S. Kumar, "IDEAS: A Tool for VLSI CAD" IEEE Design and Test of Computers, Vol. 6, No. 5, October 1989, Pages 50-57.

[Lakos 91] C.A. Lakos, "LOOPN - Language for Object-Oriented Petri Nets", Technical Report 91-1, Department of Computer Science, University of Tasmania, 1991.

[Lanneer 91] Dirk Lanneer et al. "Architectural Synthesis for Medium and High Throughput Signal Processing with teh new CATHEDRAL environment", High-Level VLSI Synthesis, (edited by R. Camposano and W. Wolf), Kluwer Academic PublishersNorwell Massachusetts, 1991, Pages 27-54.

[McFarland 90] Michael C. McFarland, A.C. Parker and R. Camposano, "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 78, No. 2, February 1990, Pages 301-318.

[Milner 80] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[Moore 90] Andrew P. Moore, "The Specification and Verified Decomposition of Ssytem Requirements Using CSP", IEEE Transactions on Software Engineering, Vol. 16, No. 9, September 1990, Pages 932-948.

[Morrison 93] John D. Morrison and C.O. Newton, "ELLA", Fundamentals and Standards in Hardware Description Languages - NATO ASI series, Kluwer Academic Publishers, 199? Netherlands, Pages 385-394.

[Moszowski 85] Ben Moszowski, "A Temporal Logic for Multilevel Reasoning about Hardware", IEEE Computer magazine, Februaury 1985, Pages 10-19.

[Murata 89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, April 1989, Vol. 77, No. 4, Pages 541-580.

[Parent 91] Pierre Parent and O. Tanir, "Voltaire: a discrete event simulator", Proceeding of the PNPM-91 workshop (tools ammendment), Melbourne, December 1991.

[Peterson 81] James L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall inc., Englewood Cliffs, N.J., 1981.

[Petri 62] C. F tri, "Kommunikation mit Automaten (Communication with Automata)", Ph.D. dissertation, University of Bonn, Bonn, West Germany, 1962.

[Piloty 80] R. Piloty et al. "A overview of Conlan", Proc. of IFIP Congress, Tokyo, 1980.

[Pritsker 86] A.A. Pritsker, Introduction to Simulation and SLAM II, Halsted Press, West Lafayette, Indiana, 1986.

[Prycker 91] Martin de Prycker, Asynchronous Transfer Mode - Solution for Broadband ISDN, Ellis Horwood Ltd., Cornwall, England, 1991.

[Ramming 93] Franz J. Ramming, "System Level Design", Fundamentals and Standards in Hardware Description Languages - NATO ASI series, Kluwer Academic Publishers, 1993, Netherlands, Pages 109-151.

[Rattray 89] C. Rattray (Ed.), Specification and verification of concurrent systems, Springer Verlag, NY, NY, 1989.

[Reed 87] Daniel A. Reed and R.M. Fujimoto, Multicomputer Networks - Message Based Parallel Processing, The MIT Press, Cambridge, Mass., 1987.

[Reisig 82] Wolfgang Reisig, "Petri-Nets - An Introduction", EATCS Monographs on Theoretical Computer Science, Springer-Verlag, NY, 1982.

[Saultz 92] J.E. Saultz, RASSP Final Technical Report (CLIN 0002AB), CMDA972-92-R-0017, GE Aerospace - Advanced Technology Laboratories, Moorestown, New Jersey, October 21, 1992.

[Schriber 74] Thimas Schriber, Simulation using GPSS, John Wiley and Sons, New York, NY, 1974.

[Srivas 90] Mandayam Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor", IEEE Software Magazine, September 1990, Pages 52-64.

[Tanir 92] Tanir, O. V.K. Agarwal and P.C.P. Bhatt, "A System Level Synthesis Framework for Computer Architecture", Proc. of the third International Workshop on Rapid System Prototyping, Research Triangle Park, N.Carolina, June 23-25, 1992, pp.94-111.

[Tanir 93a] Tanir, O. V.K. Agarwal and P.C.P. Bhatt, "The Design of a Library Support System for a Telecommunication System Synthesis Environment", Proc. of the fourth International Workshop on Rapid System Proto-typing, Research Triangle Park, (N.Carolina, June 28-30), pp. 54-67.

[Tanir 93b] Tanir, O., V.K. Agarwal and P.C.P. Bhatt, "On the Design of Real-Time Telecommunication Systems", Proc. of IEEE workshop on Real-Time Applications, (New York, May 11-12).

[Tanir 93c] Oryal Tanir, V.K. Agarwal, P.C.P. Bhatt, "Architectural Telecommunication Modelling", Presented at the Co-Design Workshop,

Cambridge, USA, October 7 1993.

[Tanir 93d] Oryal Tanir, V.K. Agarwal and P.C.P. Bhatt, "System Design Exploration in a Specification Driven Simulation Environment", Proceedings of the European Simulation Symposium, SCS International, Delft, Netherlands, October 25-28, 1993, pp. 591-596.

[Tanir 94a] Oryal Tanir and S. Sevinc, "Defining the Requirements for a Standard Simulation Environement", IEEE Computer Magazine, February 1994, pp. 28-34.

[Tanir 94b] Oryal Tanir, V.K. Agarwal and P.C.P. Bhatt, "DASE: An Architectural Level System Design and Modelling Environment", Proceedings of the International Conference on Concurrent Engineering and Electronic Design Automation, SCS International, Bournemouth, UK, April 7-8, 1994, pp. 414-421.

[Tanir 94c] Oryal Tanir , V.K. Agarwal and P.C.P. Bhatt,"DASE: An environment for system level telecommunication design exploration and modelling", 4th International CAST 94 workshop, Ottawa, Canada, 16 May 1994, Proceedings to be published by Springer-Verlag.

[Thomas 91] D.E. Thomas and T.E.Fuhrrman, "Industrial Uses of the System Architect's Workbench", High-Level VLSI Synthesis, (edited by R. Camposano and W. Wolf), Kluwer Academic PublishersNorwell Massachusetts, 1991, Pages 307-330.

[Verilog 91] Verilog Hardware Description Language Reference Manual, Open Verilog International, 1991.

[VHDL 87] IEEE Standard VHDL Language Reference Manual, IEEE IStd 1076, 1987.

[Ward 85] Paul T. Ward and S.J. Mellor, Structured Development for Real-Time Systems, Vol. 1, Yourdon Press Computing Series, Prentice Hall Inc., Enlewood Cliffs, New Jersey, 1985.

[Wernik 91] Marek Wernik and E.A. Munter, "Broadband Public Network and Switch Architecture", IEEE Communications Magazine, January 1991, Pages 83-89.

[Zeigler 84] Bernard P. Zeigler, Multifacetted Modelling and Discrete Event Simulation, Academic Press, Orlando, Florida, 1984.

[Zegura 93] Ellen Witte Zegura, "Architecture for ATM Switching Systems", IEEE Communications Magazine, Februaury 1993, Pages 28-37.

Appendices

Appendix A - DSL pre-defined types

For constraints: A constraint type is a name given to label a given constraint in DSL. Constraint types are evaluated within Prolog using the test_constraint predicate. Users can code their own constraint types using this predicate. The predicate accepts four arguments: the type name, module, and message applied to as well as any numeric value associated with the type. The predicate is evaluated as true or fail depending upon any constaint violations on the defined type. For example, the built-in type upper_limit is coded as below:

```
test_constraint(upper_limit, Module, Mess, Value):-
        message_queue(Module,[(Mess,Qtime)|Rest]),
        sim_time(Curr_time),
        Queued_time is (Curr_time - Qtime),
        Value < Queued_time,
        print('   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!'),nl,print_sim_time,
        print('   DSL SIMULATOR: WARNING: Upper_limit Constraint Violation! '),nl,
        print_sim_time,print('   Module: '),
        print(Module), print(' Message: '),print(Mess),nl, print_sim_time,
        print('   Constrained Variable '), print(Queued_time),print(' exceeds upper limit of '),
        print(Value),nl,print_sim_time,print('   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!'),nl.

test_constraint(upper_limit, Module, Mess, Value):-
                print('DSL SIMULATOR: constraint satisfied for '),print(Module),
                print(' on message '),print(Mess),nl.
```

The built-in types are:

upper_limit: tests to see if an upper bound has been violated. Can be applied to a message, or resource.

lower_limit: test to see if a lower bound has been violated. As above.

test_message: tests to see if a message is currently being executed at any other module. If yes, the constaint fails

record: maintains a list corresponding to queue time for a given message. The list can later be used to plot statistics.

Appendix B - DSL DTSS example module listings

The following is a complete listing of the DTSS example introduced in the text.

```
/*line_
==========================================================================*/
ho_module(main(10, 0, 5, 2), [subscriber(X), mcgill_switch(X)]).
/* Main parameters are: no of telephones, switch no, no of channels/card, no of int. cards */

use_dsl_library([
                dts_switch,
                telephone
                ]).

isa(subscriber(X), user(X)).
isa(telephone(X), gen_tel(X)).

configure_library(main):-
        ho_module(main(T, SW, CH, CARD), XXX),
        asserta(isa(mcgill_switch(SW), dts_switch(SW, T, CARD, CH))),
        asserta(ho_module(mcgill_switch(SW), [])),
        configure_tel_con(T, SW).


configure_tel_con(1, SW):-
        asserta(module(subscriber(0), [])),
        asserta(module(telephone(0), [])),
        asserta(resource(subscriber(0), state, onhook)),
        asserta(path(subscriber(0), telephone(0), [hand(0), headset(0)])),
        asserta(path(telephone(0), subscriber(0), [headset(0), hand(0)])),
        asserta(path(telephone(0), mcgill_switch(SW), [tel_line(0), tel_line(0)])),
        asserta(path(mcgill_switch(SW), telephone(0), [tel_line(0), tel_line(0)])),
        write('Telephones connected to the switch...'),nl.

configure_tel_con(T, SW):-T>1,
        TT is T-1,
        asserta(module(subscriber(TT), [])),
        asserta(module(telephone(TT), [])),
        asserta(resource(subscriber(TT), state, onhook)),
        asserta(path(subscriber(TT), telephone(TT), [hand(TT), headset(TT)])),
        asserta(path(telephone(TT), subscriber(TT), [headset(TT), hand(TT)])),
        asserta(path(telephone(TT), mcgill_switch(SW), [tel_line(TT), tel_line(TT)])),
        asserta(path(mcgill_switch(SW), telephone(TT), [tel_line(TT), tel_line(TT)])),
        configure_tel_con(TT, SW).

start_cond:- retract(current_module_processed(XXX)),
        asserta(current_module_processed(subscriber(0))),
        send(start_a_call(telno(0,1,5)) ).

/* ==========================================================================*/
ho_module(dts_switch(NUMBER, INPUTS, CARDS, CHANNELS), [switch_net(NUMBER),
interface_card(Y)]).
```

```
use_dsl_library([
                generic_switch_element,
                interface_comp
                ]).

configure_library(dts_switch):-
                isa(INSTANCE, dts_switch(N,I,C,CH)),
                II is I//C,
                asserta(isa(switch_net(N), generic_switch_element(C,CH))),
                asserta(isa(interface_card(Y), interface_comp(II, CH))),
                asserta(path(interface_card(Y), switch_net(N), [into_port(Y), trunk_in(Y)])),
                asserta(path(switch_net(N), interface_card(Y), [trunk_out(Y), inti_port(Y)])),
                asserta(ho_module(switch_net(N), [])),
                example_config(N, II, C).

example_config(N, IN, 1):-
                Index is N*16,
                asserta(ho_module(interface_card(Index), [])),
                tel_port_config(Index, IN, 0).

example_config(N, IN, CARD):-
                CARD1 is CARD-1,
                Index is N*16+CARD1,
                asserta(ho_module(interface_card(Index), [])),
                tel_port_config(Index, IN, CARD1),
                example_config(N, IN, CARD1).

tel_port_config(Index, 0, CARD1):-
                nl.

tel_port_config(Index, IN, CARD1):-
                isa(INSTANCE, dts_switch(N,I,C,CH)),
                IPRIME is I//C,
                INN is IN-1,
                Index2 is IPRIME*CARD1 + INN,
                asserta(path(interface_card(Index), INSTANCE, [t_pair(Index2), tel_line(Index2)])),
                asserta(path(INSTANCE, interface_card(Index), [tel_line(Index2), t_pair(Index2)])),
                tel_port_config(Index, INN, CARD1).

/* ================================================================*/
module(generic_clock(INDEX, NO_OF_CHANNEL),
    [
    (clock_count(NO_OF_CHANNEL):- send(clock_count(0)),
            send(-,clock_port(INDEX),frame_cycle)),
    (clock_count(NEW):- NEW<NO_OF_CHANNEL,
                        Clock_rate is (125/NO_OF_CHANNEL),
            delay(Clock_rate),
            COUNT is NEW+1,
            send(-,clock_port(INDEX),clock(NEW)),
            send(clock_count(COUNT)))
    ]).

start_cond:- write('Setting up initial messages '),nl,
```

```
        isa(X, generic_clock(_, NO)),
        retract(current_module_processed(XXX)),
        asserta(current_module_processed(X)),
        send(clock_count(NO)).

configure_library(generic_clock).

/* ================================================================================*/
module(generic_linein(Z),
   [
   (channel_sync:- set_res(channel_count,0),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print('line_ line in card:channel sync received \n')
        ),
   (channel_message(DATA):- check_res(channel_count,CHANNEL),
                check_res(card_number, CARD),
                CHAN is (CHANNEL*16+CARD),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print('line_ line in card: updating channel '),
                print(CHANNEL),print(' with data: '),printl(DATA),
                CHAN2 is CHANNEL+1,
                set_res(channel_count,CHAN2),
                send(-, sp_port(Z), buffer_update(CHAN,DATA))
        )
   ]).

configure_library(generic_line_in):-
        isa(X,generic_line_in(_)),
        module(X, List),
        get_index(X, Value),
        asserta(resource(X, card_number, Value)),
        asserta(resource(X, channel_count,0)),
        fail.

configure_library(generic_line_in):-
        write(' generic_line_in: All resources successfully configured...'),nl.

/* ================================================================================*/
module(generic_lineout(Z),
   [
   (frame_cycle:-
                send(-,out_port(Z),channel_sync),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print(' DS-30 line output card: Frame sync message sent \n')
        ),
   (clock(0):-
                check_res(card_number, CARD),
                check_res(channel_out_buffer(CARD), MESSAGE),
                send(-,out_port(Z),channel_message(MESSAGE)),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print('Line out card number: '), print(CARD),
                print(' on channel 0 OUTPUTTED command '),
                printl(MESSAGE),
                set_res(channel_out_buffer(CARD), unused)
```

```
        ),
    (clock(COUNT):-
                check_res(card_number, CARD),
                COUNT2 is (COUNT*16+CARD),
                check_res(channel_out_buffer(COUNT2), VOICE),
                send(-,out_port(Z),channel_message(VOICE)),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print('line_ out card number: '), print(CARD),
                print(' : OUTPUTTED message '),
                print(VOICE),
                print(' in channel '),
                printl(COUNT)
        ),
    (update_buffer(ADD,DATA):- ADD>15,
        set_res(channel_out_buffer(ADD), DATA),
        print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
        print('line_ out card: new channel data updated \n'),
        print(' address is '), printl(ADD)
        ),
    (update_buffer(ADD,DATA):-
        check_res(channel_out_buffer(ADD),unused),
        set_res(channel_out_buffer(ADD), DATA),
        print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
        print('line_ out card: new channel data updated \n'),
        print(' address is '), printl(ADD)
        ),
    (update_buffer(ADD,DATA):-
        delay(0.5),
        send(update_buffer(ADD,DATA))
        )
    ]).

configure_library(generic_line_out):-
        isa(X,generic_line_out(Z)),
        isa(YYY,generic_switch_element(SIZE, RATE)),
        module(X, List),
        get_index(X, Value),
        asserta(resource(X, card_number, Value)),
        generic_line_out1(X,Value,RATE),
        fail.

configure_library(generic_line_out):-
        write(' generic_line_out: All resources successfully configured...'),nl.

generic_lineout1(X,Y, 0):- nl,
        write(' generic_line_out: Resource channel_out_buffer for '),
        write(X),write(' has been configured...'),nl.

generic_lineout1(Name,Index,COUNT):- COUNT>0, CC is COUNT-1,
        TEMP is (CC*16+Index),
        asserta(resource(Name, channel_out_buffer(TEMP), unused)),
        generic_line_out1(Name, Index, CC).

/* ================================================================================*/
```

```
module(generic_frame_control(Z, LIMIT),
        [
        (frame_start:- check_res(speech_frame_counter, COUNT),
                       check_res(number_of_in_cards, INPUT),
                       COUNT<LIMIT,
                       TEMP is LIMIT*INPUT,
                       FDELAY is 125/TEMP,
                       delay(FDELAY),
                       COUNT2 is COUNT+1,
                       print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                       print('FRAME CONTROL: new frame count = '),
                       printl(COUNT2),
                       set_res(speech_frame_counter, COUNT2),
                       send(channel_detect(COUNT2, INPUT))),
        (channel_detect(COUNT, INPUT):-
                       COUNT<INPUT,
                       send(sp(Z), sp_sync_port(Z), command_frame_select(COUNT)),
                       send(control_processor(Z), proc_in_port(Z), command_frame_select(COUNT)),
                       send(frame_start) ),
        (channel_detect(COUNT, INPUT):-
                       send(sp(Z), sp_sync_port(Z), frame_select(COUNT)),
                       send(control_processor(Z), proc_in_port(Z), frame_select(COUNT)),
                       send(frame_start) ),
        (frame_start:-check_res(speech_frame_counter,COUNT),
                       LIMIT2 is LIMIT-1,
                       COUNT>LIMIT2,
                       set_res(speech_frame_counter, -1),
                       send(frame_start) )
        ]).

configure_library(generic_frame_control):-
        isa(X,generic_frame_control(ZZ,_)),
        module(X, List),
        get_index(X, Z),
        asserta(resource(frame_control(Z), speech_frame_counter, -1)),
        generic_frame_control1(Z),
        fail.

configure_library(generic_frame_control):- write(' generic_frame_control: All resources successfully configured...'),nl.

generic_frame_control1(Z):-
        asserta(resource(frame_control(Z), number_of_in_cards, 0)),
        isa(X, generic_line_in(XX)),
        module(X,_),
        resource(frame_control(Z), number_of_in_cards, COUNT),
        COUNT1 is (COUNT+1),
        retract(resource(frame_control(Z), number_of_in_cards, XXX)),
        asserta(resource(frame_control(Z), number_of_in_cards, COUNT1)),
        write(' generic_frame_control: Number of line_ cards detected is '),
        write(COUNT1),nl,
        fail.

/* ================================================================================*/
```

```
module(generic_sp(X),
   [
   (buffer_update(ADDRESS,DATA):- delay(0.5),
                  set_res(channel_in_buffer(ADDRESS), DATA),
                  print('>>>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                  print('SP MODULE: In_buffer updated for channel '),
                  printl(ADDRESS) ),
   (frame_select(COUNT):- check_res(channel_in_buffer(COUNT), VOICE),
                  delay(0.4),
                  check_res(channel_in_buffer(COUNT), VOICE),
                  send(speech_bus(X),sp_port(X),speech_data(VOICE))),
   (frame_select(COUNT):- nl,print('>>>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                  print('SP module: speech bus unused at frame '),
                  printl(COUNT)),
   (command_frame_select(Channel):-
                  check_res(channel_in_buffer(Channel), COMMAND),
                  set_res(channel_in_buffer(Channel), unused),
                  send(-, proc_in_port(X), channel_command(Channel,COMMAND)))
   ]).

configure_library(generic_sp):-
        isa(NAME, generic_sp(I)),
        isa(DSNAME, generic_line_in(XX)),
        module(NAME,_),
        get_index(NAME, Index),
        module(DSNAME,_),
        get_index(DSNAME, Value),
        isa(NAME1, generic_switch_element(NNLINES,MMCHAN)),
        MM is MMCHAN-1,
        Low_bound is (Index*16-1),
        Up_bound is (Index*16+16),
        Low_bound < Value,
        Up_bound > Value,
        generic_sp1(Index,Value,MM),
        fail.

configure_library(generic_sp):-
        write(' generic_sp: All resources successfully configured...'),nl.

generic_sp1(NO, Y, -1):- write(' generic_sp: Resource channel_in_buffer for SP has been configured...'),nl.

generic_sp1(NO, Index,COUNT):- COUNT > -1, TEMP is (COUNT*16+Index),
        asserta(resource(sp(NO), channel_in_buffer(TEMP), line_indata)),
        Next is (COUNT-1),
        generic_sp1(NO,Index, Next).

/* ================================================================================*/
ho_module(generic_switch_element(M,R),
                                [
                                line_in(N), line_out(N),
                                outclock(N,R), frame_control(M),
                                time_switch(M), sp(M),
                                speech_bus(M),
                                control_processor(M), memory(M)
```

```
                                        ]).

use_dsl_library([
            generic_line_out,
                    generic_line_in,
            generic_frame_control,
            generic_sp,
            generic_time_switch,
            generic_clock,
                    pots_server
            ]).

isa(line_in(X), generic_line_in(X)).
isa(line_out(X), generic_line_out(X)).
isa(frame_control(X), generic_frame_control(X, 20)).
isa(outclock(X), generic_clock(X, _)).
isa(sp(X), generic_sp(X)).
isa(time_switch(X), generic_time_switch(X)).
isa(control_processor(M), pots_server(M)).

/*------------------------- Speech Bus Module      ------------------*/

module(speech_bus(M),
    [
      speech_data(VOICE):- set_res(persistent_port,(speech_line(M), VOICE))
    ]).

/*------------------------- Control Processor Module ----------------*/

module(control_processor(M),
    [
    (frame_select(COUNT):-
      delay(0.05), send(-, mem_port(M), mem_rd_rqst(COUNT))
          ),
    (memory_data(DATA):-
      delay(0.02), send(-,time_port(M),address_select(DATA))
          ),
    (command_frame_select(CH):-print('Control Proc: Channel command period : '),printl(CH)
          ),
    (line_indata:- print(' No command message detected for current channel \n')
          ),
    (channel_command(CH, COMMAND):-
          send(COMMAND)
          )
    ]).

/*-------------------------memory behaviour -------------------------
memory read delay is set at 0.1 microsecs= 100 nsecs
*/

module(memory(M),
      [
      (mem_rd_rqst(ADDRESS):- delay(0.1),
            print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
```

```
            print('word being read from memory address of '),
                printl(ADDRESS),
                check_res(route_table(ADDRESS), DATA),
                send(control_processor(M), proc_port(M), memory_data(DATA))),
        (mem_print(A,D):- delay(0.1),
                set_res(route_table(A), D),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                printl('routing table in RAM updated. ')
        )
]).
constraint(sp(M), frame_select(X), upper_limit, LIMIT):-
                resource(frame_control,number_of_in_cards, INPUT),
                LIMIT is 3.90625/INPUT.
constraint(control_processor(M), frame_select(X), upper_limit, 0.051).

configure_library(generic_switch_element):-
        isa(X,generic_switch_element(SIZE, RATE)),
        ho_module(X, []),
        get_index(X, Index),
        B_SIZE is (2*SIZE*RATE),
        retract(isa(INSTANCE,generic_clock(CARENOT, _))),
        asserta(isa(INSTANCE,generic_clock(CARENOT, RATE))),
        generic_switch_element1(X, Index, SIZE),
        generic_switch_element2(Index),
        create_route_table(Index, B_SIZE),
        fail.

configure_library(generic_switch_element):- write(' generic_switch_element: All resources successfully
configured...'),nl.

generic_switch_element1(X, Index, 0):- write('Generic switch ('),write(Index),write('): Created internal
modules.'),nl.

generic_switch_element1(X, Index, Size):- Size > 0,
        Next_size is (Size-1),
        Index1 is (Index*16+Next_size),
        asserta(module(line_out(Index1), [])),
        asserta(module(line_in(Index1), [])),
        asserta(module(outclock(Index1), [])),
        asserta(path(time_switch(Index),          line_out((Index1)),          [line__port((Index1)),
ts_line__port((Index1))])),
        asserta(path(line_in(Index1), sp(Index), [sp_port(Index1), ds_port(Index)])),
        asserta(path(outclock(Index1), line_out(Index1), [clock_port(Index1), dscp(Index1)])),
                asserta(path(X, line_in(Index1), [trunk_in(Index1), in_port(Index1)])),
                asserta(path(line_out(Index1), X, [out_port(Index1), trunk_out(Index1)])),
                asserta(path(control_processor(Index),      line_out(Index1),      [line_port(Index1),
ds_line_port(Index1)])),
        generic_switch_element1(X, Index, Next_size).

generic_switch_element2(Index):-asserta(module(sp(Index), [])),
        asserta(module(time_switch(Index), [])),
        asserta(module(frame_control(Index), [])),
        asserta(resource(speech_bus(Index), persistent_port, (speech_line(Index), voice_data))),
        asserta(path(sp(Index), speech_bus(Index), [sp_port(Index), speech_line(Index)])),
```

```
        asserta(path(sp(Index), control_processor(Index), [proc_in_port(Index), sp_int_port(Index)])),
        asserta(path(time_switch(Index), speech_bus(Index), [speech_line(Index), speech_line(Index)])),
        asserta(path(frame_control(Index), control_processor(Index), [proc_in_port(Index),
frame_port(Index)])),
        asserta(path(frame_control(Index), sp(Index), [sp_sync_port(Index), frame_sync(Index)])),
        asserta(path(memory(Index), control_processor(Index), [proc_port(Index), mem_port(Index)])),
        asserta(path(control_processor(Index), memory(Index), [mem_port(Index), proc_port(Index)])),
        asserta(path(control_processor(Index), time_switch(Index), [time_port(Index),
proc_port(Index)])).

create_route_table(Index, -1):-
        write('setup routing table...'),nl.
create_route_table(Index, SIZE):-
        SIZE1 is SIZE-1, SIZE > -1,
        asserta(resource(memory(Index), route_table(SIZE),SIZE)),
        create_route_table(Index, SIZE1).

/*------------------------ Time Switch Module ------------------------*/

module(generic_time_switch(Z),
        [
        (address_select(ADDRESS):- delay(0.1),
                check_res(address_map(ADDRESS), CARD),
                probe(speech_line(Z), DATA),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print('Time_switch: latched onto data: '), print(DATA),
                send(-, line__port(CARD), update_buffer(ADDRESS,DATA)) ),
        (address_select(ADD):-
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      Time_switch: Address '),
                print(ADD),print(' undefined. \n')
        )
        ]).

configure_library(generic_time_switch):-
        isa(X,generic_time_switch(XX)),
        isa(NAME, generic_line_out(_)),
        module(X,_),
        get_index(X, I),
        Low_bound is (I*16-1),
        Up_bound is (I*16+16),
        module(NAME,_),
        get_index(NAME,Index),
        Low_bound < Index,
        Up_bound > Index,
        resource(NAME, channel_out_buffer(Value), LL),
        asserta(resource(X, address_map(Value), Index)),
        fail.
configure_library(generic_time_switch):-
        write(' generic_time_switch: All resources successfully configured...'),nl.

constraint(time_switch(Z), address_select(A), upper_limit, 0.11).

/* ================================================================*/
ho_module(interface_comp(INPUT, CH),
```

```
                    [
                    line_card(CARD), controller(CARD), clock(CARD)
                    ]).

use_dsl_library([
                    sub_server,
                    line_card,
                    generic_clock
                    ]).

configure_library(interface_comp):-
                    isa(X, interface_comp(INPUT,CH)),
                    ho_module(X, YYY),
                    get_index(X,CARD),
                    asserta(module(controller(CARD), [])),
                    asserta(module(line_interface(CARD), [])),
                    asserta(module(timer(CARD), [])),
                    asserta(path(X, line_interface(CARD), [inti_port(CARD), lin_port(CARD)])),
                    asserta(path(line_interface(CARD), X, [lout_port(CARD), into_port(CARD)])),
                    asserta(path(line_interface(CARD),          controller(CARD),          [proc_port(CARD),
proc_port(CARD)])),
                    asserta(path(controller(CARD),          line_interface(CARD),          [proc_port(CARD),
proc_port(CARD)])),
                    asserta(path(timer(CARD),line_interface(CARD),
[clock_port(CARD),clock_port(CARD)])),
                    asserta(isa(line_interface(CARD), line_card(CARD, CH))),
                    asserta(isa(timer(CARD), generic_clock(CARD, CH))),
                    asserta(isa(controller(CARD), sub_server(CARD, INPUT))),
                    config_controller(X, CARD, CH),
                    fail.
configure_library(interface_comp):- nl,nl, write(' Completed configuring interface components.'),nl.

config_controller(X, CARD, 0):-
                    write('set-up controller paths...'),nl.

config_controller(X, CARD, CH):-CH>0,
                    CC is CH-1, C is CARD*5,CH2 is C+CC,
                    asserta(path(X, controller(CARD), [t_pair(CH2), twisted_pair(CH2)])),
                    asserta(path(controller(CARD), X, [twisted_pair(CH2), t_pair(CH2)])),
                    config_controller(X, CARD, CC).

/* ================================================================================*/
module(line_card(Z, INPUT),
   [
   (frame_cycle:-
                    send(-,lout_port(Z),channel_sync),
                    print('>>>>>>>>>>>>>>>>>>>>>>>>      '),
                    print(' line output card: Frame sync message sent \n')
         ),
    (clock(0):-
                    check_res(channel_buffer(0), VOICE),
                    send(-,lout_port(Z),channel_message(VOICE)),
                    print('>>>>>>>>>>>>>>>>>>>>>>>>      '),
                    print('Line card : '),
```

```
                print('channel 0 OUTPUTTED command '),
                printl(VOICE),
                set_res(channel_buffer(0), unused)
        ),
  (clock(COUNT):-
                check_res(channel_buffer(COUNT), VOICE),
                send(-,lout_port(Z),channel_message(VOICE)),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print(' line out: OUTPUTTED message '),
                print(VOICE),
                print(' in channel '),
                printl(COUNT)
        ),
  (update_buffer(ADD,DATA):- check_res(channel_buffer(ADD),XX),
                set_res(channel_buffer(ADD), DATA)
        ),
  (channel_sync:- set_res(channel_count,0),
                print('>>>>>>>>>>>>>>>>>>>>>>>>>>      '),
                print('line card:channel sync received \n')
        ),
  (channel_message(DATA):- check_res(channel_count,CHANNEL),
                CHAN2 is CHANNEL+1,
                set_res(channel_count,CHAN2),
                send(-, proc_port(Z), channel_update(CHANNEL,DATA))
        )
 ]).

configure_library(line_card):-
        isa(X,line_card(Z, CH)),
        module(X, List),
        asserta(resource(X, channel_count, 0)),
        get_index(X, Value),
        configure_buffers(X,Value,CH),
        fail.

configure_library(line_card):-
        write(' line card: All resources successfully configured...'),nl.

configure_buffers(X,Y, -1):- nl,
        write(' line_card: Resource channel_out_buffer for '),
        write(X),write(' has been configured...'),nl.

configure_buffers(Name,Index,COUNT):- COUNT > -1,
        asserta(resource(Name, channel_buffer(COUNT), unused)),
        Decrement is (COUNT-1),
        configure_buffers(Name, Index, Decrement).
```

Appendix C - Experimental Results

This section summarizes the results of two experiments performed using the DASE tool. The experiments are presented to elaborate some of the simulation performance capabilities of the DSL simulator. The first simulation is several simulations of the DTSS switch example presented in sections 2 and 3 of the dissertation. The second simulation is that of the ATM switch case study presented in section 4. All simulations were performed on a Sun 10/30 platform with 64 Mbytes of RAM. The workstation was dedicated only for the simulations. The resulting figures include operating system overhead.

1. DTSS simulation:

A DTSS switch was simulated with different numbers of subscribers. The numbers of subscribers were:
2, 64, 640, 64,000, and 126,000.

Telephone traffic was simulated to last for exponential intervals between 2-8 seconds per call. Each simulation run was for a simulated period of 15 minutes of switching activ-

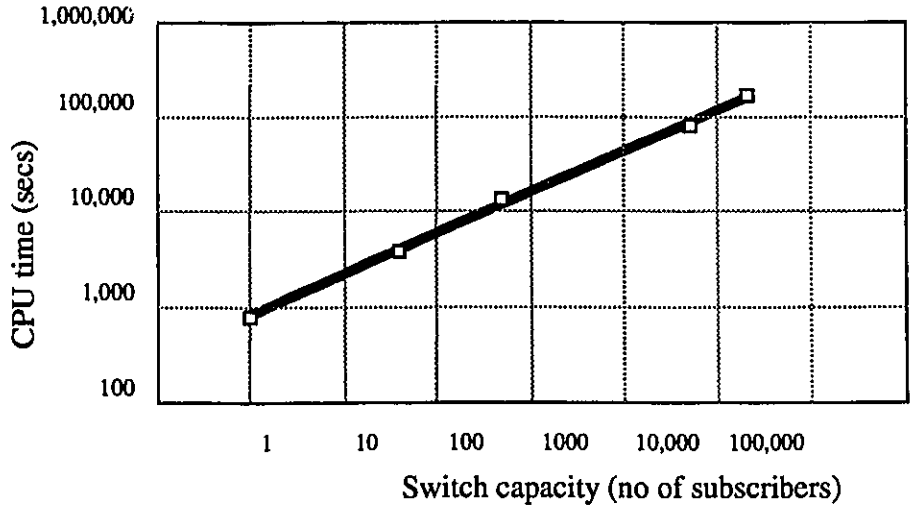


FIGURE A-1. 15 minute simulation of telephone traffic

ity. The figure above presents the performance figures for the example. The time to simulate increases linearly as the complexity (number of telephone subscribers) increases. At the extremes, a two subscriber interaction can be simulated under 13 minutes of CPU time, whereas a 126,000 subscriber simulation will take over 33 hours of simulation time.

2. ATM simulation:

An ATM switch (as described in section 4) was simulated. The simulation consisted of different switch sizes of 2, 4,8 and 16 input/output ports. Each port was loaded by a con-

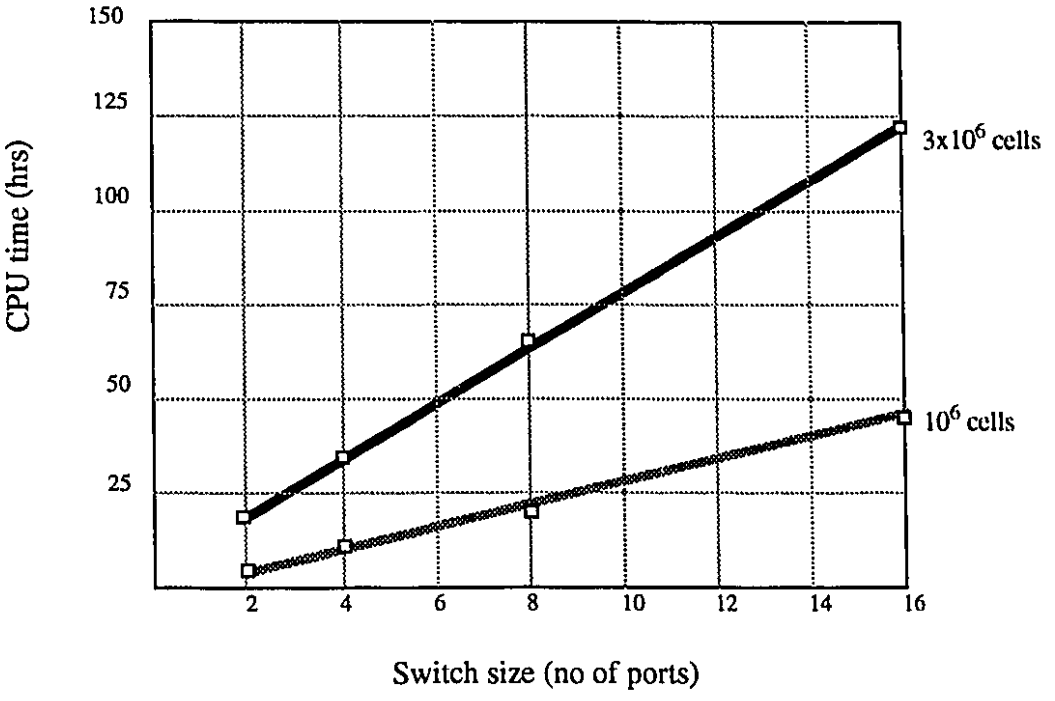


FIGURE A-2. Simulation of different ATM switch sizes

tinuous traffic stream of ATM cells at a rate of 155 Mb/sec. this results in an effective cell rate of 2.8 microseconds / cell. A recording of the CPU time for each configuration was taken at 2.8 and 8.4 seconds of simulated time. Approximately 1 million cells were generated at each port every 2.8 seconds.

The results are plotted in figure A-2. As can be observed, the simulation times are generally extensive. However the times increase linearly.

Conclusions

The simulations indicate a linear progression of simulation times as model complexity increases. The simulator performs reasonably well, however for demanding simulations such as the ATM switch, accelerated simulation techniques would be desirable in order to obtain quicker results. Designer experience and knowledge is truly required to reduce unnecessary simulation runs and focus on the important aspects of a design.

Appendix D - DSL - VHDL Differences

This section highlights the major differences between DSL and a specialized hardware description language VHDL. The semantics of the languages are similar in some respects but strongly different in many others.

It is worthwhile understanding some of the major issues surrounding design automation tool application. As one moves from an abstract level of design (architectural) to more detailed levels (such as RTL), there is a strong shift of detail in the models created at each level. At the architectural level, little timing detail is provided compared to the lower levels. However, freedom is given to structural aspects of the model so that design exploration can be facilitated. At this point of design, a certain level of ambiguity is introduced which is result of the flexibility in modeling with loosely typed constructs.

Ambiguity must be resolved by the designer and environment until a synthesizeable design is reached. At this point, when synthesis to a lower level is possible, a mapping must take place from low timing detail to higher timing detail. To accommodate this, the environment must be able to fill the essential details. Hence some type of compromise is required by the synthesis tool (such as cost, area or performance) so that details can be added. It is this fact that can make synthesis tools very poor performers as compared with experienced human counterparts.

The strong distinction between design exploration (a key requirement at the architectural level) and design formalization (a major requirement between the lower levels of abstraction) dictate the need for different languages. Hence, DSL is an internal language to support the former whereas VHDL the latter.

The major difference in the languages is the level of abstraction in which they are to be applied. VHDL's applicability is approximately limited to the RTL to circuit level design. Application to higher levels is also probable, but unlikely for several reasons. The language contains many simulator oriented semantics (such as wait on event of a simulator) which are superfluous at higher levels of design. The language carries primitives which are not highly desirable for architectural level designers (especially not for software designers) such as signals and transport (low level timing) commands. Hence architectural level representation could be possible, but would be highly cumbersome using a language that was not intended for this purpose.

VHDL is a good language for hardware formalization, but very poor for design exploration. This is not a negative feature, but again related to the area of applicability of VHDL. As an RTL description language, it provides strong typing of communication and timing between entities. However, as the abstraction level increases, detail is lost and less restrictions on the typing of the entities is required to permit design exploration. This necessitates the use of a language such as DSL.

DSL and VHDL both use a simulator for timing verification. However, simulator semantics are not embedded within the DSL language like it is for VHDL. This is a highly desir-

able feature of an architectural language since it permits formal verification. Such techniques have been cumbersome and fleeting for VHDL representations, whereas Petri-net based verification (as well as others not explored) can be applied to DSL modules.