Diss. ETH No. 11869

An Object Oriented Petri Net Approach to Embedded System Design

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
ROBERT ESSER
B.E. University of Adelaide
born 31st of August, 1958
citizen of Australia and the Netherlands

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Wolfgang Fichtner, co-examiner

1996

VERSION 18 October 1996 9:43 am

Copyright 1996 by Robert Esser.

All rights reserved.

Abstract

A methodology is described for the automated design of complex systems that are embedded in a technical environment and are subject to real time constraints. The functionality of such systems is typically implemented in the form of mixed hardware and software subsystems where the partitioning of an implementation into various hardware and software based devices is dependent on various constraints such as cost, performance, power consumption, area and maintainability.

The proposed methodology is built upon a formal kernel language consisting of a class of high level timed Petri nets, augmented by object-oriented mechanisms:

- formal model enables unambiguous specification and supports a high level of analysis;
- homogeneous language (does not rely on an external implementation language) is general enough to support other more specialised formalisms, such as finite state machines and data flow graphs;
- modelling of real time requirements is supported, where certain time relationships between particular inputs and outputs are prescribed.

Abstraction and refinement allow complex systems to be modelled and successively refined towards an implementation. A system is a composition of communicating components at every level of abstraction. During the design of an embedded system classes are defined, refined, reused and configured:

- facilities for encapsulating behaviour and state within components (described with classes) are provided by the kernel language;
- classes define how inheritance properties are propagated and provide mechanisms for refinement, abstraction, configuration, reuse and sharing;
- refinement is supported by substituting less refined components with components containing an increased amount of detail.

The flexibility supported by the formal language is available to all other kernel based formalisms, as they are also object oriented and can be refined and reused in the same manner as the kernel language. This approach is an extension of well known concepts and the resulting system is powerful and easy to understand.

Kurzfassung

In dieser Arbeit wird eine Methodik für den automatischen Entwurf von komplexen Systemen definiert, die in ein technisches Umfeld eingebunden sind und an die Echtzeitanforderungen gestellt werden. Solche Systeme werden typischerweise sowohl aus Hardware- als auch aus Softwarekomponenten zusammengesetzt. Die tatsächliche Aufteilung in Hardware und Software hängt dabei von vielfältigen Anforderungen ab, wie beispielsweise Kosten, Performanz, Leistungsverbrauch, Fläche oder Wartungsfreundlichkeit.

Der hier vorgeschlagene Ansatz baut auf einer formalen Kernelsprache auf, die aus einer Klasse von High-level Petri-Netzen besteht und um objekt-orientierte Mechanismen erweitert ist. Sie ist charakterisiert durch:

- Ein formales Modell, das eine eindeutige Spezifikation ermöglicht und Analysemethoden bereitstellt;
- Eine homogene Sprache, die sich nicht auf eine externe Implementationssprache stützt, und die allgemein genug ist, um speziellere Formalismen wie endliche Zustandsmaschinen oder Datenflussgraphen zu unterstützen;
- Eine Kernelsprache, die die Modellierung von Echtzeitbedingungen, d.h. zeitlichen Abhängigkeiten zwischen beliebigen Ein- und Ausgängen unterstützt.

Durch Abstraktion und Verfeinerung ist es möglich, komplexe Systeme zu modellieren und schrittweise zu einer Implementation zu verfeinern. Auf jeder Abstraktionsebene besteht ein System aus kommunizierenden Komponenten. Während des Entwurfs eines eingebetteten Systems werden Klassen definiert, verfeinert, wiederverwendet und konfiguriert:

- Die Kernelsprache stellt mittels Klassen Möglichkeiten zum Einkapseln von Verhalten und Zustand der Komponenten zur Verfügung;

- Klassen geben an, wie Eigenschaften vererbt werden und ermöglichen das Konfigurieren, Wiederverwenden und Mitbenutzen von Komponenten;
- Verfeinerung wird durch das Ersetzten von Komponenten durch Komponenten mit höherem Detailierungsgrad unterstützt.

Die Flexibilität der formalen Sprache kann für alle anderen Formalismen, die auf dem Kernel aufbauen, genutzt werden, da diese Formalismen ebenfalls objekt-orientiert sind und genauso wie die Kernelsprache verfeinert und wiederverwendet werden können. Dieser Ansatz ist eine Erweiterung bekannter Konzepte und das resultierende System ist mächtig und leicht verständlich.

Contents

CHAPTER 1 *Introduction* ***5***

1.1 Embedded Systems **6**
1.2 Codesign **10**
1.3 Formal Kernel Language **10**
1.4 Object Orientation **11**
1.5 Overview **12**

CHAPTER 2 *Requirements for a Codesign Methodology* ***13***

2.1 Major Abstraction Levels **16**
2.1.1 Behavioural Level **16**
2.1.2 Structural Level **16**
2.1.3 Physical Level **17**
2.2 Additional Requirements **17**
2.2.1 Multi-Paradigm Support **18**
2.2.2 Refinement **18**
2.2.3 Time **19**
2.2.4 Configuration **19**
2.2.5 Design Exploration **20**
2.2.6 Open Framework for Partitioning and Interface Generation **21**
2.3 Summary **23**

CHAPTER 3 *Existing Approaches* ***25***

3.1 Models **25**
3.1.1 Models for Reactive Systems **26**
3.1.2 Models for Transformational Systems **31**
3.1.3 Control / Data Flow Models **32**
3.1.4 General Purpose Languages **34**
3.1.5 Remarks on Models for Embedded System Design **38**
3.2 Methodologies **39**
3.2.1 Heterogeneous System Design Methodologies **39**
3.2.2 General Purpose Methodologies **44**
3.2.3 General Purpose Object Oriented Methodologies **47**

CHAPTER 4 *Proposed Approach* ***51***

4.1 Object Orientation **52**
4.2 Modelling Language **53**
4.2.1 Components **56**
4.3 Proposed Methodology **58**
4.4 Summary **62**

CHAPTER 5 *Kernel Language* ***63***

5.1 Petri Nets **65**
5.1.1 Place-Transition Petri Nets **66**
5.1.2 Petri Nets with Inhibitor Arcs **72**
5.1.3 High Level Petri Nets **75**
5.1.4 High Level Time Petri Nets **82**
5.2 Combining Object Oriented Concepts and Petri Nets **95**
5.2.1 Existing Object Oriented Petri Net Approaches **98**

5.3 Object Oriented Time Petri Nets **100**
5.3.1 Class Definition **100**
5.3.2 Inheritance **103**
5.3.3 Example **106**
5.3.4 Components **112**
5.3.5 Configuration **113**
5.4 Summary **114**

CHAPTER 6 *Support for Other Formalisms* ***115***
6.1 Other Approaches **117**
6.2 Describing Formalisms with Graph Grammars **118**
6.2.1 A Graph Grammar Notation **118**
6.2.2 Describing a Formalism **121**
6.3 Example: StateCharts **127**
6.3.1 Modelling Finite State Machines with Petri Nets **128**
6.3.2 Supporting StateCharts **131**
6.3.3 Defining the Syntax Graph Grammar **132**
6.3.4 Defining the Class Hierarchy **135**
6.3.5 Defining the Semantic Graph Grammar **137**
6.3.6 Defining the Graphical Representation **143**
6.4 Other Formalisms in an Object Oriented Environment **143**
6.5 Summary **148**

CHAPTER 7 *Example* ***149***
7.1 Requirements **149**
7.2 System and Environment **150**
7.3 Scenarios **151**
7.4 Environment Model **155**
7.5 System Structure **160**

7.6 System Model **161**
7.7 Refinement and Configuration **163**
7.8 Summary **167**

CHAPTER 8 *Conclusions* ***169***

Appendices ***171***

A Implementation **171**
A.1 Language **172**
A.2 Project **173**
A.3 Class **174**
A.4 Component **176**
B Language Syntax **181**
B.1 General **181**
B.2 Class **182**
B.3 Export (Interface) **183**
B.4 Connection **183**
B.5 Place **183**
B.6 Transition **183**
B.7 Function **184**
B.8 Expression **184**
B.9 Code **185**
B.10 Sundries **186**

Literature ***187***

Index ***201***

CHAPTER 1 *Introduction*

This monograph presents a methodology for the design of embedded real time systems. This methodology is object oriented and is based upon a high level Petri net kernel language. Gains achieved in efficiency and mastering complexity when designing embedded systems are largely due to the introduction of methodologies that add additional levels of abstraction. This enables designers to view the ever increasing system complexity at appropriate levels of detail. The methodology presented in this monograph does not replace existing methodologies but complements the strengths of existing approaches by effectively adding an extra level of abstraction to better cope with the increasingly heterogeneous nature and complexity of today's systems.

Embedded systems often have complex behaviour that may not be able to be completely or easily described in a single formalism, therefore it is desirable to support a number of formalisms that are well suited to modelling particular parts of an embedded system. Using the object oriented Petri net based kernel language it is shown how other, more application specific formalisms, can be defined in terms of the kernel language. The advantage of this approach is that the interaction semantics between formalisms are given by the underlying formal kernel language resulting in a well defined formal system model. The designer has the ability to use the most appropriate formalism to model particular components of an embedded system.

Furthermore embedded systems are often implemented on heterogeneous components, for example general purpose microcontrollers, application spe-

cific hardware, specialised programmable hardware (digital signal processors etc.). Cost effectiveness requires that an implementation architecture is chosen that meets a particular set of functional and non functional constraints. The proposed methodology offers mechanisms for design exploration based on flexible system configuration functions and version management.

1.1 Embedded Systems

The number of embedded systems has increased rapidly over the last few years. This growth has been driven by the extensive use of electronics in systems such as automobiles, home appliances and communication devices. The increased use of electronics has resulted in systems being more effective, more reliable and less expensive and has been accompanied by an increase in the use of computers in the form of microprocessors and microcontrollers.

Figure 1 shows the percentage of different types of microprocessors and microcontrollers sold in 1994 from a total of 2.8 billion. Assuming that 60 million personal computers and workstations were sold, each with approximately 10 microprocessors, the remainder of around 2.2 billion processors found their way into embedded systems. Embedded systems constitute a huge market that in many respects is more cost, time and performance critical than that of personal computers and work stations.

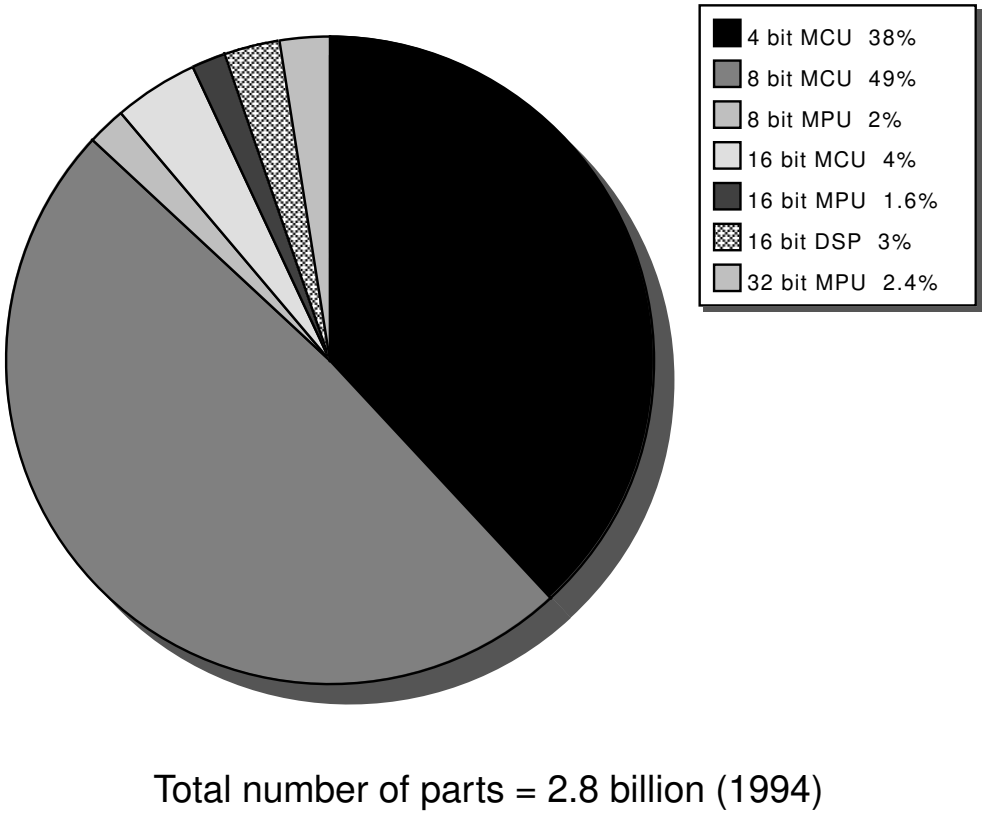

FIGURE 1. Volume of microprocessors and microcontrollers sold in 1994. Source: World Semiconductor Trade Statistics, 1994

There are various definitions of what constitutes an embedded system. The term *embedded system* is often synonymous with *process control system* and *real time system* and encompasses systems ranging in size and complexity from vacuum cleaner controllers to complete aircraft control systems and beyond. The following definition is based on that presented in [75]:

An embedded system is an electronic system embedded within an external process. It influences the external process to ensure that functional and performance requirements are met. The external process may be composed of subprocesses and include human interaction through a man machine interface (MMI).

Figure 2 shows a schematic of an embedded system.

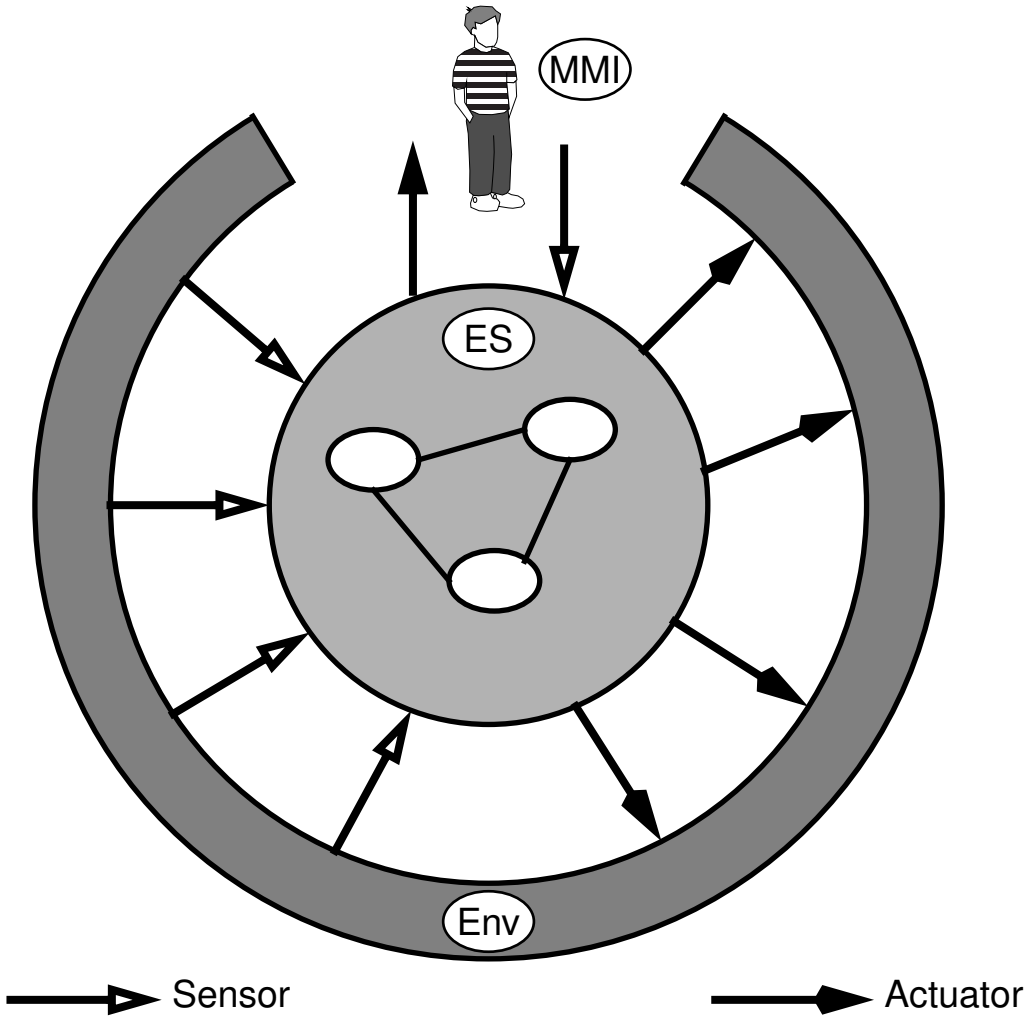

FIGURE 2. An embedded system (ES) embedded within its environment (Env) and interacting with a user (MMI)

Embedded systems are best characterised in terms of their interaction with their environment, which then implies certain internal properties. Typical of external processes making up the environment of an embedded system is that they have their own dynamics which typically, in case of failure, can not be restored to a previous state. This, in addition to the requirement that certain inputs have to be serviced within prescribed time limits, set hard constraints on the embedded system that must be met.

Embedded systems are coupled with their environment through sensors and actuators. This coupling may be broadly classed into three main types:

1. an asynchronous event having a predetermined meaning, e.g. a valve is open;
2. a message in which some state of a process is encoded;
3. continuous signals are usually sampled arriving or being produced at a deterministic time.

Real time requirements prescribe certain time relationships between particular inputs and outputs. Furthermore the environment of an embedded system may be physically distributed, requiring a communication system with perhaps distributed processing. An embedded system might not be homogeneous. Internal processes may have significantly different tasks often resulting in heterogeneous implementations with the use of specialised hardware and software components. In addition to processors and the associated software, embedded systems may consist of application specific integrated circuits (ASIC), field programmable gate arrays (FPGA) and devices that interact with the analogue world. The design problem is then how to design an embedded system that will have software parts running on processors and hardware components with usually a fixed or limited functionality working harmoniously together. There may be many solutions to implement an embedded real time system that meets the functional and non functional design constraints. Furthermore many embedded systems have a complexity that can only be implemented using specific models within a methodology specifically tailored for embedded system design.

1.2 Codesign

In traditional embedded system design approaches, designers make the hardware and software partitioning decisions at an early stage in the development cycle. The individual parts are then designed independently with little interaction, due to the lack of a unified representation, simulation and synthesis framework.

Modern embedded systems require a more flexible design strategy in which heterogeneous system components are designed in parallel using specialised models. The designer makes partitioning decisions after first evaluating alternative structures with respect to performance, programmability, area, power, recurring development costs, manufacturing costs, reliability and maintenance. This strategy requires tools that support unified hardware and software representation, heterogeneous models and simulation and hardware and software synthesis. This strategy is called codesign.

1.3 Formal Kernel Language

To support embedded system design, models are required that are formal i.e. they can be executed and verified. Formal methods have been advocated by the scientific community for over a decade. They are a prerequisite for unambiguous specification and support a higher level of analysis than informal or semi formal methods, (the ability to prove a degree of correctness and reliability). Formal methods guarantee that a wide range of semantic errors are detected so that they can be removed early in a system's life cycle.

In this monograph a formal model based on high level time Petri nets is presented. Unlike many formal languages Petri nets have an easily understood graphical representation. This representation allows difficult concepts such as concurrency and synchronisation to be visualised in an intuitive manner. In addition, a large body of theory exists for Petri nets enabling system characteristics to be verified. Petri nets also provide the formal basis upon which

many other formalisms are constructed enabling multiple formalisms to be used within a system model. This allows the optimal (for whatever reason) formalism to be used for a particular subsystem part within a formal framework.

1.4 Object Orientation

Object oriented modelling and design uses objects to represent real world entities or abstractions of systems. A system consists of a hierarchical composition of objects which combine data and a set of operations for transforming the data (*behaviour*).

Object oriented concepts are well suited to address a number of important issues relevant to embedded system design such as abstraction, refinement and concurrency. Object oriented models provide an intuitive way of supporting data abstraction and information hiding by encapsulating the data within an object, effectively hiding it from other objects. This ensures that internal changes to one object will not affect any other object.

The object oriented model can realistically represent concurrency, as objects have independent existences and behaviour and interact with other objects to model complex systems.

1.5 Overview

This monograph consists of eight chapters as follows:

- in chapter 1 embedded systems and heterogeneous system design are introduced and an overview is given of the key features of the approach taken;
- in chapter 2 the requirements are discussed for an embedded system design methodology, specifically the aspects pertaining to the heterogeneous nature of these systems and how they can be modelled, validated, verified and implemented;
- in chapter 3 existing models and methodologies for embedded system design and also more general approaches for system design are presented;
- in chapter 4 a methodology is proposed based on a formal kernel language within an object oriented framework;
- in chapter 5 the kernel language is formally defined. It consists of high level time Petri nets used within components defined using object oriented concepts. The particular class of Petri nets is shown to be composed of four standard Petri net classes;
- in chapter 6 a method is presented for supporting multiple formalisms. This is based on concepts from graph grammars in which other formalisms are described in terms of the kernel language;
- in chapter 7 an example illustrates the application of the proposed methodology;
- in chapter 8 some conclusions are drawn.

Finally, in the appendices, information is given on a tool that implements the proposed methodology. This tool, designed and written by the author, is the basis for further research into the design of embedded systems at the ETH. Current work is being carried out on formal analysis, partitioning and interface synthesis of models described with the kernel language, in addition to distributed simulation and implementation generation.

CHAPTER 2 *Requirements for a Codesign Methodology*

The requirements for a design methodology have changed dramatically over the last two decades. Particularly in the area of integrated circuit design, the number of available gates has increased by at least three orders of magnitude. This rapid increase in size has also been accompanied by an increase in the complexity of the implemented systems and where previously only one function could be implemented on an integrated circuit, now complete systems are being implemented. Similarly embedded systems have grown in complexity often requiring distributed, heterogeneous implementations.

Figure 3 shows the different abstraction levels defined to implement systems on an integrated circuit. The mask level was initially used to design the first integrated circuits. The next popular design methodology was based on capture and simulate methodology, where the designer used predefined blocks to implement the required functionality. The system was then simulated at the signal level and when considered correct, a layout was generated. This logical level abstracted away from the preceding switch, electrical and mask levels allowing the designer to concentrate more on functionality rather than low level implementation details. The capture and simulate methodology was followed by a functional methodology where the designer described a system's functionality in terms of boolean equations, truth tables etc. These elements were then automatically translated into blocks which were placed and routed to a form, from which a geometry was generated.

The previous methodologies can be considered as being exclusively hardware based. The last few years have seen the introduction of behavioural

descriptions where the behaviour of a system is described using a standard language. This description is transformed using state and logic minimisation, validated through simulation and subsequently through the use of logic synthesis tools transformed into blocks etc. Within the behavioural level abstraction has increased allowing more abstract and therefore more complex systems to be efficiently described.

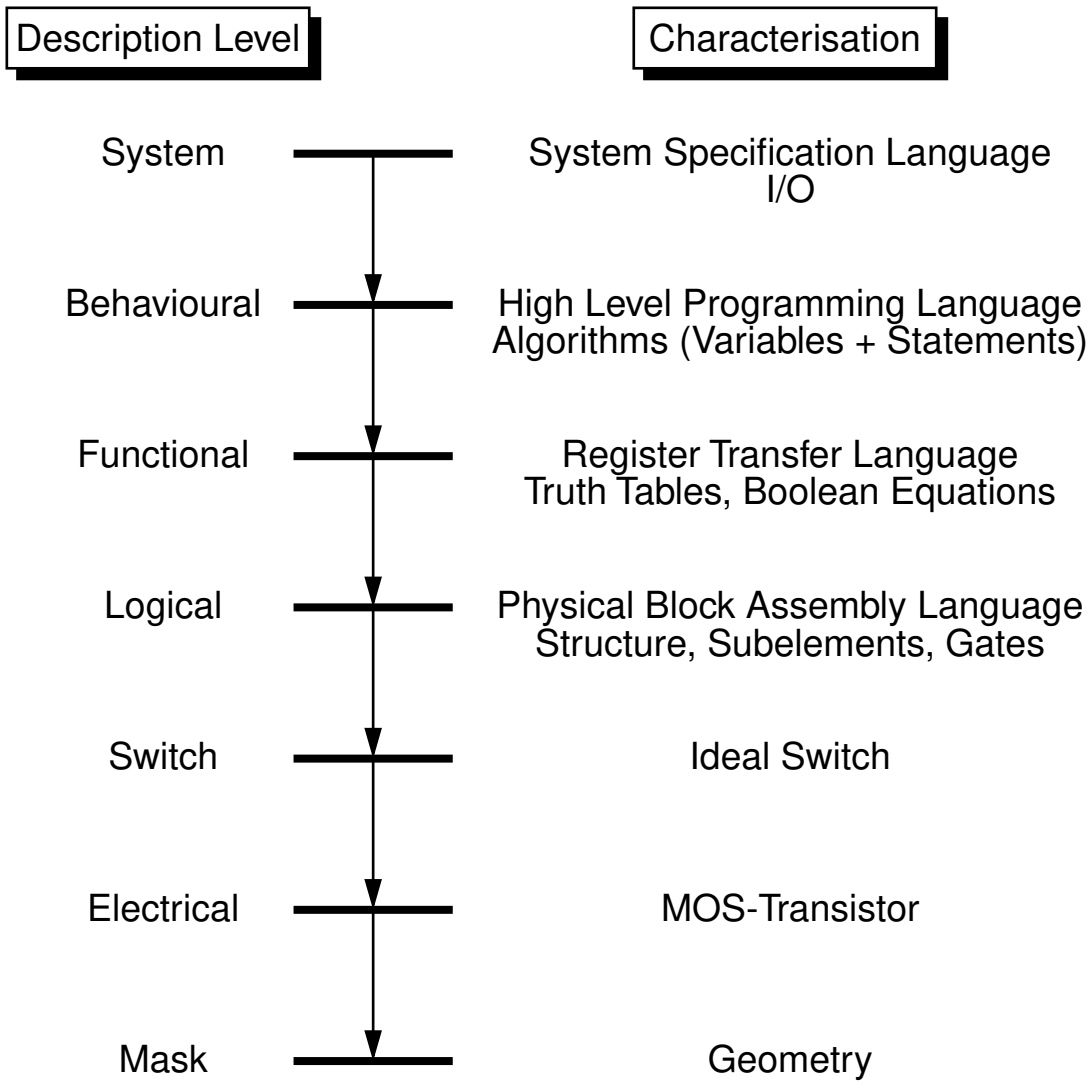

FIGURE 3. The VLSI Design Hierarchy, from [50]

The evolution of hardware design methodologies has been characterised by the successive addition of abstraction levels. This reuse of concepts and especially tools and methods is efficient and has enabled designers to rapidly adapt to implementing increasingly complex systems on an integrated circuit.

Similarly, in software development a large number of methods have been developed to address the requirement for designing systems with ever increasing complexity. There has been a shift from machine code generation to assembly language and to processor independent high level languages with their associated compilers, linkers, debuggers etc. The existence of standard operating systems and application programming interfaces has aided the development of more complex systems. Traditional design methodologies for software have served to capture this process. Despite the lack of formal modelling techniques they have often proved useful in the analysis and communication of system designs. However a significant difference between software and hardware methodologies is the rigorous approach to design in the hardware area. This is motivated by the cost of an error which has always been measurable and high.

As systems become increasingly complex, heterogeneous and distributed, methodologies are required that enable designers to specify, design, implement and test them. For example, it is no longer practical to partition a complex system into hardware and software components early in the design process and have hardware and software teams design and implement their parts independently. An optimal implementation of a system can only be made when all of the functionality and design constraints are known.

It is proposed in this monograph to model the complete behaviour of a system in a further abstraction level. This model is used as the basis for further refinements and for eventual implementation, partitioning and code synthesis. The choice of modelling languages should also consider the fact that exhaustive testing of complex systems is impractical, therefore requiring that multi level simulation and component verification is possible.

2.1 Major Abstraction Levels

In [54] a design methodology for embedded systems is proposed to consist of three major levels: behavioural, structural and physical. These levels can also be seen as a grouping of hardware abstraction levels, shown in figure 3, and high level abstractions for the support of codesign. The three levels are described in the following sections.

2.1.1 Behavioural Level

At the behavioural level it is advantageous to specify a system's functionality with an executable model. This model can be considered to be a system prototype, testing whether it meets all design requirements. An executable model is very useful at all stages of the design process as it is unambiguous and can be used as a reference for the system's functionality. A model may be verified very early in the design to test whether certain constraints are feasible and it is also easier to evaluate the impact of proposed changes. Furthermore it can be used as a starting point for product upgrades and maintenance.

The requirements for a behavioural system model include the ability to model a system at high levels of abstraction without imposing an implementation regime. The model should be easily understood, even for non technical people, enabling the system functionality to be communicated. In addition it should be formal, executable and amenable to analysis.

2.1.2 Structural Level

At the structural level components are defined that interact with one another to implement the required functionality. At this level important decisions are made regarding the number, connectivity and function of system components in order to define the full functionality and constraints of the system as a set of communicating components. The design and test of algorithms may

be considered a part of this level, which may also include the analysis of the effects of the use of restricted word lengths and general algorithm robustness.

2.1.3 Physical Level

The physical level is where implementation decisions are made and implementation scenarios are evaluated. The models used are refined to include information details such as hardware or software implementation technology and number of physical implementation units. For example a particular function may be implemented on a number of dedicated concurrently executing implementation elements or it may be implemented on a shared resource together with other functions. At this level scheduling is introduced to ensure the optimal use of resources while meeting constraints.

This abstraction level relies on traditional methodologies and implementation techniques to provide information regarding the quality of a particular implementation option. This may be in the form of a performance and area estimate for a particular function implemented in hardware or the size of code generated for a software based component etc. This requires the use of many tools specific to particular implementation targets, such as compilers for microcontrollers, DSPs etc. and logic synthesis tools targeting programmable or custom hardware.

2.2 Additional Requirements

The requirements for a codesign methodology discussed above are very general. In this section a number of specific additional requirements are presented. These requirements are a direct consequence of the heterogeneous nature of embedded real time systems where it is desirable to produce implementations that meet functional and non functional constraints.

2.2.1 Multi-Paradigm Support

The nature of many real time systems is such that no single formalism is suitable for modelling all aspects of them. An embedded system design methodology should allow designers the flexibility of choosing the most appropriate set of formalisms for the particular system to be modelled. Also many designers will prefer to apply familiar methods and notations from their working environment, application domain and experience. Allowing systems to be modelled with familiar formalisms reduces the risks, costs and difficulties of introducing new methods. Many existing formalisms also have well known implementation transformations and analysis algorithms and it is desirable that these continue to be applicable in a heterogeneous modelling environment.

2.2.2 Refinement

Refinement is the process of successively inserting more information into a model. A model is defined as being an abstraction of the real system omitting uninteresting details. The goal of a model is to provide the ability to test system characteristics that would be more difficult or even impossible on the real system. This is especially true if the real system does not even exist, in which case the model represents the system to be designed and on which validation and verification can be carried out to determine whether constraints imposed by the environment can be met. In a typical design process models are initially abstract, requiring further refinement before they represent an implementation with the required level of detail. An important practical issue is the ability to simulate models consisting of components having different levels of abstraction. This enables large systems to be simulated where only the components of interest are modelled in detail. A model may be refined in a number of ways to evaluate design options. Eventually, as a result of this design exploration, a model is found which better represents the system.

2.2.3 Time

Embedded systems are further characterised in terms of their interaction with their environment. An embedded system must not only be functionally correct but must also ensure that the appropriate responses are produced at the correct time. Real time requirements prescribe certain time relationships between particular inputs and outputs and models used for hardware / software codesign should support the formalisation of time constraints.

2.2.4 Configuration

Configuration is the process of generating and managing system variants. Figure 4 shows two variants of a system where the first is optimised for speed and the second for size and cost. System configurations handled within a single model are easier to maintain and keep consistent. The ability to easily configure systems must also take into account the heterogeneous nature of embedded systems where a function may be implemented on different targets.

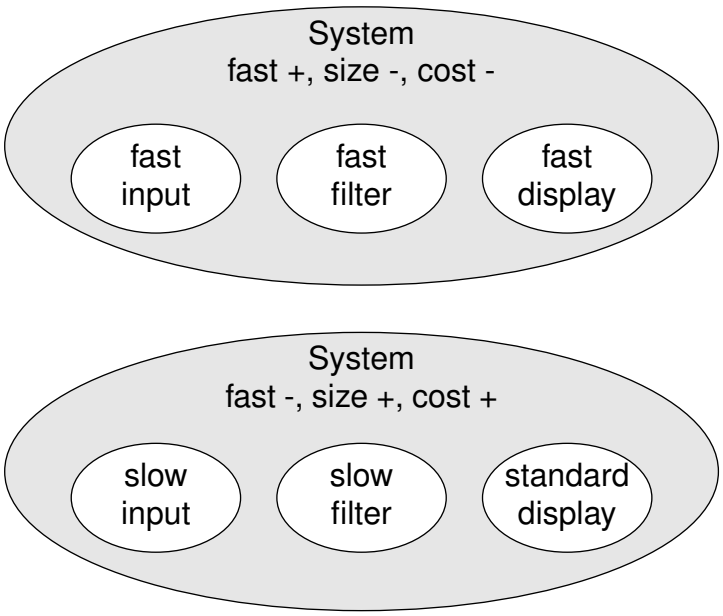

FIGURE 4. Two possible system configurations

2.2.5 Design Exploration

It is necessary to ensure that a system methodology allows an easy exploration of design alternatives once the system's functionality has been defined. It must be possible to allow designers to allocate architectural components and constraints and to cost the resulting system. The cost of a system not only consists of measurable criteria such as delay, power consumption etc. but also, for example, ease of maintenance and extensibility. Figure 5 represents the design exploration process. Many paths exist leading to an implementation that meets the system's functional, non functional and architectural constraints indicated by the grey volume.

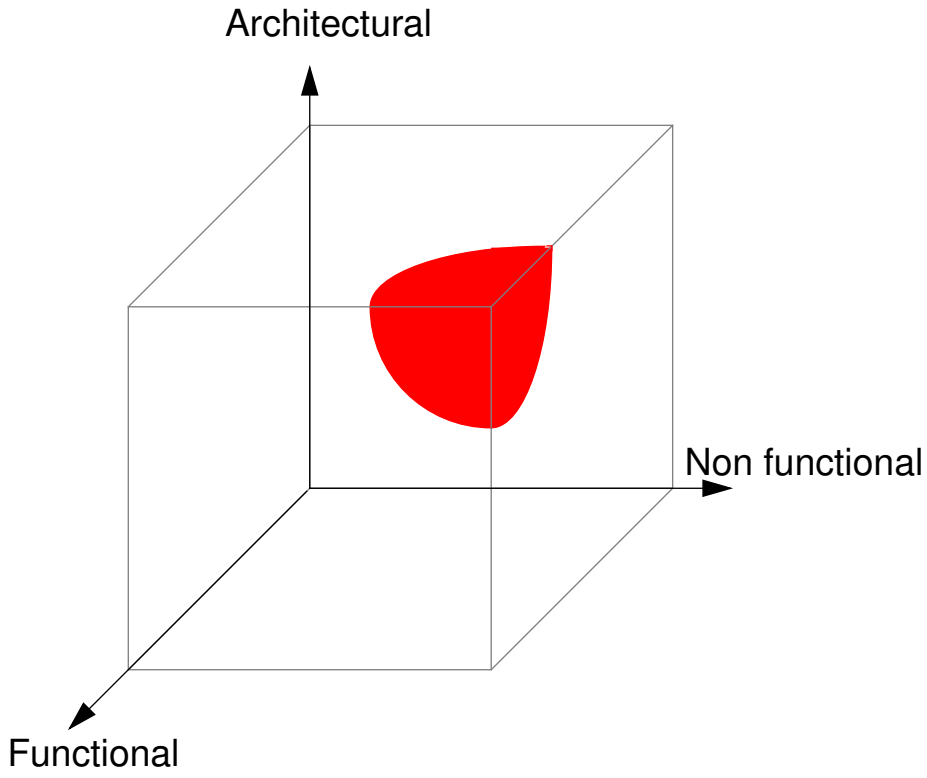

FIGURE 5. Codesign design exploration where the grey volume indicates valid implementations that meet functional, non functional and architectural constraints.

2.2.6 Open Framework for Partitioning and Interface Generation

The partitioning and interface insertion process is central to the implementation of heterogeneous systems. It is advantageous to be able to back annotate a high level system model with partitioning and interface information, as this can constitute a refinement step. High level models are needed for target devices to describe performance and interfacing options (whether they are software or hardware based) and interfacing components, ranging from straight forward unidirectional interfaces to buses with defined transaction protocols. The following series of figures shows how a system model may be partitioned.

Figure 6 shows an abstract system model consisting of two components which communicate through an unidirectional channel.

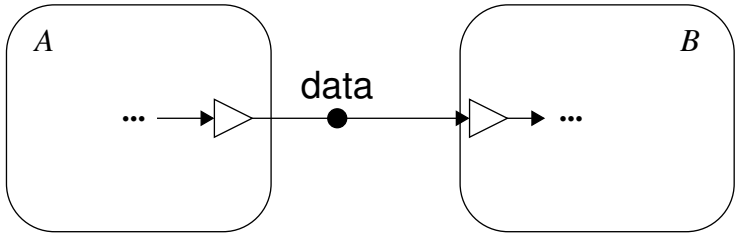

FIGURE 6. Abstract system specification

Figure 7 shows the same system after partitioning. Here the channel is still abstract and needs to be refined. The dashed boxes represent implementation target devices, e.g. microcontroller X and FPGA Y. These devices can only communicate efficiently in a finite number of ways and therefore constrain the type of interface that can be inserted.

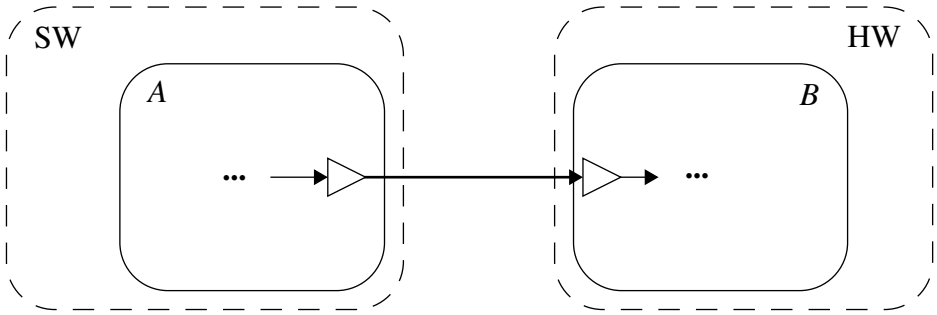

FIGURE 7. System specification after HW/SW partitioning

In figure 8 A' and B' have been refined from A and B with the addition of the greyed interfaces. The channel has been refined to reflect a hand shake protocol.

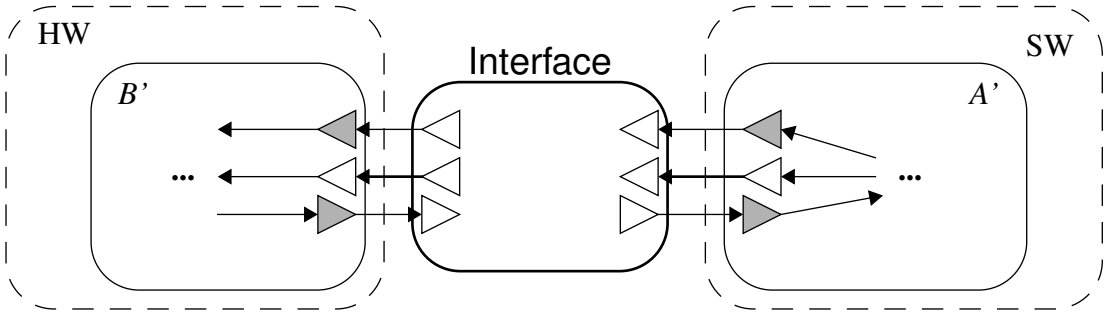

FIGURE 8. Interface insertion

2.3 Summary

The main requirements for an embedded system design methodology can be summarised in the following points:

- abstraction and refinement allow complex systems to be modelled and successively refined towards an implementation;
- a design methodology must be based on formal languages enabling system validation through simulation and verification of critical system characteristics;
- real time requirements prescribe certain time relationships between particular inputs and outputs. Models used for embedded system design should support the formalisation of time constraints;
- an embedded system design methodology should allow designers the flexibility of choosing the most appropriate set of formalisms for the particular system to be modelled;
- configuration support allows system variants to be modelled and generated from a single system model;
- support for an easy exploration of design alternatives once the system's functionality has been defined.

CHAPTER 3 *Existing Approaches*

In this chapter an overview of existing approaches is presented, all of which address some aspects of (embedded) system design. The first section considers models used for describing systems. These models may be used within a design process with the goal of describing embedded systems. The second section discusses methodologies (design processes) for specifying, modelling, validating, verifying and implementing heterogeneous systems.

3.1 Models

A central premise of this monograph is that any design approach should be based on the use of formal models to describe the behaviour of the system before any implementation decisions are made. The use of formal models may allow extensive verification to be made at high levels of abstraction, enabling potential errors to be identified very early in the design cycle and reducing the risk of producing an incorrect implementation. The implementation of the modelled system onto heterogeneous hardware / software components should be made automatically using synthesis techniques.

3.1.1 Models for Reactive Systems

A reactive system is characterised as being to a large extent event driven, i.e. having to continuously react to external and internal stimuli. The behaviour of a reactive system can be summarised as being the set of possible sequences of input and output events, conditions, actions and timing constraints. Examples of reactive systems include communication networks, automobiles, watches, household appliances (washing machines, ovens, dish washers etc.) and man machine interfaces.

Finite State Machines

A finite state machine (FSM) is a network of nodes representing states and directed edges representing present state, next state relationships. Edges are usually annotated with a tuple denoting the events for which a state-transition is sensitive and events that are produced due to the state-transition having occurred. In a classical FSM the system is always in a unique state. Event communication can be considered to be broadcast, synchronous and non-blocking.

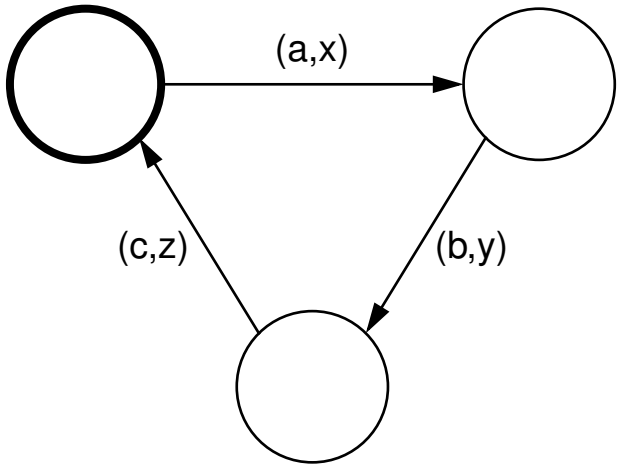

FIGURE 9. A graphical representation of a classical finite state machine. The current state is indicated by a thick border.

FSMs are well suited for modelling control dominated systems. However due to the lack of concurrency and hierarchy support, the size of the system that can be modelled with them is limited. Without explicit support for concurrency a complex system will cause an explosion in the number of states. For example if two concurrent subsystems each having 100 states, were to be described with a single FSM, 100 x 100 = 10'000 states would be required. Models containing such a large number of explicit states are incomprehensible for the designer.

Behaviour Finite State Machines

A behaviour finite state machine (BFSM) [105] is a FSM whose inputs and outputs are partially ordered in time, i.e. augmented with linear inequality timing constraints between inputs and outputs. A system is described as a network of BFSMs where the communication between BFSMs is synchronous, as in classical FSMs. The network behaviour is formally defined by a *simulation procedure*, therefore finding a valid schedule for a network is not equivalent to finding a schedule for an individual BFSM. In fact the behaviour of a network of BFSMs can not be derived and analysed without an explicit simulation of the network.

Codesign Finite State Machines

A codesign finite state machine (CFSM) [34] is similar to a FSM in so far as it transforms a set of inputs into a set of outputs and has internal state. The synchronous event communication of FSMs is replaced with finite non zero (unbounded) reaction times. A CFSM consists of sets of input and output event types and a translation relation. The translation relation is a set of cause-relation pairs, where each cause and reaction is a set of event names and values. Each transition is triggered by input events and emits the output events after an unbounded non-zero time.

An implementation is a refinement of a CFSM model and implies choosing values for the unbounded reaction delays.

StateCharts

StateCharts [62] is a graphical specification language that extends classical FSMs by allowing hierarchical decomposition, timing specifications, concurrency and synchronisation. Hierarchical decomposition is accomplished by clustering states through the use of parallel (AND) or sequential (OR) operations. State-transitions can be defined between any states at any hierarchy level. Timing is specified with linear inequalities on states at any hierarchy level. Concurrency is handled through the parallel (AND) composition of states. Concurrent FSMs can be synchronised through transition edges. Statecharts do not define a data flow specification. StateCharts are described in more detail in 'Example: StateCharts' on page 127.

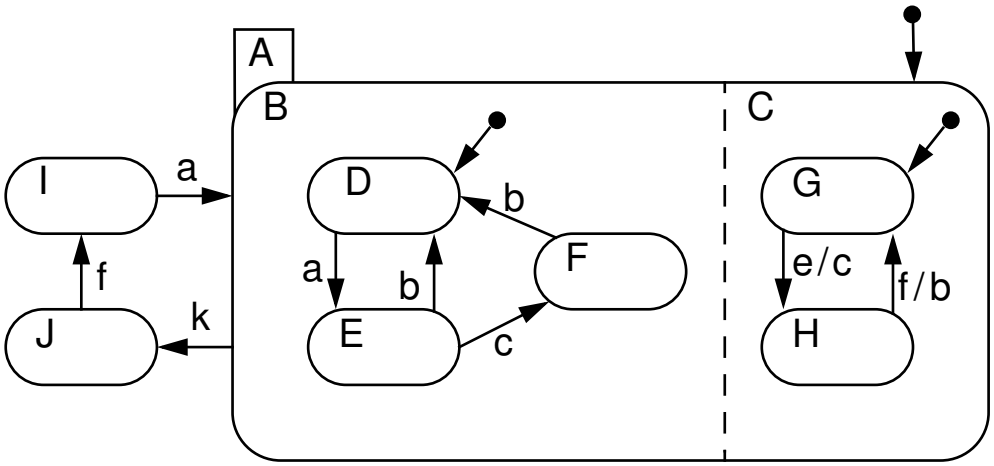

FIGURE 10. A typical StateChart where states are represented as bubbles, state-transitions by connected arcs and initial states by unconnected arcs.

Program State Machines

Program State Machines (PSM) [108] is a model combining hierarchical / concurrent finite state machine model with the programming language paradigm. The programming language used in the PSM model allows data and data transformations to be defined that can be referenced within state transitions. The SpecCharts language supports the PSM model and uses VHDL as the programming language. This is an example of a very pragmatic

approach to modelling embedded systems where no constraints are placed on the programming language portion of the model. Complete formal analysis is therefore traded for flexibility and efficient implementation.

Communicating Interacting Processes

A CIP [51] system consists of several concurrent clusters communicating asynchronously with one another. A cluster is a sequential subsystem composed of a set of synchronously cooperating processes. A process is described as a FSM which interacts with other processes within a cluster through the instantaneous transmission of pulses (software events). Each process transition can carry out internal operations. A process may have local memory, i.e. variables which may be accessed by transition operations and communicated synchronously to other processes within a cluster or asynchronously to other processes between clusters.

An interesting extension to standard FSM is the concept of modes. A process may have a number FSM representations which are controlled by a moderator FSM. All FSM representations share the same set of states but may define different state transitions to implement mode dependent behaviour. Another feature of CIP is the method used to control the flow of pulses within a cluster. Using interaction diagrams it is possible to describe how pulses are propagated between processes. This is in stark contrast to StateCharts where all events are globally broadcast.

Esterel

Esterel [18] is a synchronous language in which time is considered to be a series of instants, between which nothing happens. In each instant some events occur in the environment and a reaction is computed instantly by the modelled system. Thus it is assumed that communication and computation take no time. This assumption is very convenient as it allows a complete system to be modelled as a single FSM (which may be very large) whose behaviour is totally deterministic.

The Esterel language is very simple and includes constructs for hierarchy (procedure calls and module instantiation), pre-emption, concurrency and sequencing. Common to all FSM based models is that data manipulation is not very natural. The synchronous model simplifies specifying in Esterel but also forces timing constraints to be specified elsewhere.

Specification Description Language

SDL is an acronym for Specification Description Language recommended by the CCITT [32] (Comité Consultatif International Telegraphique et Telephonique) for unambiguous specification and description of the behaviour of telecommunications systems. SDL is used to describe the behaviour and structure of real time distributed systems at different levels of abstraction.

A SDL system description consists of a number of blocks communicating via channels. Blocks may be decomposed into further blocks and finally into processes. A process consists of a finite state machine where state-transitions are broken up into component parts such as guard, task, inputs, outputs and decisions. Processes may be dynamically started and terminated. Time is treated as a global variable and may be used to define the duration of an action. A timer element is also provided, which may be controlled and used to implement delays and time-outs. SDL uses the concept of abstract data types to describe data types in an implementation independent way, i.e. data types are described in terms of their properties and not their concrete structure. SDL has two representation forms, textual and graphical, which enable SDL models to be created graphically and communicated textually to other tool environments. All communication in SDL is asynchronous, with blocking read, non blocking write and assumes the presence of infinite buffers.

Synchronised Transitions

The language Synchronised Transitions [104] implements a fine grained parallel model suitable for describing hardware and software components. It describes a computation (transition) as a collection of atomic, guarded assignments without any explicit flow of control, e.g.

$<<c > 0 \rightarrow out, c{:=}true, c - 1>>$ where c and *out* are state variables. Each transition can occur concurrently with other transitions. A set of transitions operating on the same set of data (states) can be considered to be a state machine, thus a model written in synchronised transitions can be thought of as a number of concurrent finite state machines.

3.1.2 Models for Transformational Systems

Transformational systems typically transform an input data stream into an output data stream and can usually be specified with a function describing the input / output relation. While transformational systems can be highly complex, methods exist to decompose a system's behaviour into smaller parts that are coherent and rigorous. Specialised models exist for transformation systems allowing them to be modelled, analysed and efficiently implemented. Transformational systems are often components of larger embedded systems, e.g. mobile telephone, video phones and radar surveillance systems.

Synchronous Data Flow

Synchronous Data Flow (SDF) graphs [84] is a restricted form of the data flow model of computation [41]. Data Flow consists of nodes (*actors*) which represent computations and arcs which represent the data paths between nodes. Nodes can have any number of inputs and outputs. In SDF, for an operation to be executed, a fixed number of data items (*tokens*) need to be present for every input arc and a fixed number of output data are generated for every output arc. SDF graphs constrain the number of outputs connected to a particular input to 1. SDF graphs are widely used in digital signal processing systems because their representation exposes the natural parallelism of the modelled algorithm and imposes minimal constraints upon the execution order. Analysis techniques exist for SDF graphs and efficient implementations can also be generated.The SDF model is unable to model

data dependent behaviour, i.e. selecting the nodes to compute, based on data values. SDF is discussed further in 'Describing a Formalism' on page 121.

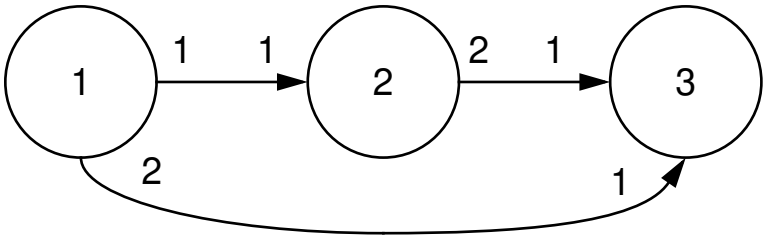

FIGURE 11. A SDF graph with three actors. The numbers inscribed on the arcs represent the number of data items produced or consumed by a node.

3.1.3 Control / Data Flow Models

Control / data flow graphs allow systems to be modelled as networks of operations. They are particularly well suited to systems that require fine grain implementation partitioning. A particular strength of these models is their ability to describe systems that are neither purely control nor data dominant. This makes them suitable for modelling heterogeneous systems.

Boolean Data Flow

Boolean Data Flow [29], [83] overcomes the lack of data dependent operators in SDF models by the inclusion of nodes which act as *switches.* These are able to select an output based on an input token value and *selects*, where an input can be selected. With these two types of nodes, data (tokens) can be routed through an SDF model enabling more complex algorithms to be modelled and efficiently implemented.

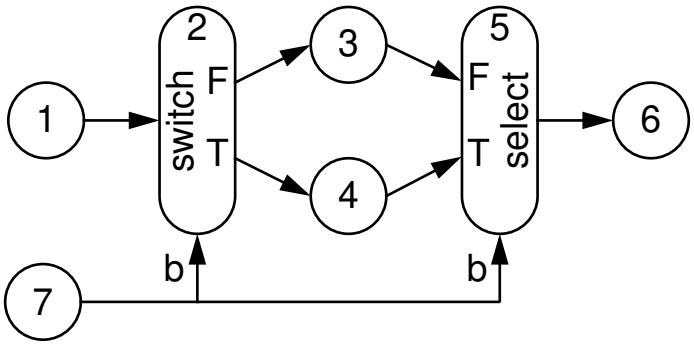

FIGURE 12. A Boolean Data Flow graph modelling an if-then-else. Arc inscriptions are not shown. Note that node 7 produces a Boolean valued token used to configure the switch and select nodes.

Hierarchical Flow Graphs

Hierarchical flow graphs consist of nodes and edges, where nodes denote operations and edges dependencies. Operations can be conditional, computational, wait or loop operations. Conditional operations allow data dependent paths in the graph model. Communication can be blocking (wait) and non blocking (receive) and data dependent loops are also available. A hierarchical flow graph model is used for partitioning, scheduling and synthesis in the Vulcan system [40].

Extended Syntax Graphs

A Syntax graph is a directed acyclic graph describing a sequence of declarations, definitions and statements. Extended syntax graphs augment syntax graphs with a symbol table and local data and control dependencies. COSYMA [46], a design system for co-processor generation, uses extended syntax graphs as an internal representation.

Petri Nets

A Petri net [95] is a bipartite graph consisting of buffers (*places*), operations (*transitions*) and directed edges (*arcs*). Data on places are referred to as *tokens*. Tokens on transition input places can *enable* a transition to *occur* (*fire*) where the enabling tokens are consumed and new tokens are produced for all output places. In traditional Petri net formalisms the kind of data modelled by the net is determined by the token types. In elementary nets there is only one type with only one value, therefore the state of the modelled system is completely defined by the net *marking* (number of tokens per place). In high level Petri nets [55], [69] tokens can have arbitrary types such as boolean, integer, real, arrays or records etc. In addition to the extended token types, high level Petri nets have net inscriptions to manipulate the values of tokens. The state of a system is the token marking with the value of each token in that marking. A high level Petri net can be used to model control flow as well as data flow. The occurrence rule is extended to allow a transition to be enabled depending on the input token data values. When a transition occurs the output token values are calculated using only the input token data values. Petri nets have been used to model complex systems for hardware / software codesign [42]. The generality and flexibility of Petri nets and their suitability as a kernel language for an embedded system design environment is discussed further in 'Kernel Language' on page 63.

3.1.4 General Purpose Languages

Many general purpose languages have been used to describe embedded systems. These languages were developed to implement particular types of systems and therefore may lack certain features required for a modelling language. When using a general purpose language care must necessarily be taken to avoid constructs that imply a particular implementation. In particular embedded real time systems require support for concurrency, communication and time and often these languages have been extended with necessary features to enable such systems to be adequately modelled. Such languages are familiar to many engineers who unfortunately often under estimate the difficulty in mastering the particular set of language extensions.

ANSI C

The internal control / data flow model of ANSI C has been used as a basis for modelling embedded systems. ANSI C has the benefit that it is widely used and known, in addition to being low level enough to describe implementation aspects in sufficient detail.

The language C^x [46] is ANSI C extended with minimum / maximum delays, tasks and task communication. A C^x program is translated into an extended syntax graph from which a fine grain model is created which is suitable for partitioning and synthesis. C^x is used in the COSYMA system for system specification. COSYMA has the goal of implementing as much of the system as possible in software and only maps functionality into hardware, in the form of coprocessors, when timing constraints cannot be met.

Hardware-C [40] is another example of an ANSI C based language. It is a synthesis oriented hardware description language that enables hardware to be described with declarative and procedural semantics, including the ability to formulate constraints on the hardware implementation. It adds two further design abstractions to those available in ANSI C, namely *block* and *process*. A block contains an interconnection of logic and instances of other blocks and processes. A process consists of a hierarchy of procedures and functions and executes repeatedly, restarting itself upon completion, thus implementing a coarse grain parallelism at the functional level. Communication in Hardware-C is via parameter passing and message passing with synchronous semantics.

Occam

The Occam language [66] is based on Communicating Sequential Processes (CSP) [63]. Occam, like CSP, defines processes and communication in a rigorous way enabling semantic preserving program transformations. Hardware / Software partitioning can also be regarded as an application of program transformations. This has been exploited in [10], enabling Occam models to be partitioned onto heterogeneous hardware / software components with guaranteed behaviour.

ADA

In [44] the programming language ADA [2] is used for the modelling of embedded systems. A system can be described as a network of ADA tasks that communicate through a *rendezvous* mechanism. A task can choose to wait for a rendezvous or to continue processing based on the value of a guard. As ADA is a very large language, the implementation described in [44] uses a subset that permits hardware synthesis. ADA has powerful encapsulation mechanisms which enable generic models to be refined towards an implementation.

VHDL

The VHSIC Hardware Description Language (VHDL) [65] was developed by the American Department of Defence and standardised in 1987. The language is intended to assist in the development, documentation and exchange of hardware designs. Since its development, VHDL has been extensively used as a description language for a wide variety of graphical capture tools, for simulation, synthesis and debugging.

The primary abstraction in VHDL is the design unit represented as an *entity* describing interfaces and one or more associated *architectures*. A design unit represents a part of a larger design that performs a specific function with well defined inputs and outputs. The functionality of each unit is described using programming statements, data flow, control structures or any combination of these. VHDL supports a two level behavioural hierarchy where a system is specified into a set of concurrent processes at the top level. The second level consists of sequential decomposition of these processes into procedures.

Communication in VHDL is by *signals* which can be thought of as representing wires. Signals are also used for synchronisation in two ways; the first relies on a process' sensitivity list, which ensures that the process will begin to execute when an event occurs on any signal named in the sensitivity list. The second way employs the wait statement, which suspends a process

until it detects either the occurrence of an event on one of the specified signals or the presence of a specified condition.

The specification of timing in VHDL is limited to the specification of functional timing. Using the *after* clause, as in *data <= 4 after 20 ns*, the point in time is specified at which the value of the signal data is updated. Time-outs can also be specified, such as *wait on <<aSignal>> for 100 ns*, and can be used to specify the maximum amount of time to wait for the occurrence of a signal.

VHDL is a poor modelling language as the signal concept is too inflexible, being very close to an eventual hardware implementation. It is not ideal as an implementation language as a number of language constructs cannot be implemented and also because no standard exists, defining a subset that can be synthesised.

3.1.5 Remarks on Models for Embedded System Design

Simulation remains the most common way of validating a formal system model. The application of formal verification techniques on real time system models is still quite new despite being highly desirable, due to the safety critical nature of these systems. Fortunately this is changing with the emergence of formal models for which time and functional analysis can be carried out. Consequently it is becoming possible to consider systems with regard to their (often critical) timing behaviour [5], [48] in addition to properties that can be verified in a time independent way, such as safety and deadlock freeness [15].

The benefits of using a standardised modelling language is well illustrated by SDL. SDL is a CCITT standard and its use is often mandatory in the telecommunications industry. It has enabled unambiguous specifications of telecommunication standards to be propagated and to some extent has made the current boom in the telecommunications industry possible. Embedded system designers have tended to use only one modelling language to design a system. The large number of modelling languages available is an indication that the concept of one general modelling language is unrealistic.

It is perhaps premature to propose a set of modelling languages for embedded system design. However the standardisation of modelling languages would encourage tool developers to support these languages to the benefit of all users. Fortunately interest now exists in not only developing new modelling languages, but also in how the use of different modelling languages can be best combined within the same system model.

3.2 Methodologies

A design methodology defines the process of designing a system, typically from conception to implementation. It is usually derived from heuristics obtained over many designs and therefore provides a set of well tested design steps that, when followed, ensures that critical analysis and design aspects are not neglected. A design methodology may require the use of different models at various stages to highlight particular system characteristics that need to be addressed. A challenge to defining design methodologies is the requirement that it be free from semantic discontinuities which force the designer to manually remodel the system in yet another formalism. Design methodologies are necessarily application specific, for example real time behaviour is critical for embedded system design but may not even be addressed in methodologies for designing commercial software applications. However methodologies for different application areas often have many design steps in common.

In this section various system design methodologies are discussed. In the first part some heterogeneous system design methodologies are presented. This is followed by two representatives of traditional software oriented methodologies. Finally two object oriented software methodologies are presented, one of which is tailored for embedded system design.

3.2.1 Heterogeneous System Design Methodologies

Design methodologies for heterogeneous system design are mostly based on methodologies for ASIC design, with emphasis on efficient implementation. At the system level such methodologies are not well defined, often simply assuming that a formal system specification exists. This is illustrated in figure 13 where a typical codesign methodology is presented which emphasizes heterogeneous implementation generation. Two examples implementing this design methodology are 'COSYMA' and 'Olympus' discussed later in this chapter.

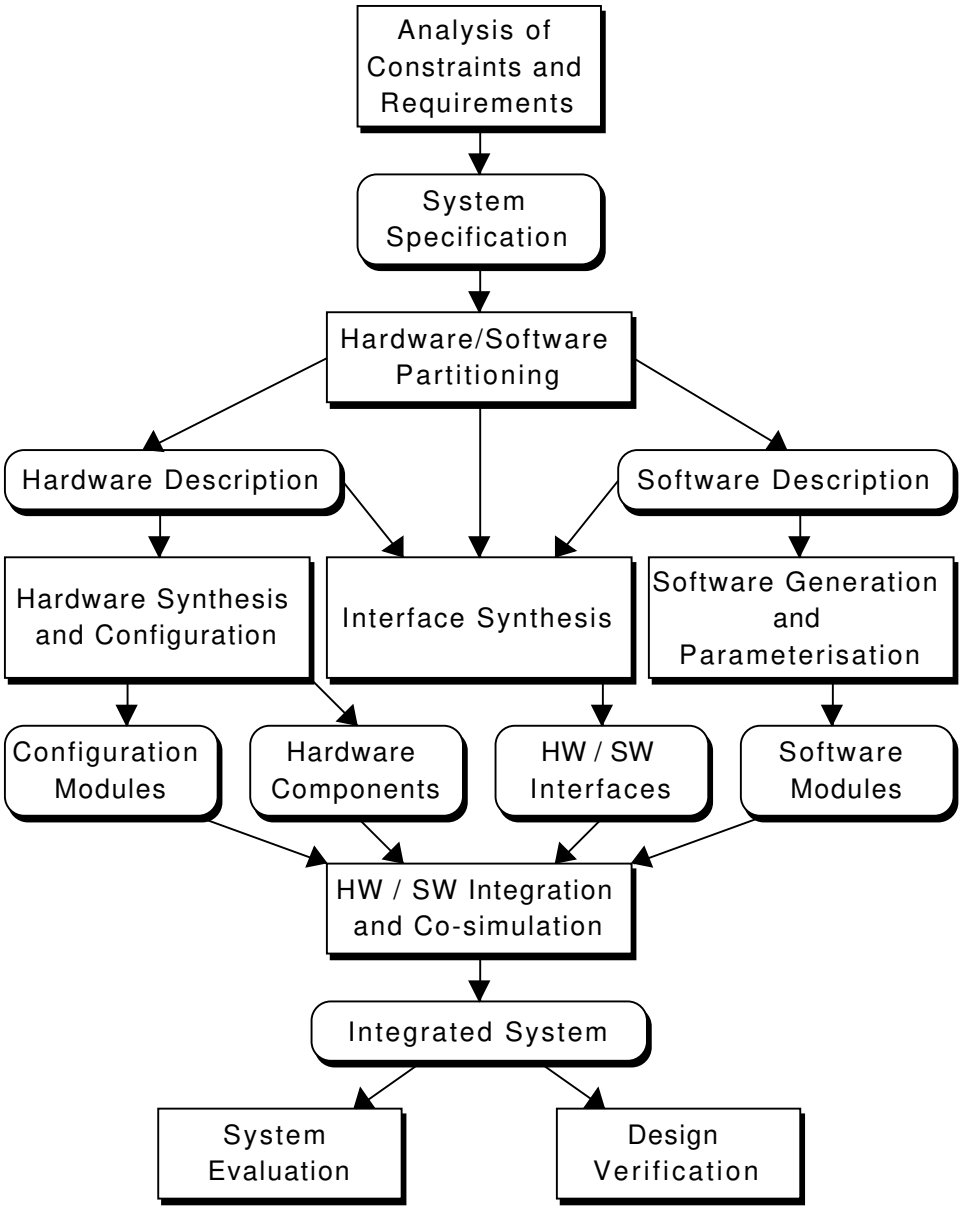

FIGURE 13. The traditional hardware / software codesign methodology

The 'Ptolemy' and 'MOOSE' environments represent a new approach to hardware / software codesign. In this approach greater emphasis is placed on system specification and modelling. This approach attempts to define the system in a formal way very early in the design process where it can be validated and perhaps even verified. Figure 14 taken from [27] shows a model

based methodology where system modelling and refinement are emphasised.

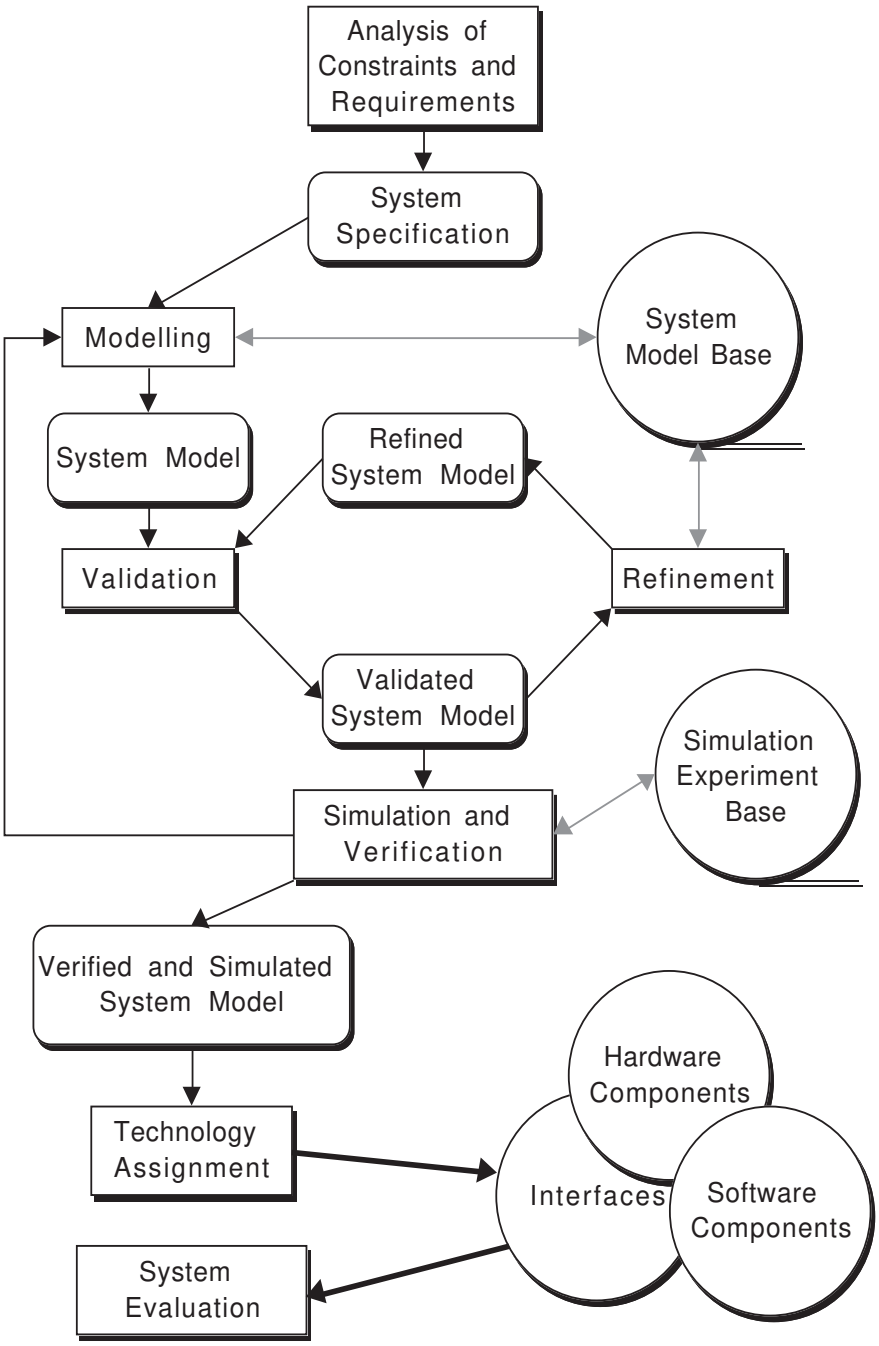

FIGURE 14. A model based codesign methodology

COSYMA

The approach taken in the COSYMA [16] project assumes that a specification exists written in the C^x language, see 'ANSI C' on page 35. A C^x description is translated into an internal extended syntax graph representation which can be simulated and profiled. The environment supports automatic partitioning based on a simulated annealing algorithm, where it is initially assumed that the complete system is implemented in software. The partitioning process migrates system blocks to hardware, attempting to find an implementation that meets the desired timing constraints. After partitioning the hardware parts are translated into the Hardware-C language and implemented via the Olympus high level synthesis system.

Olympus

In the Olympus system [40] systems are specified with the Hardware-C language. The Hardware-C model is translated into an internal representation which can then be partitioned into hardware and software. In contrast to COSYMA where initially a software solution is investigated and functionality moved to hardware, in Olympus practically the whole system is assumed to exist as hardware with only functionality characterised by non deterministic delays (data dependent loops and synchronisation primitives) residing in software. An iterative procedure moves processes between partitions with the goal of reducing communication while satisfying timing and other constraints.

Ptolemy

The Ptolemy [72] design environment is a framework that allows hardware and software components to be integrated from specification through the synthesis, simulation and evaluation phases. In Ptolemy objects described at different levels of abstraction and using different semantic models can communicate with each other using a hierarchy based mechanism where:

- an abstraction level with a specific semantic model is called a *domain*, such as data flow or discrete event;

- atomic objects called *stars* are primitives of the domain, i.e. data flow operators, logic gates etc. These objects are used in simulation mode, reacting to or producing events and in synthesis mode, producing target specific code;
- *galaxies* are collections of stars and other galaxies. Instantiated galaxies may possibly belong to different domains than that of the encompassing domain.

Each domain includes a scheduler which decides in which order the stars are to be executed. Whenever a galaxy instantiates another galaxy belonging to a different domain, a wormhole mechanism is created facilitating the communication between the two schedulers with appropriate data type conversion.

MOOSE

MOOSE [60] (Model-based Object Oriented Systems Engineering) is a graphical / textual method in which a system is described and investigated using abstract, executable and implementation neutral *(uncommitted)* models. Once the behaviour of the uncommitted model is satisfactory its components are gradually *committed* to hardware or software on the basis of *systems engineering* decisions. These are based on non functional requirements and on the results of executing and analysing the uncommitted model.

In MOOSE systems are specified as asynchronous communicating processes. Synchronous parts of a model are described using class implementation diagrams which, similarly to data flow diagrams, have operations as processes and state information as data stores. Communicating processes are specified as a hierarchy of Object Interaction Diagrams (OIDs). The top level OID is progressively decomposed to the primitive level, where any further decomposition would result in technology specific decisions (depending on the hardware and software implementation). MOOSE distinguishes between two kinds of communication: continuous data (non destructive read) and events (destructive read). MOOSE models are executable and object oriented, facilitating reuse.

3.2.2 General Purpose Methodologies

In the following section two 'traditional' methodologies are presented. Structured Analysis is often used to design traditional electronic data processing (EDP) applications which are to be implemented in traditional imperative programming languages. Structured Development for Real-Time Systems has a number of concepts in common but is extended to cover those aspects important for real time systems, namely concurrency and communication. Both methodologies are a combination of formal and informal models used to describe a system. Therefore, apart from simple consistency checks, no model verification can be carried out.

Structured Analysis

Structured Analysis (SA) [113] proposes a step wise decomposition of data flow diagrams (DFD) so that each process in a *parent* DFD is broken down into a number of *child* DFDs. SA proposes that a system be modelled in two main steps. Firstly a *context diagram* is developed which shows how the system is connected to its environment. The user defines this context diagram in terms of sources and sinks of the environment, processes, data flows and files. In the second step the user partitions and refines the system with each process of a DFD being described in increasing detail until atomic processes emerge. The user then describes the algorithmic structure in an informal mini spec. A data dictionary is created containing the data structure and also enforces consistent naming of processes, data flows etc.

SA has a graphical representation where the sources and sinks of the environment are shown as boxes, processes as circles, data flows as arcs and data stores as bars. The mini specs are written as pseudo code and the data is described in a Backus-Naur Form notation. Figure 15 taken from [85] shows a data flow diagram for a display controller. Here the three processes are abstract and are subject to further decomposition.

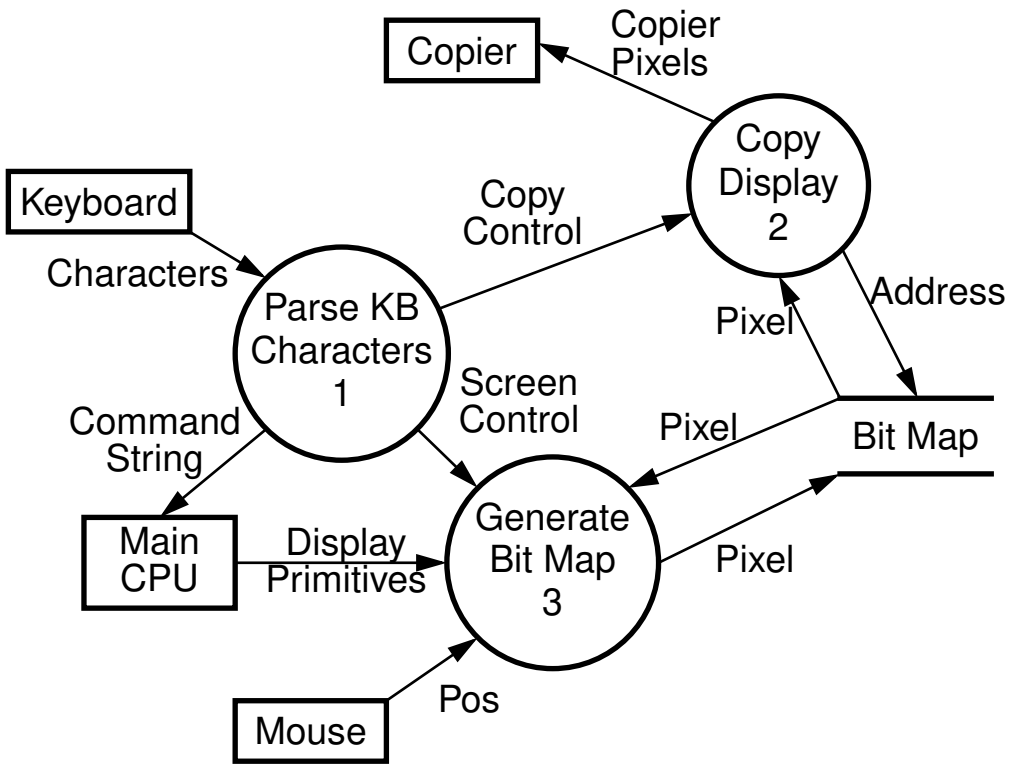

FIGURE 15. SA Data Flow Diagram for a display controller with data flow shown as arcs between processes (circles), interfaces (boxes) and data stores (double bars).

Structured Development for Real-Time Systems

Structured development for real time systems [111] is an answer to the inadequacies of both a purely data flow oriented approach to real time systems analysis and a purely program design oriented approach to real time systems design. The design process is considered to consist of three main activities:

1. the construction of an *essential model* which describes the required system behaviour;
2. the production of an *implementation model* where the implementation technology that embodies the required behaviour is described;
3. *system build* where the implementation model is transformed into hardware and software.

The main tools for constructing models are data flow graphs extended with the addition of events (dashed arcs), double headed arcs that represent continuous data flow (i.e. there is always a data value present) and control transformations (dashed circles) which represent finite state machines reacting to and producing events controlling transformations. Figure 16 shows part of a climate control system described using the extended data flow notation.

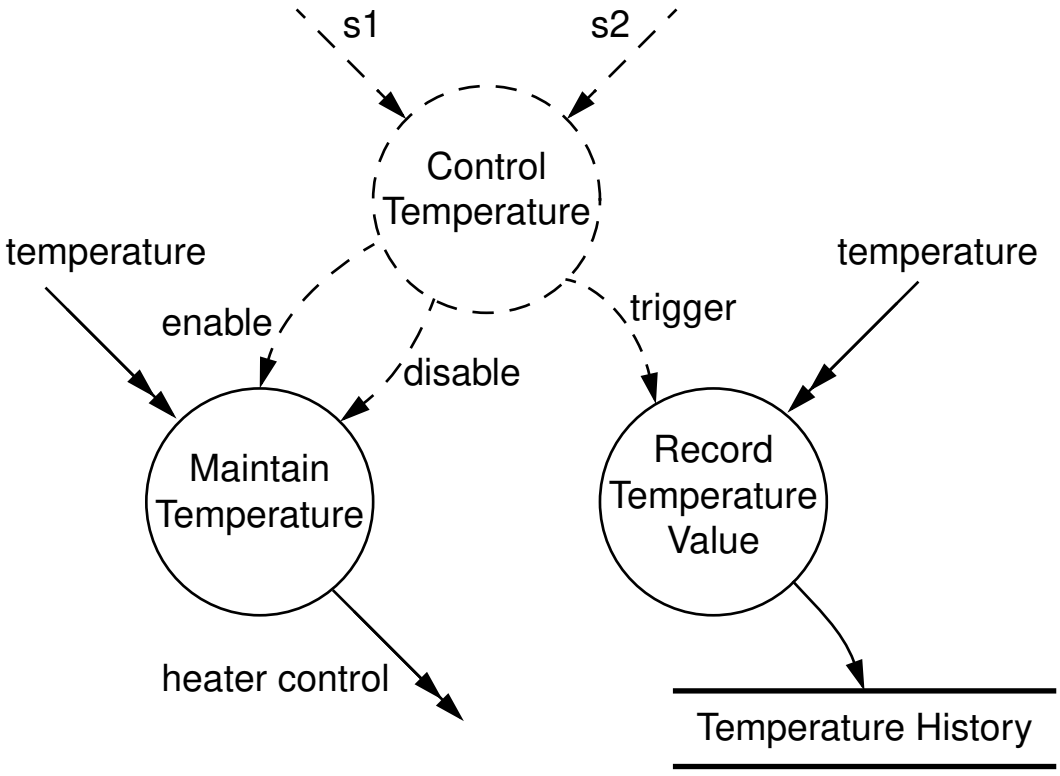

FIGURE 16. Part of a structured development system model.

This approach is interesting as it combines event based and transformational modelling in a single representation. System models are semi formal, as executable models are not created and only basic consistency checks are possible. A proposal to formalise this approach with an executable model based on Petri nets is described in [97].

3.2.3 General Purpose Object Oriented Methodologies

The popularity of the object oriented programming paradigm has resulted in a number of methodologies being developed with the aim of creating object based systems. The object oriented approach is a continuation of the work done in systems engineering and maintains proven concepts of information hiding, coupling and cohesion. Here two methodologies are presented - the Booch Method, designed for the creation of standard object oriented applications and Real-Time Object-Oriented Modelling, for the design of object oriented real time systems.

The Booch Method

The Booch Method [22], [112] is a development method used to develop and communicate the analysis and design of a *software* system. It is object oriented in that objects are used to model an abstraction of the real world. A system model is built in stages from which executable releases are produced, which are true subsets of the final implementation. The model is continually refined towards an implementation and avoids the semantic gaps associated with other traditional methods.

The Booch method consists of three steps:

1. *requirements analysis*, in which the basic system functionality is defined;
2. *domain analysis*, which provides the key logical structure of the system;
3. *system design*, which provides the physical structure of the system, maps its logical structure and from which executable releases are produced.

Requirements analysis is the process of determining the key functions that the system is to perform. It defines the scope of the domain that the system will support and documents the key practices and policies of the company that the system must support. Requirements analysis is not formalised as it is seen as a teamwork of knowledgeable experts, managers and customers. A useful tool for this is *Use Case Analysis* [68] - a way of describing typical scenarios usually instigated by some external event.

Domain analysis is the process of defining a quality object oriented model of the problem domain. This process serves to obtain the detailed knowledge required to create a system capable of fulfilling the functional requirements. It involves the identification of all major objects in the problem domain including all data and major operations needed to carry out the system's function. A successful domain analysis produces *class diagrams* which identify the key classes or types of the system, *class specifications* which contain the semantic definitions, attributes, relationships and key operations, *object scenario diagrams* illustrating how objects interact to perform the system functionality and a *data dictionary*, used as a repository of all the above information.

System design is the process of determining a (cost) effective and efficient implementation to carry out the functions and store the data defined in the domain analysis. The system design process maps the ideal or logical results of the domain analysis to a structure that will allow these objects and classes to be coded and executed. It is an iterative and incremental process producing amended implementation class specifications.

The Booch method is a set of guidelines and techniques for creating object oriented software systems. It is not formal, as only at the system design stage are executable models (implementations) produced. Nevertheless it is a valuable guide in producing any type of object oriented system.

Real-Time Object-Oriented Modelling

Real-Time Object-Oriented Modelling (ROOM) [100] is an approach for the design of real time systems. In ROOM a system is described using three distinct types of objects: *actors* are used to describe concurrent behaviour and can have a hierarchical structure while *protocols* are objects used in a message passing environment and *data*. Actors exchange messages consisting of protocol objects and data modelled in the implementation language.

The behaviour of actors is defined with ROOMCharts - a StateCharts like language. ROOMCharts are hierarchical finite state machines where messages replace the events of StateCharts and a state-transition is triggered by

the arrival of a message from a predetermined port and the execution of a guard function. As a result of a state-transition occurring, messages can be sent, and the extended state (not explicitly defined in the ROOMChart) of an actor modified through the execution of functions.

Associated with ROOM is a suggested design methodology:

1. define the system and its boundary with the environment. This boundary may change during the course of developing a system if, for example, more intelligence is delegated to the input sensors and output actuators;
2. formulate *scenarios* [99], equivalent to *Use Case Analysis* [68], that the system must implement. Initial scenarios focus on system interaction leading to more detailed scenarios, including error scenarios. Scenarios are the main method of describing the required system functionality;
3. from the scenarios derive the key protocols that cross the system boundary;
4. capture an early view of the system structure. This structure can be derived by analysing scenarios and identifying *candidates* for concurrent execution between which messages flow, implementing scenarios;
5. define an initial set of actors implied by the concurrent physical components described in the scenarios;
6. define the associated protocol and data classes;
7. key actors are developed and prototyped, the system is validated at various stages during the development of the component class definitions;
8. the consequences from a distributed implementation are analysed, resulting in further refinement iterations.

This approach is model based and consists of continual refinement of models representing the system and its environment. This refinement uses previous models, requirements and scenarios to produce new or refined models.

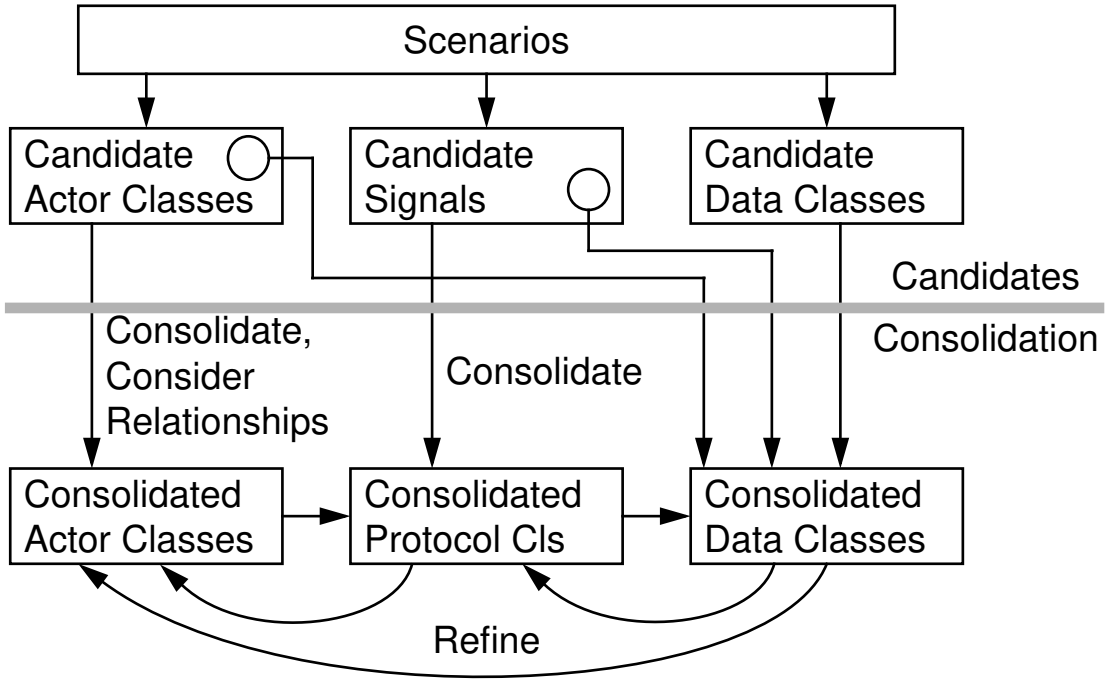

FIGURE 17. ROOM element identification and consolidation process

ROOM is well suited to designing software based, control dominated, real time systems. The asynchronous nature of communication between actors makes it well suited for modelling distributed systems. ROOM is less effective in supporting transformational systems and it has no facilities for modelling time. Time dependent specifications can not be formally captured. To ensure that a system exhibits correct time related behaviour it is recommended that a model be run on the target implementation hardware - a less than satisfactory solution for our purposes.

CHAPTER 4 *Proposed Approach*

Important requirements for an embedded system design methodology include the support for multiple formalisms, refinement, configuration and design exploration. A number of approaches discussed in chapter 3 partially address these requirements, however refinement and configuration require direct language or tool support. Multiple formalism support can be achieved using various tools supporting different formalisms, which communicate via a simulation backplane or by using a general and formal language in which formalisms are implemented. Approaches using traditional programming languages are formal but do not inherently support analysis of models described with multiple formalisms. Petri nets are ideally suited to this task as they are formal and, in the case of high level Petri nets, expressive enough to describe other formalisms. Furthermore their graphical representation enables concurrency to be displayed in an intuitive way.

In this chapter a flexible methodology is proposed based on well known object oriented concepts. The associated modelling (kernel) language is formal and amenable to analysis for system properties. It is based on high level time Petri nets augmented with object oriented concepts and is expressive enough to not only describe embedded systems but also other formalisms, which can be combined to model these systems. The proposed approach can be summarised as follows:

- formal modelling language enables unambiguous specification and supports a high level of analysis;

- homogeneous language (does not rely on a external implementation language) is general enough to support other more specialised formalisms, such as finite state machines and data flow graphs;
- modelling of real time requirements is supported where certain time relationships between particular inputs and outputs are prescribed;
- facilities for encapsulating behaviour and state within components (described with classes) are provided by the kernel language;
- classes define how inheritance properties are propagated and provide mechanisms for refinement, abstraction, configuration, reuse and sharing;
- refinement is supported by substituting less refined components with components containing an increased amount of detail.

4.1 Object Orientation

Traditional function oriented approaches place emphasis on decomposing a system's functionality. While this is well suited to many systems the result can be fragile and changes often require major restructuring [99]. The object oriented approach emphasises the identification of objects from the application domain, resulting in a more robust model, that is based on the application structure rather than the functional requirements of the system. The following essential features are well supported by the object oriented approach:

- abstraction consists of accenting the essential aspects of a system. Properties that are irrelevant at a particular stage in a design are ignored. Emphasizing what an object represents and ignoring how it can be implemented is an example of an abstraction used during system analysis. Proper use of abstraction allows objects to be reused, especially when an implementation neutral language is used to describe them. Abstraction is extremely potent when coupled with inheritance and polymorphism;

- inheritance allows objects to reuse data and behaviour defined by *parent* objects. An object only needs to be differentiated from its parents by either modifying or extending their behaviour. Polymorphism allows functions to be selected and executed which are defined by the parameter types, i.e. different objects may implement a function in different ways. Inheritance and polymorphism allow increasingly refined (less abstract) objects to be created and managed where changes are propagated in a controlled way;
- encapsulation consists of separating the public *interface* of an object from its internal representation. It prevents a system from becoming inter-dependent where small changes may have large repercussions. This allows the implementation of an object to be changed without affecting the functioning of the system. This is particularly useful for design exploration where different implementation strategies are considered.

Object oriented concepts facilitate the reuse of common structures. Behaviour in the form of code is reused through inheritance. This is a consequence of defining behaviour at the appropriate level in the inheritance hierarchy where it can be reused by all inherited objects. The object oriented approach also promotes good design with the prospect of objects being able to be reused in other models.

It is not a prerequisite that a system described in an object oriented methodology is implemented in an object oriented language and reasons may exist why it should not, such as efficiency or a restricted choice of languages.

4.2 Modelling Language

An embedded system design methodology should allow designers the flexibility of choosing the most appropriate set of formalisms for the particular system to be modelled. One approach for supporting multiple formalisms is with a simulation backplane, where individual tools supporting different formalisms communicate during simulation. This has the advantage that

only the communication with the simulation backplane is implemented and not the individual tools themselves. However different tools support various methods of refinement and have diverse user interfaces. Alternatively an environment can be defined where different formalisms are described in an implementation language and communicate with each other through defined interfaces. This approach has the disadvantage that $n^2 - n$ interfaces are required to support n formalisms and that no high level basis exists for these interfaces which allows models described with multiple formalisms to be analysed.

The approach taken in this monograph is based on a formal *object oriented kernel language* and concepts from *graph grammars*. Graph grammar productions are defined which specify the editing rules (syntax) of a formalism. The semantics are modelled using kernel language components to represent the elements of a formalism. Multiple formalisms communicate with one another with semantics defined by the kernel language. An advantage of this approach is that tools need only consider the kernel language rather than each formalism individually, therefore only one simulator, code generator and refinement mechanism etc. is needed. It is however still possible to capitalise on formalism specific tools and methods, for example for efficient code generation.

Figure 18 is an example of high level time Petri nets which are used as the kernel language. This class of Petri nets is an extension of standard place-transition Petri nets where tokens carry data and transitions have associated functions and time constraints. Figure 18 shows a model of system containing four transitions (squares) representing a process fork, concurrency and synchronisation. The transitions are inscribed with tuples *(1,2)* representing time constraints, guards functions ($g_2(in)$) that define additional enabling conditions and functions ($f_4(in_1, in_2)$) to determine the type and values of tokens produced as a result of a transition occurrence. This class of Petri nets is formally defined in 'High Level Time Petri Nets' on page 82.

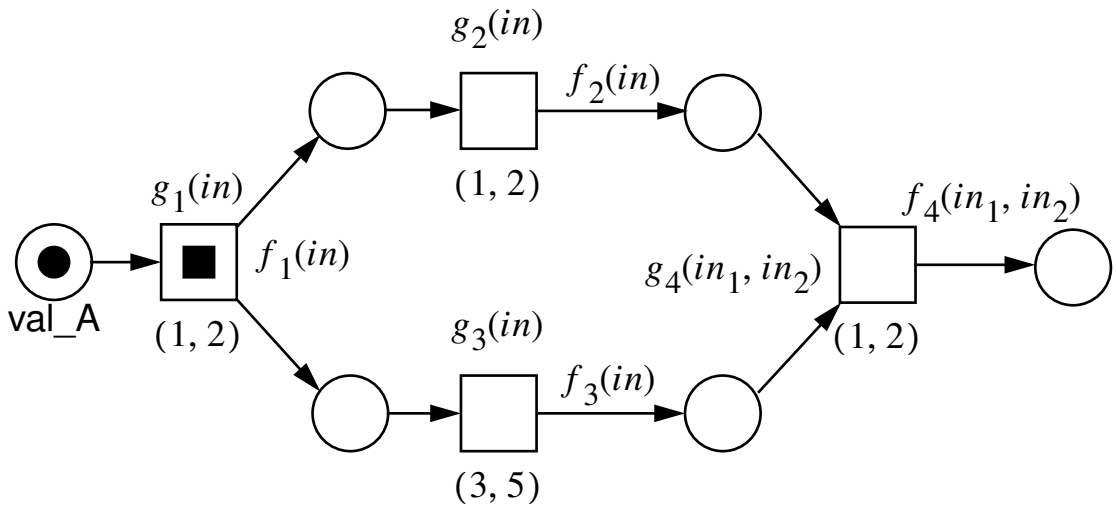

FIGURE 18. A high level time Petri net where tokens represent data and transitions contain timing intervals and functions

The use of high level time Petri nets as a kernel language has several advantages:

- simple formal definition, reducing the possibility of ambiguous interpretation of system models;
- intuitive graphical representation;
- operational semantics that support simulation and execution;
- analysis techniques that prove a number of interesting structural properties;
- ability to be analysed for temporal and functional properties;
- successful use in many different application areas.

4.2.1 Components

In the object oriented approach systems are modelled as communicating components or objects. Components have behaviour which is modelled with the kernel language or another formalism described with the kernel language. Components communicate by sending and receiving objects to and from one another. Components are hierarchically structured, i.e. contain other components, which leaves scope during the creation process to decide which component will be inserted. This mechanism allows design exploration and configuration to be supported in an elegant way.

Unlike other approaches where only special communication objects may be sent, communicating standard objects allows for introduced interfaces to wrap protocols around objects if necessary.

In general it is not advisable to have objects that can only be used in special contexts. A well designed methodology and language does not require that these distinctions be made and requires users to only understand *one* general concept. This is a major motivation for the approach taken and has resulted in a system that is both powerful and easy to understand.

Figure 19 shows a compound component that implements the function $f(a, b) = out = 2a + b^2$ containing three embedded components using the object oriented high level time Petri net kernel language. Components have defined interfaces over which objects are communicated. Here it is possible to imagine a partitioning that places the three embedded components on different implementation target devices and different configurations with implementation specific components that represent this partitioning.

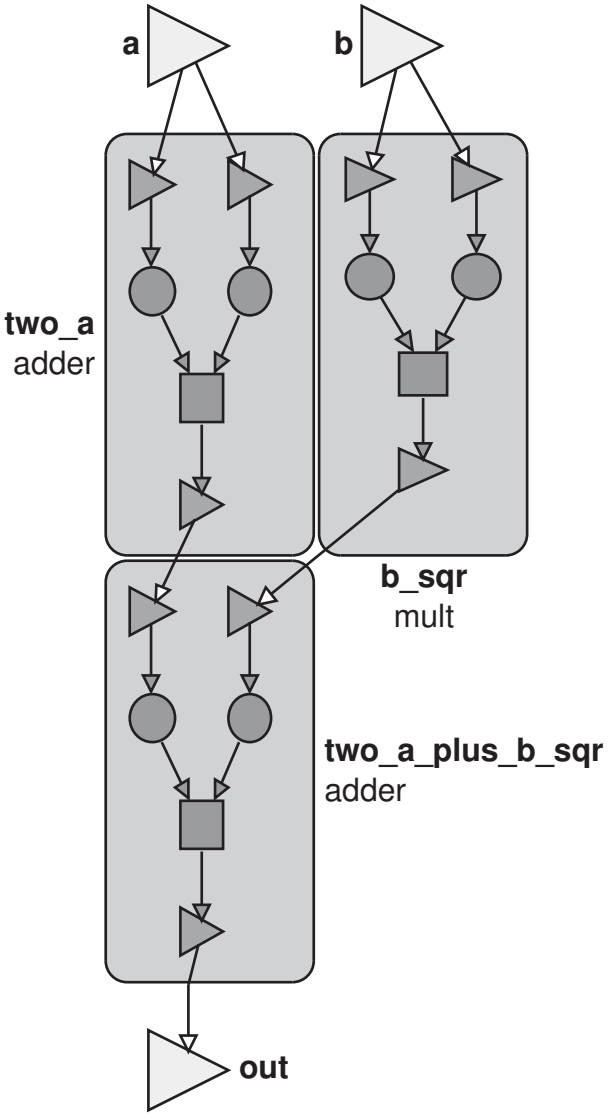

FIGURE 19. Graphical representation of a component containing embedded components. Interfaces are represented as triangle shaped elements (see "Object Oriented Time Petri Nets" on page 100.)

Figure 20 shows a component with an equivalent functionality as that in figure 19. It can be seen that the synchronous data flow (SDF) representation (see page 31) is better suited to describe the required functionality. The use of an appropriate formalism can significantly simplify the representation of a component. The round elements are SDF nodes representing an operation that can be carried out when sufficient input data is present. Graph

grammar productions specify the editing rules (syntax) of SDF and the kernel language specifies the semantics of each formalism element. An embedded component in figure 19 shows a valid representation of an SDF node. In addition the graphical representation of a SDF node is defined to reflect the expected functionality.

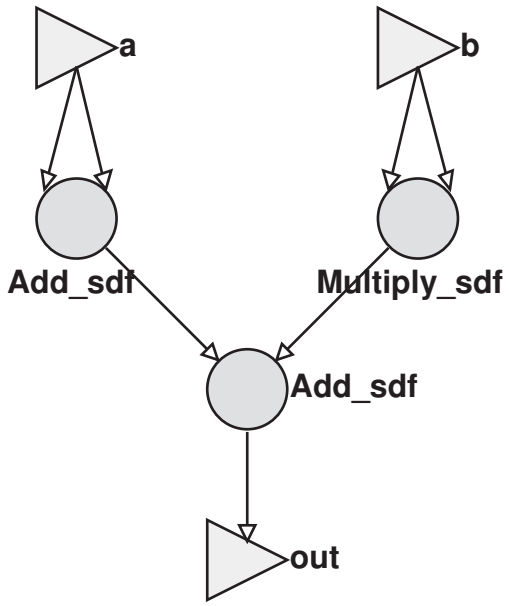

FIGURE 20. Synchronous data flow (SDF) representation of a component containing embedded components with an equivalent functionality as that in figure 19.

4.3 Proposed Methodology

An object oriented methodology describes how systems are designed, based on the concept of communicating objects. The main steps of requirements analysis, domain analysis and system design are common to most existing methodologies. These steps involve modelling a system in increasing detail until an implementation can be derived from the refined models. The choice of modelling language is the main difference between various methodologies. Real-time object-oriented modelling (ROOM) [100] uses hierarchical

finite state machines and Object Modelling Technique (OMT) [99] uses hierarchical finite state machines and data flow representations of an object's behaviour. Model-based Object Oriented Systems Engineering (MOOSE) [60] uses object interaction diagrams and the Booch method [22], [112] simply defines relationships between objects.

The proposed methodology combines aspects of object oriented methodologies such as ROOM [100] and OMT [99] and the model based codesign methodology [27]. It is based on object oriented high level time Petri nets, a modelling language in which system models may be created directly or indirectly by supporting other formalisms. It is highly iterative and is well suited to creating models which are continually refined towards an implementation. As many implementation possibilities exist for a given system functionality and constraints set, design exploration is actively supported. The proposed methodology includes the following steps:

1. define the system and its boundary with the environment. This boundary may change during the course of developing a system if, for example, more intelligence is delegated to the input sensors and output actuators;
2. formulate *scenarios* [99] that the system must implement. Initial scenarios focus on system interaction leading to more detailed scenarios. Scenarios are the main method of describing the required system functionality;
3. using object oriented high level time Petri nets kernel language and / or a derived formalism create a sufficiently abstract model of a system's environment. This model is refined during the course of a design. It is used as a test environment and may be successively replaced by the real environment;
4. define a set of kernel language components representing the system structure. This structure is derived by analysing scenarios to identify candidates for concurrent execution, between which messages flow, and by implementing further scenarios;
5. key components are developed and prototyped and the system is validated at various stages during the development of the component class definitions;

6. partition the system model, introduce interfaces between implementation devices and analyse the result to determine whether it meets the design constraints;
7. refine model by repeating the previous two steps in such a way that backtracking is possible. Ensure that the design space has been sufficiently traversed. Generate code as input into software and hardware implementation processes.

Steps 3 to 7 are directly supported by the kernel language. They describe the activities to be executed using kernel language based models that are required when designing an embedded system. This methodology is applied to the design example in chapter 7.

Figure 21 shows a particular instance of the proposed methodology using the steps defined above. This approach is model based and consists of the continual refinement of models representing the system and its environment to produce new or refined models.

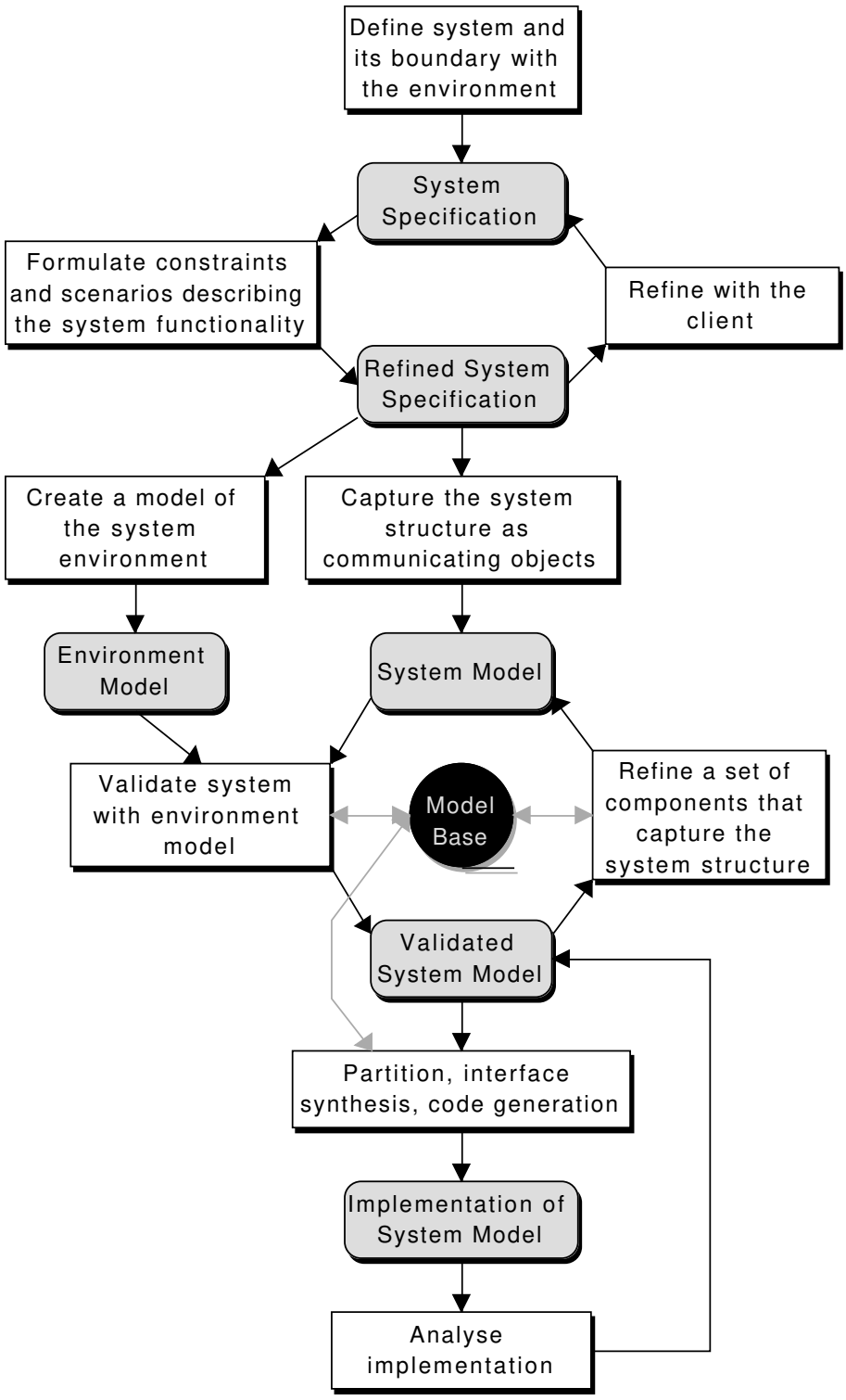

FIGURE 21. Flow diagram of the proposed methodology.

4.4 Summary

The approach in this monograph does not present a complete design process applicable to all types of embedded system design but rather describes a modelling language that addresses the special needs of embedded system design. Critical steps in a design process make use of this language and are combined to form a methodology for a specific application domain or to be used within an existing methodology.

Using the proposed language, component classes are defined which can be refined, reused and configured during the design of an embedded system. The flexibility supported by the kernel language is also available to all kernel based formalisms, as they are also object oriented and can be refined and reused in the same manner as the kernel language. This approach is an extension of well known and tested concepts.

CHAPTER 5 *Kernel Language*

Many existing methods and tools for developing real time systems lack formal foundations and offer no support for managing complexity and quality requirements of such systems. They focus primarily on syntactic issues ignoring critical semantic issues. Formal methods have been advocated by the scientific community for over a decade. They enable unambiguous specification and support a higher level of analysis, i.e. the ability to prove a degree of correctness and reliability, than informal or semi formal methods. Formal methods guarantee that a wide range of semantic errors are detected and removed early in a system's life cycle. However industry has been very slow to adopt such methods [37] citing the difficulty of applying them to specify real time systems. Existing established formal methods such as VDM [6] or Z [101], [43] do not address the requirements of real time systems, while other approaches, such as Esterel [18], LUSTRE [31], APTL [110] and [36], [73] either lack tool support or do not support large specifications.

Several methods use a number of domain specific formalisms with which different aspects of a systems behaviour are modelled [8], [26], [33], [52], [72]. These formalisms may be wholly formal methods or a combination of formal methods and informal ones. Often neglected in the multi formalism environments is the question of the semantics of the combination of formalisms within one specification.

Other approaches are based on augmented standard languages such as Occam [10], [52], ANSI C [40], [46], C++ [74], [106] and VHDL [45],

[106] in addition to approaches based on state machines [7], [26], [62], [64], [67], [108] with their often informal means of modelling data transformations.

A number of approaches are also based on Petri nets [30], [38], [39], [58], [70], [80], [93], [96]. Petri nets, unlike many other formal methods, have an intuitive graphical representation which allows concurrency and synchronisation to be represented in an understandable way. Petri nets have the additional benefit of being general enough to be able to describe many other formalisms. This is exploited in the Cabernet tool [96] which supports user defined formalisms. These formalisms are defined semantically in terms of the underlying Petri net kernel and can be added to by so called meta-users, thus allowing the tool to be user extended in a formal way.

In [57] a method is proposed for supporting multiple paradigms based on a formal kernel language. Also discussed in [57] is the possibility of adding object oriented concepts to a kernel language to aid, not only in supporting multiple paradigms, but also in composition, abstraction and refinement of system models.

The kernel presented in this monograph is based on high level time Petri nets. This class of Petri nets has several advantages:

- simple formal definition, reducing the possibility of ambiguous interpretation of system models;
- intuitive graphical representation;
- operational semantics that support simulation and execution;
- availability of analysis techniques for proving a number of interesting properties;
- ability to be analysed for temporal and functional properties;
- successful use in many different application areas.

Similarly to Cabernet [96], other formalisms can be built from this kernel language. However unlike Cabernet this has been combined with object oriented concepts aiding explorative system design, component reuse and shar-

ing. An important side effect of this approach is that any formalism described in the kernel language is also object oriented with all its inherent benefits.

This chapter formally defines the kernel language with its object oriented extensions. It consists of high level Petri nets used in components, defined using object oriented concepts. The particular class of Petri nets is shown to be composed of four standard Petri net classes. Multi paradigm support is discussed in the following chapter.

5.1 Petri Nets

The following sections describe various classes of Petri nets. All of these classes build upon the basic place-transition Petri net described in [95]. The extensions to the basic Petri net model are motivated by the desire to allow a designer to use the most intuitive and expressive language possible within a formal framework. The model used in this monograph is a composition of various well known and understood Petri nets classes. It satisfies the requirement to provide a formal kernel that is intuitive and expressive without sacrificing formal analysis to prove important system characteristics. In figure 22 the relationship between various Petri net models is shown. It can be seen that the final Petri net model used to formalise the kernel language is derived successively from simpler Petri net classes.

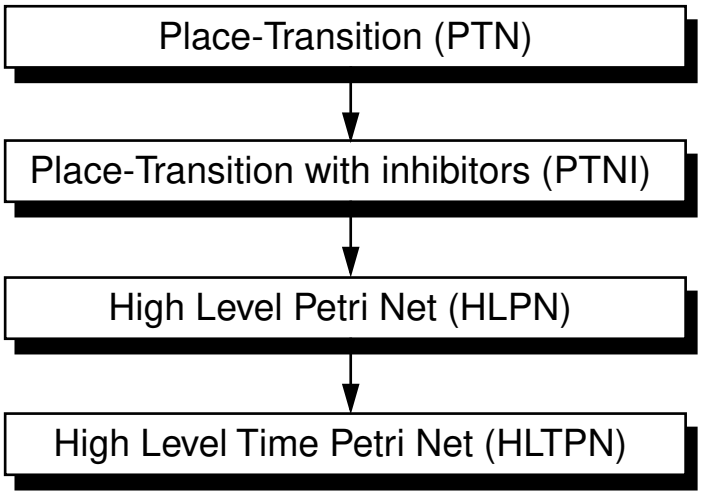

FIGURE 22. The relationship between Petri net classes defined in this chapter

5.1.1 Place-Transition Petri Nets

Petri nets [95] were developed in the early 1960's to study the behaviour of concurrent automata. They are well suited to model event driven distributed systems and have gained in popularity over the intervening years mainly due to their graphical representation and mathematical foundations [89], [94]. The following definitions follow the style presented in [20].

A Petri net is a 3-tuple $N = (P, T, A)$ with:

- $P \cap T = \varnothing$;
- $P \cup T \neq \varnothing$;
- $A \subseteq (P \times T) \cup (T \times P)$.

A Petri net is a bipartite graph with nodes P *(places)*, T *(transitions)* and A *(arcs)* connecting places to transitions or transitions to places. In the usual Petri net graphical representation (figure 23) a circle represents a place, a rectangle represents a transition and arcs are represented as directed arrows.

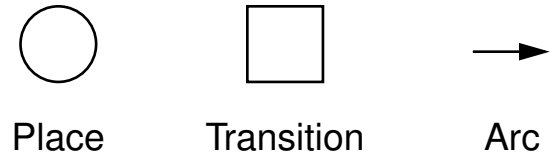

FIGURE 23. Basic Petri net elements

- $\bullet x = \{y | (y, x) \in A\}$ are called the *inputs* of element x;
- $x\bullet = \{y | (x, y) \in A\}$ are called the *outputs* of element x;
- $\bullet x\bullet = \bullet x \cup x\bullet$ are called the *neighbours* of element x;
- $pre : P \cup T \rightarrow A$

  $pre(x) = \{(y, x) \in A\}$ is called the *input arc set* of node x;
- $post : P \cup T \rightarrow A$

  $post(x) = \{(x, y) \in A\}$ is called the *output arc set* of node x.

Figure 24 shows a typical Petri net as defined above.

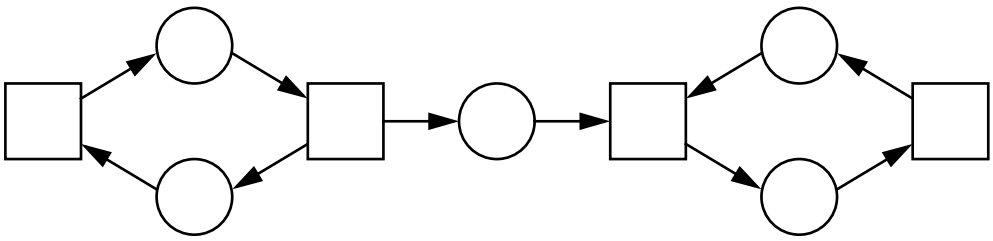

FIGURE 24. A Petri net

Petri net places can hold *tokens*. A token distribution is called a *marking M* and represents a *system state S*, i.e. a system is modelled as having as many states as there are possible markings.

Structure

A place-transition Petri net is defined as a 6-tuple $PTN = (P, T, A, C, W, M_0)$ where:

- (P, T, A) is a Petri net N as above;
- $C : P \rightarrow \mathbf{N} \cup \{\infty\}$ The *capacity* function $C(p)$ defines the maximum number of tokens (by default unlimited) a *place* $p \in P$ can hold;
- $W : A \rightarrow \mathbf{N}$ where $W(a)$ is the *weight* function of arc $a \in A$. The *weight* function defines the number of tokens removed from or added to a place when a transition *occurs* (fires). In the absence of an explicit weight function it is assumed to be 1;
- $M_0 : P \rightarrow \mathbf{N}_0$ is an initial marking and $\forall p \in P : M_0(p) \leq C(p)$. The initial marking of a place is represented graphically by $M_0(p)$ dots as shown in figure 25.

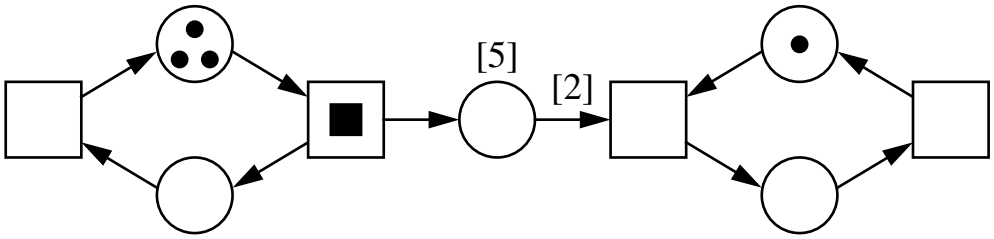

FIGURE 25. Place-transition Petri net with a place capacity and arc weight. A solid box indicates that the containing transition is enabled (see following definition).

The above definitions define the *static* structure of a Petri net. Rules exist for the movement of tokens which represent the *dynamic* behaviour of a system, to be described. Such a movement of tokens is interpreted as a change in system state.

A function $M : P \rightarrow \mathbf{N}_0$ is called a net marking where $M(p)$ is the number of tokens on place p and $M(p) \leq C(p)$ for all $p \in P$. This is represented

graphically by $M(p)$ dots contained within place p. The dynamic behaviour of a system is modelled by the movement of tokens. Token movement is instigated by transitions according to the following rules.

Enabling Conditions

Transition $t \in T$ is said to be *enabled* with marking M when the following relations hold:

- $\forall p \in {\bullet t} : M(p) \geq W(p, t)$, if for all input places there are enough input tokens;
- $\forall p \in t{\bullet} \backslash {\bullet t} : M(p) \leq (C(p) - W(t, p))$, if for all places that are only outputs of t there is enough remaining capacity;
- $\forall p \in (t{\bullet} \cap {\bullet t}) : M(p) \leq (C(p) - W(t, p) + W(p, t))$, for all places that are inputs and outputs of t there is enough remaining capacity after the input tokens have been removed.

Occurrence

Transition t is said to have *occurred* (fired) from M to M' when, with marking M transition t is *enabled*, tokens have been removed from every input place and tokens have been generated for every output place (figure 26) as follows:

- $\forall p \in {\bullet t} \backslash t{\bullet} \Rightarrow M'(p) = M(p) - W(p, t)$, for all places that are only inputs of transition t the number of tokens defined by the weight function for the arc from the place to the transition is removed;
- $\forall p \in t{\bullet} \backslash {\bullet t} \Rightarrow M'(p) = M(p) + W(t, p)$, for all places that are only outputs of transition t the number of tokens defined by the weight function for the arc from the transition to the place is added;

- $\forall p \in (t\bullet \cap \bullet t) \Rightarrow M'(p) = M(p) - W(p, t) + W(t, p)$, for all places that are inputs and outputs of transition t the number of tokens defined by the weight function for the arc from the place to the transition is removed and the number of tokens defined by the weight function for the arc from the transition to the place is added.

M' is called the next marking of *M* after the occurrence of *t*.

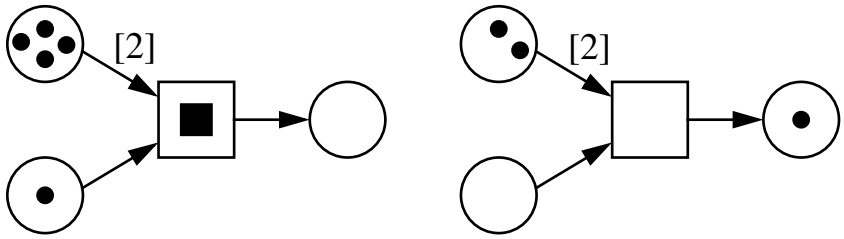

FIGURE 26. Place-transition Petri net before and after the occurrence of the transition. Note default weight = 1 and default place capacity = ∞

A system state *S* modelled with a place-transition Petri net is defined by a marking *M*. An important characteristic of Petri nets is that state changes are determined locally, that is transitions can occur independently of each other without the need of a *global* arbiter if they have no common input and capacity limited output places. Consequently Petri nets are particularly well suited for modelling distributed systems.

It is important to note that when a transition is enabled it is not forced to occur. There is no implicit synchronism in Petri nets that forces all enabled transitions to fire before any subsequently enabled transitions. In effect after the occurrence of a transition a new set of enabled transitions exists in which the next transition to occur is a member. In addition a Petri net does not define an ordering of occurrences of concurrently enabled transitions. In figure 27 transitions t1 and t2 are both enabled. The following occurrence sequences $(t1, t2, t3)$, $(t2, t1, t3)$ and $(t2, t3, t1)$ are all possible. Thus the net needs to be extended when a particular order of transition occurrences is desired.

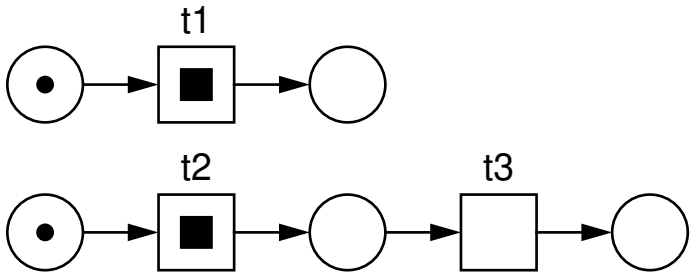

FIGURE 27. A place-transition Petri net with two *concurrently* enabled transitions

Conflicts model non determinism and are useful in describing abstract system behaviour. In a refined system model such non determinism is undesirable but may still be useful for modelling a system's (test) environment. Two transitions are said to be in conflict if under a particular marking *M*, they are both enabled and if after one transition occurs the other transition is no longer enabled. In figure 28 a conflict situation is shown where two transitions share a common input place.

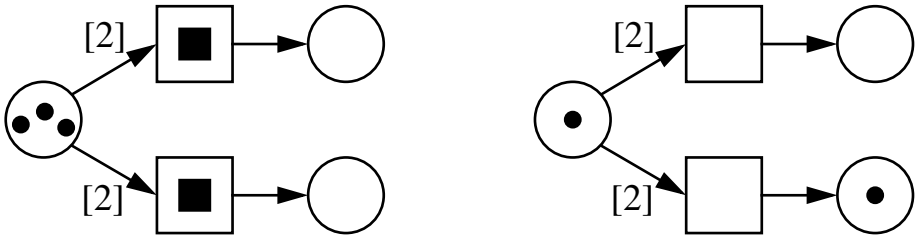

FIGURE 28. A place-transition Petri net in *input* conflict before and after the occurrence of a transition

A conflict may also occur when two transitions share a common output place with a limited capacity which would be reached after the occurrence of a single transition as shown in figure 29.

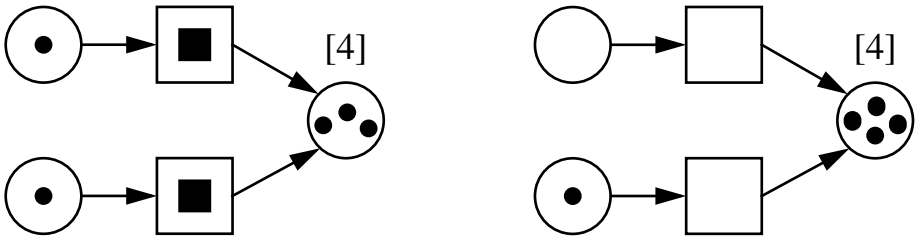

FIGURE 29. A place-transition Petri net in *output* conflict before and after the occurrence of a transition

5.1.2 Petri Nets with Inhibitor Arcs

Many additions to Petri nets have been proposed mainly in order to improve support for particular application specific domains. Some additions such as arc weights and place capacities enhance the syntax of Petri nets, allowing certain structures to be defined more succinctly. *Inhibitor* arcs [13], [76], however, extend the expressiveness of place-transition Petri nets as they can not, in general, be represented by a Petri net without inhibitors arcs.

Structure

A Petri net with inhibitor arcs is a 4 tuple:

$PTNI = (P, T, A, I)$ such that (P, T, A) is a Petri net and $I \subseteq P \times T$.

Inhibitor arcs only connect places to transitions. Figure 30 shows a net containing an inhibitor arc from *place* to *t1*. An inhibitor arc prevents a transition from being enabled if the associated place contains tokens. In this way it is possible to test whether a place contains tokens or not. Thus in the following net if *place* contains a token, transition *t1* is not enabled. This effectively defines an occurrence order between transitions *t1* and *t2*.

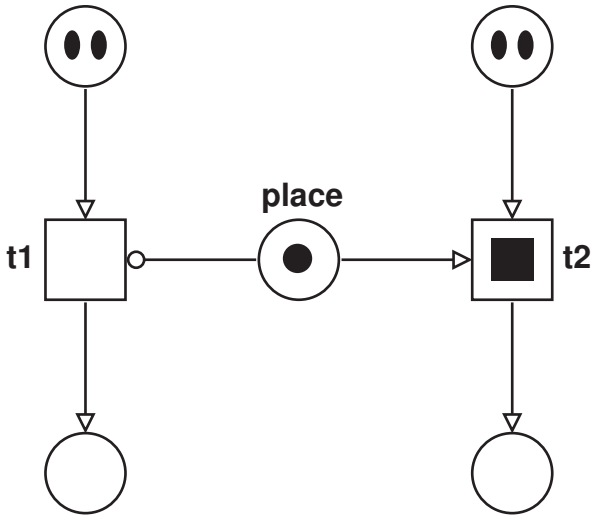

FIGURE 30. A Petri net with an inhibitor arc

After transition *t2* has occurred *place* does not contain any tokens. This is shown in figure 31. Note: transition *t1* is now enabled and the marking of *place* will not change when *t1* occurs.

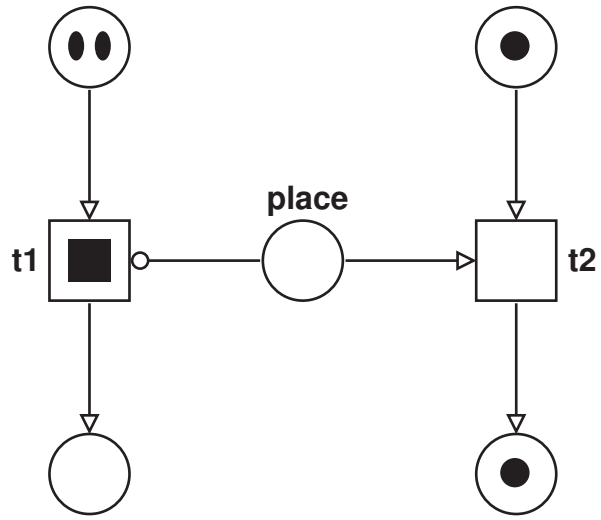

FIGURE 31. A Petri net with an inhibitor arc after the occurrence of t2

It is also possible to define *weighted* inhibitor arcs where the number of tokens on the associated place must be less than the inhibitor arc weight. A place-transition Petri net with weighted inhibitor arcs can be defined as a 8-tuple $PTNI = (P, T, A, I, C, W, W_I, M_0)$ and:

- (P, T, A, C, W, M_0) is as defined for a place-transition Petri net;
- $I \subseteq P \times T$ are called *inhibitor* arcs;
- $W_I : I \rightarrow \boldsymbol{N}$ where $W_I(i)$ is the inhibitor weight function of an inhibitor arc $i \in I$. The inhibitor weight defines the minimum number of tokens that must be present to prevent a transition from being enabled.

Enabling Condition / Occurrence

The activation rule for place-transition Petri nets with weighted inhibitor arcs is defined as:

- $t \in T$ is enabled if it is enabled in PTN and $\forall p \in \bullet t,\ (p, t) \in I : M(p) < W_I(p, t)$;
- the effect of the transition occurrence is the same as in PTN.

In general it is not decidable whether the number of tokens per place in a Petri net with inhibitor arcs is bounded [3], [94]. This would seem to be a major problem since it is highly desirable to be able to prove that the Petri net is bounded, as it is directly related to the memory requirement of the modelled system's implementation. However inhibitor arcs can be modelled with standard constructs if the place to be tested has a bounded token capacity as shown in figure 32. Here, in the top Petri net fragment, place *p* has a capacity *b* and contains *m* tokens. When place *p* is empty, transition *t* is enabled. In the lower net fragment place *p* is augmented with a complementary place *p'* holding *b-m* tokens. When tokens are added to place *p* the same number of tokens is removed from the complementary place *p'*. Similarly when tokens are removed from place *p* the same number of tokens is added to the complementary place *p'*. When the number of tokens in place *p'* is

equal to *b*, place *p* will be empty and transition *t* enabled.Thus system models containing inhibitor arcs can be analysed, provided the tested place has an additional capacity constraint or can be proved to have bounded capacity.

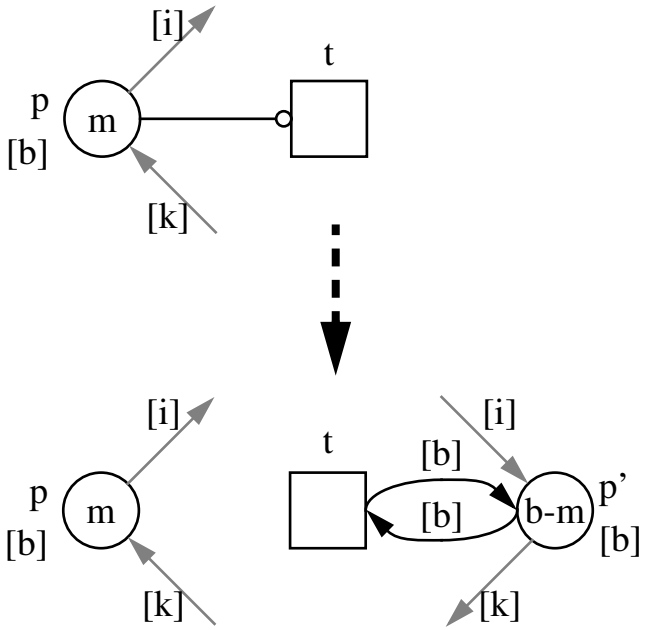

FIGURE 32. Equivalent nets with and without an inhibitor arc

5.1.3 High Level Petri Nets

Place-transition nets have been used extensively to model aspects of concurrent systems. Unfortunately this class of Petri nets suffers from an explosion in size when realistic systems of even moderate size are modelled. In the early 1980's *high level Petri nets* were proposed to overcome some of the problems with the classic Petri nets.

In traditional Petri net formalisms the kind of data modelled by the net is determined by the token types. In elementary nets there is only one type with only one value, therefore the state of the modelled system is completely defined by the net marking (number of tokens per place). In high level nets tokens can have arbitrary types such as boolean, integer, real, arrays or records etc. In addition to extended token types, high level Petri

nets have net inscriptions to manipulate the data values of tokens. Thus the state of a system is the token marking together with the data value of each token in that marking. A high level Petri net can be used not only to model control flow but also data flow. The fire rule is extended to allow a transition to be enabled, depending on the result of a Boolean function on the input token data. When a transition occurs the output token data values are calculated using only the input token data values. A number of high level Petri net models have been proposed [55], [69]. The following definition builds upon the *PTNI* Petri net class defined before and adds a simplified form of the predicate-transition Petri net model.

Structure

A high level Petri net is an 11-tuple:

$HLPN = (P, T, A, I, C, W_I, M_0, X, O, G, F)$ and

- (P, T, A, I, C, W_I) as previously defined;
- O is a set of all possible *objects* in the *HLPN*. A token is represented as a single object (the token data) whose value represents the token data value;
- $X : A \rightarrow \wp(V)\backslash\varnothing$ is called the *arc inscriptions* and is the non empty power set of V where V is a sufficiently large set of variable names.

  $\forall t \in T, \forall a, b \in pre(t) \cup post(t) : a \neq b \Rightarrow X(a) \cap X(b) = \varnothing$

  For every transition the variables associated with input and output arc inscriptions are unique.

  An arc is inscribed with a set of variables where the number of variables can be considered to be the *weight* of the arc (see figure 33);

FIGURE 33. Representing weights in place-transition and high level Petri nets. Both arcs have an equivalent weight function = 2.

- $M_0 : P \rightarrow O^*$ where $\left|M_0(p)\right| \leq C(p) \quad \forall p \in P$.

  O^* is defined as a multi set of objects O.

  The initial marking is a multi-set of objects (tokens) in which every possible object may appear more than once. A token is an element of the set of all objects O. The initial marking must conform with the capacity function;

- $In(t) = \bigcup_{a \in pre(t)} X(a)$.

  $In(t)$ is the set of all the input variables of transition $t \in T$;

- $Out(t) = \bigcup_{a \in post(t)} X(a)$.

  $Out(t)$ is the set of all the output variables of transition $t \in T$;

- $b_{in}(t) = O^{In(t)}, t \in T$.

  Input *bindings* $b_{in}(t)$ is defined to be the set of all functions that map input variables of transition t with objects;

- $b_{out}(t) = O^{Out(t)}, t \in T$.

  Output *bindings* $b_{out}(t)$ is defined to be the set of all functions that map output variables of transition t with objects;

- $G(t) : b_{in}(t) \rightarrow Boolean, t \in T$.

  $G(t)$ is called the guard function of transition t. It provides an additional guard on input bindings that must evaluate to the Boolean value *TRUE* for a *transition instance* to be enabled;

- $F(t) : b_{in}(t) \rightarrow b_{out}(t), t \in T$.

  Each transition is assigned a function $F(t)$ that produces output bindings with input bindings as parameters.

The *State S* of a *HLPN* is defined as:

- $S = M : P \rightarrow O^{*}$ and $\forall p \in P, |M(p)| \leq C(p)$.

  The state of a *HLPN* is defined as the net marking where the number of objects per place is always less than or equal to the place capacity.

Figure 34 shows a high level Petri net where v_1, v_2, v_3, v_4 are the arc inscriptions of transition *t1*, i.e. $X(p1, t1) = \{v_1\}$, $X(p2, t1) = \{v_2, v_3\}$, $X(t1, p3) = \{v_4\}$, $In(t) = \{v_1, v_2, v_3\}$ and $Out(t) = \{v_4\}$. As places *p1, p2, p3* have no explicit capacity function it is by default ∞. Transition *t1* has a guard function $G(t) = v_1 \leq v_2$ and function $F(t)$ where $v_4 = v_1 - v_3$.

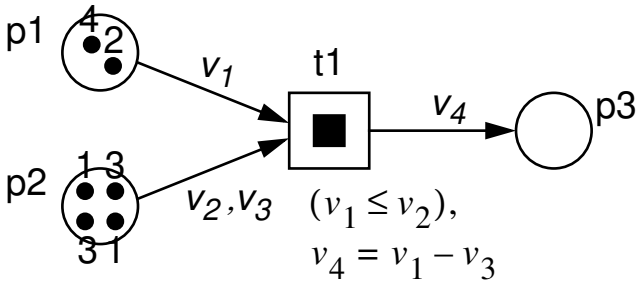

FIGURE 34. A high level Petri net

Enabling Conditions

In high level Petri nets a distinction is made between whether a transition is *active*, i.e. the correct number of tokens are present on the input and out places and *enabled transition instances* which represents bindings of token objects to input variables. The *active* transitions in marking M are defined as:

- $$TA(M) = \left\{ t \in T \middle| \begin{array}{c} \forall p \in {\bullet t} : |M(p)| \geq |X(p,t)| \\ \forall p \in (t{\bullet} \backslash {\bullet t}) : |M(p)| \leq (C(p) - |X(t,p)|) \\ \forall p \in (t{\bullet} \cap {\bullet t}) : |M(p)| \leq (C(p) - |X(t,p)| + |X(p,t)|) \\ \forall p \in {\bullet t},\ (p,t) \in I : |M(p)| < W_I(p,t) \end{array} \right\}$$

  For transition t to be active the following four conditions must hold:

1. for all places that are inputs to t the number of objects on each place must be greater than or equal to the number of variables inscribed on the connecting arc;
2. for all places that are only outputs of t the number of objects on each place must be less than or equal to the place capacity minus the number of variables inscribed on the connecting arc;
3. for all places that are both inputs and outputs of t the number of objects on each place must be less than or equal to the place capacity, minus the number of variables inscribed on the connecting arc from the transition to the place, plus the number of variables inscribed on the connecting arc from the place to the transition;
4. for all places connected to transition t with an inhibitor arc the number of objects on each place must be less than the inhibitor weight arc weight.

- $$b_{val}(t, M) = \left\{ b \in b_{in}(t) \middle| \forall p \in {\bullet t}, \forall o \in O \ \left| [b|_{X(p,t)}]^{-1}(o) \right| \leq M(p)(o) \right\}$$

  A valid binding of t is a binding where for every input place each object is referenced at most as often as it occurs in p under the marking M.

All *enabled transition instances* in marking *M* are defined as:

- $TI(M) = \left\{ (t, b(t)) \in TA(M) \times b_{in}(t) \middle| \begin{matrix} b(t) \in b_{val}(t, M) \\ G(t)(b(t)) = TRUE \end{matrix} \right\}$

  An enabled transition instance of active transition *t* consists of a valid binding of input variables and objects and with the transition guard function operating on the binding evaluating to TRUE. A transition instance for transition *t* is referred to as t_i.

Therefore the *enabled transitions* (compared with the definition of enabled transitions for place-transition Petri nets) can be defined as:

- $ET(M) = \{ t \in T \mid \exists i \in TI(M) : i = (t, b(t)) \}$

  A transition *t* is considered enabled if an enabled transition instance exists for *t*.

Occurrence

As a result of enabled transition instance t_i occurring, the tokens (objects) from the input places that were bound to the input variables are removed and tokens produced by the transition function *F(t)(b(t))* are added to the output places. The marking *M* is transformed into marking *M'* as follows:

- $\forall p \in P \backslash {\bullet t \bullet} \Rightarrow M'(p) = M(p)$.

  For all places that are neither input nor outputs of transition *t* the marking remains the same;

- $\forall p \in {\bullet t} \backslash {t \bullet} \Rightarrow M'(p) = M(p) \backslash b(t)(In(t))$.

  For all places that are only inputs of transition *t* the objects associated with the binding variables, defined on the connecting arc inscription, are cut;

- $\forall p \in {t \bullet} \backslash {\bullet t} \Rightarrow M'(p) = M(p) \cup F(t)b(t)(Out(t))$.

For all places that are only outputs of transition *t* the objects bound with the output variables produced by the transition function *F(t)(b(t))* are added;

- $\forall p \in (t\bullet \cap \bullet t) \Rightarrow M'(p) = M(p) \backslash b(t)(In(t)) \cup F(t)b(t)(Out(t))$

  For all places that are inputs and outputs of transition *t* the objects associated with the binding variables, defined on the input connecting arc inscription, are cut and the objects bound with the output variables produced by the transition function *F(t)(b(t))* are added.

Figure 35 shows a high level Petri net before and after the occurrence of transition instance $t1_3$.

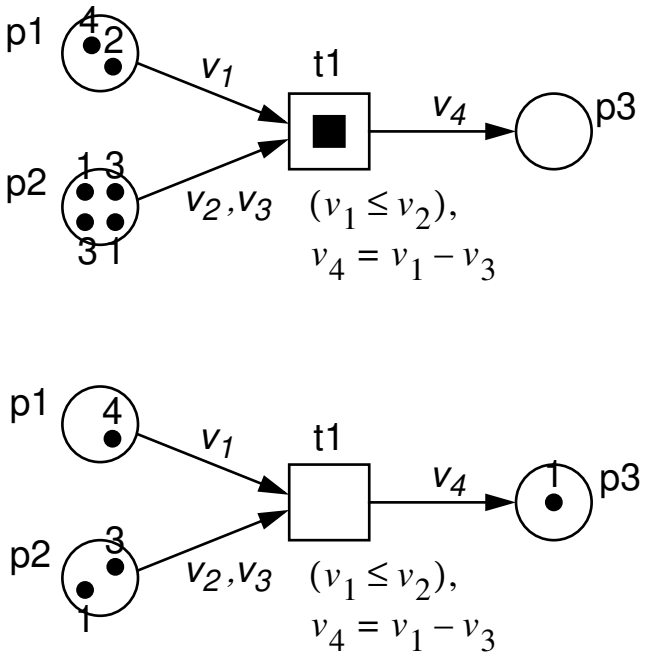

FIGURE 35. A high level Petri net before and after the occurrence of $t1_3$

The following table lists all the possible bindings for the Petri net in figure 35. From the set of unique bindings only $t1_3$ and $t1_4$ are enabled, as

the transition guard function *G(t1)(b(t1))* for the remaining bindings evaluates to FALSE.

$t1_i$	v_1	v_2	v_3	G(t1)(b(t1))
1	2	1	1	FALSE
2	2	1	3	FALSE
3	2	3	1	TRUE
4	2	3	3	TRUE
5	4	1	1	FALSE
6	4	1	3	FALSE
7	4	3	1	FALSE
8	4	3	3	FALSE

TABLE 1. All possible bindings of Petri net in figure 35

5.1.4 High Level Time Petri Nets

The Petri net model used for the kernel language is a composition of high level Petri nets, defined in the previous section, and *time* Petri nets. It defines the minimum set of expressiveness needed to describe the functional and behavioural aspects of the target systems. In this section the model will be formally defined and some observations regarding analysis will be made.

Time plays an important role in all embedded real time systems and any language that purports to model such systems in any level of detail must include a concept of time. Time models for Petri nets and their analysis are described in [19], [102]. Two basic Petri net models for handling time have been proposed - *time* Petri nets [87], [88] and *timed* Petri nets [98]. *Timed* Petri nets associate a finite firing duration with each transition of a net. The firing rule of these Petri nets is modified to include the time it takes to fire a transition under the assumption that a transition occurs as soon as it is enabled. These Petri nets and the subsequently developed *stochastic* Petri nets [4] have been mainly used for performance evaluation.

Time Petri nets associate a finite enabling duration with each transition in a net. For a transition to occur it must have been continuously enabled for at

least the minimum duration and for not more than the maximum duration. Time Petri nets are more general than timed Petri nets, i.e. a timed Petri net can be modelled with a time Petri net, as shown in figure 36, but the converse is not true. Time Petri nets are adequate for expressing most temporal constraints. Analysis techniques for formal verification of time dependent behaviour in time Petri nets are described in [19]. More interesting is the technique described in [25] where a compositional approach to the analysis of time Petri nets components is proposed.

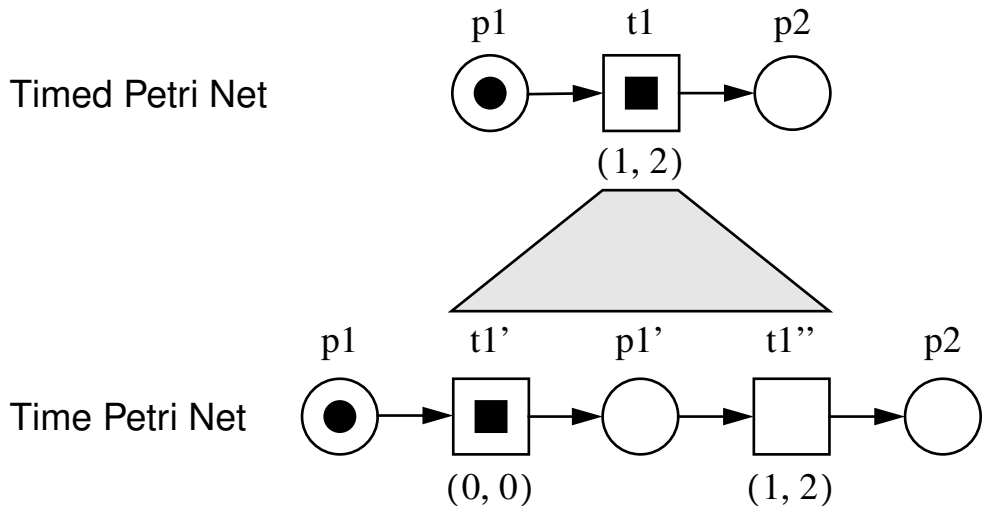

FIGURE 36. Modelling a *timed* Petri net with a *time* Petri net

Conceptually time information can be regarded as imposing additional constraints on a Petri net, reducing the number of possibilities of how the Petri net can evolve over time. In figure 37 the time constraints are interpreted as enabling delays for transitions, i.e. a *time* Petri net. A transition is inscribed with a tuple *(a, b)* denoting the earliest and latest firing time after the transition is enabled. The transition must be enabled for at least a time units and not more than b time units before it can occur and $0 \leq a \leq b$.

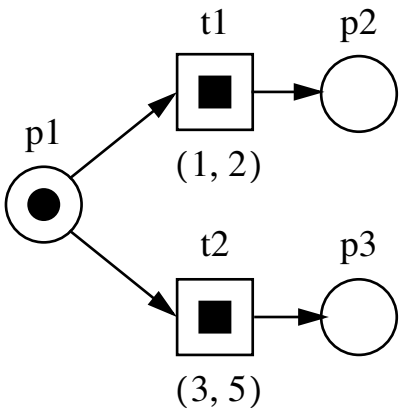

FIGURE 37. A *time* Petri net

If time is ignored, both transitions *t1* and *t2* in the above Petri net are in conflict and both could occur. However including the time model only *t1* can occur thus resolving the conflict and making *t2* and *p3* redundant. Note: if the two time intervals overlap the conflict would still exist.

Structure

This section defines the class of high level time petri nets used in the kernel language. A high level time Petri net is an 13-tuple:

$HLTPN = (P, T, A, I, C, W_I, M_0, X, O, G, F, E, L)$ and

- $(P, T, A, I, C, W_I, X, O, G, F)$ is as defined for *HLPN*;
- $E : \mathrm{T} \rightarrow \boldsymbol{Q}_0^{+}$ is the earliest *static* firing time, i.e. the earliest time a transition instance can occur after it has become enabled;
- $L : \mathrm{T} \rightarrow \boldsymbol{Q}_0^{+} \cup \{\infty\}$ is the latest *static* firing time, i.e. the latest time a transition instance can occur after it has become enabled. Together with the earliest *static* firing time it defines a *static firing interval* i.e. a tuple of rational numbers[1]. Where $\forall t \in T$, $0 \leq E(t) \leq L(t)$;

- $id : ID \xrightarrow{p} O$. *id* is a partial function mapping the set of global identifiers *ID* to objects. *ID* is defined to be sufficiently large. A token is represented as an unique identifier with the function *id* mapping (token) identifiers to objects which represents the token data.This enables objects to be identified, which is a requirement for time Petri nets. This requirement arises as two objects may have the same value and be used in separate bindings. As will be discussed below, transition instances contain an independent dynamic firing interval and continue to exist until either they occur or are deleted. The occurrence of a transition instance cuts identifiers from the input places and causes all transition instances containing references to these identifiers to be deleted. This is in contrast to *HLPN* where all transition instances can be constructed after every occurrence;
- $b_{in}(t) = ID^{In(t)}, t \in T$.

  Input *bindings* $b_{in}(t)$ is defined to be the set of all functions that map input variables of transition *t* with global identifiers;
- $b_{out}(t) = ID^{Out(t)}, t \in T$.

  Output bindings $b_{out}(t)$ is defined to be the set of all functions that map output variables of transition *t* with global identifiers.

Figure 38 shows a high level time Petri net where v_1, v_2, v_3, v_4 are the arc inscriptions of transition *t1*, i.e. $X(p1, t1) = \{v_1\}$, $X(p2, t1) = \{v_2, v_3\}$, $X(t1, p3) = \{v_4\}$, $In(t) = \{v_1, v_2, v_3\}$ and $Out(t) = \{v_4\}$. As places *p1, p2, p3* have no explicit capacity function it is by default ∞. Transition *t1* has a guard function $G(t) = v_1 \leq v_2$ and function $F(t)$ where $v_4 = v_1 - v_3$. The static firing interval for transition *t1* is

1. The assumption that the firing intervals are expressed as rational numbers ensures bounded characteristics not guaranteed by real numbers [19].

represented by the tuple *(1,2)*. The places contain tokens represented by identifiers $m1 \ldots m6$.

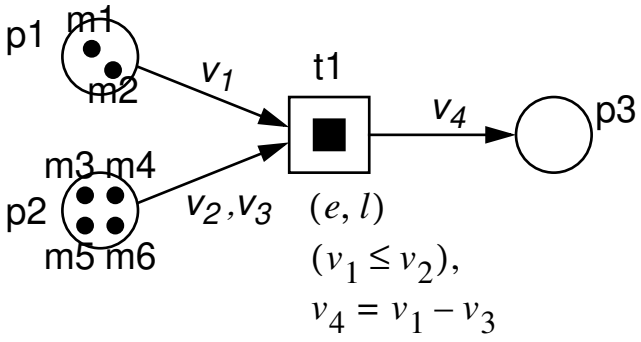

FIGURE 38. A high level *time* Petri net

The function *id* maps global identifiers to objects. The following table shows the mapping for the Petri net in figure 38.

Identifier	Object
m1	2
m2	2
m3	1
m4	3
m5	1
m6	3

TABLE 2. The mapping of global identifiers to objects for figure 38

Enabling Conditions

The state of a high level time Petri net is defined not only by the token marking but also by the function *id*, which maps global identifiers (tokens) to objects, and by the set of transition instances. A transition instance is a tuple of transition, binding and dynamic firing interval. In this way time is an attribute of every transition instance, i.e. a transition may have many transition instances each with independent firing intervals.

The *State S* of a high level time Petri net is defined as $S = (id, M, TI)$ where:

- $M : P \rightarrow \boldsymbol{P}(ID)$ where $p1 \neq p2 \Rightarrow M(p1) \cap M(p2) = \emptyset$ and $|M(p)| \leq C(p) \quad \forall p \in P$. The marking M of a *HLTPN* assigns each place a unique set of identifiers. These identifiers can be mapped with function *id* to objects;
- The set of *transition instances* is defined as:

  $TI = \{(t, b(t), \tilde{E}, \tilde{L}) \in T \times b_{in}(t) \times \boldsymbol{Q}_0^{+} \times (\boldsymbol{Q}_0^{+} \cup \{\infty\}) \Big|$

  t active $\wedge\ b(t)$ valid $\wedge\ G(t)(b(t) \bullet id) = TRUE\}$ where:

  t active in $M \Leftrightarrow \quad \forall p \in {\bullet t} : |M(p)| \geq |In(t)|$ and

  $\forall p \in t\bullet \backslash \bullet t : |M(p)| \leq (C(p) - |Out(t)|)$ and

  $\forall p \in (t\bullet \cap \bullet t) : |M(p)| \leq (C(p) - |Out(t)| + |In(t)|)$ and

  $\forall p \in \bullet t,\ (p, t) \in I : |M(p)| < W_I(p, t)\}$.

  t is active if the correct number of tokens on the inputs and outputs are present.

  b valid in $M \Leftrightarrow \quad range(b(t)) \subseteq \bigcup_{p \in \bullet t} M(p) \wedge b(t)$ injective.

  b can only bind identifiers which exist in M.

  $\tilde{E} : \mathrm{T} \rightarrow \boldsymbol{Q}_0^{+}$ is the earliest *dynamic* firing time, i.e. the minimum time a transition instance must wait before it can occur.

  $\tilde{L} : \mathrm{T} \rightarrow \boldsymbol{Q}_0^{+} \cup \{\infty\}$ is the latest dynamic firing time, i.e. the maximum time a transition instance waits before it can occur.

  TI is a set of transition instances each consisting of a transition, bindings and earliest and latest dynamic firing times.

The set of *enabled transition instances* is defined to be: $\tilde{TI} = \{(t, b, \tilde{E}, \tilde{L}) \in TI \mid \neg\exists(t', b', \tilde{E}', \tilde{L}') \in TI : \tilde{E} > \tilde{L}'\}$.

For all members of the *enabled* transition instances the earliest *dynamic* firing time is not greater than the latest firing time of any other transition instance. In other words the set of enabled transition instances is a subset of the set of transition instances and of which the next transition instance that can occur is a member.

Having defined the state *S*, the initial state S_0 will be defined. Subsequently, as the result of an enabled transition instance occurrence, the next state *S'* will be defined.

The *Initial State* S_0 of a high level time Petri net is defined as $S_0 = (id_0, M_0, TI_0)$ where:

- $dom(id_0) = \bigcup_{p \in P} M_0(p)$ is the set of identifiers in the initial marking;
- $M_0 : P \rightarrow \wp(ID)$ is called the initial marking of the *HLTPN* and is a power set of identifiers. Where $p1 \neq p2 \Rightarrow M_0(p1) \cap M_0(p2) = \varnothing$ and $\left|M_0(p)\right| \leq C(p) \quad \forall p \in P$;
- The *initial* set of *transition instances* is:

  $TI_0 = \{(t, b(t), E, L) \in T \times b_{in}(t) \times \boldsymbol{Q_0}^+ \times (\boldsymbol{Q_0}^+ \cup \{\infty\}) \Big|$

  t active $\wedge\, b(t)$ valid $\wedge\, G(t)(b(t) \bullet id_0) = TRUE\}$ where:

  t active in $M_0 \Leftrightarrow \quad \forall p \in \bullet t : \left|M_0(p)\right| \geq |X(p, t)|$ and

  $\forall p \in (t\bullet \backslash \bullet t) : \left|M_0(p)\right| \leq (C(p) - |X(t, p)|)$ and

  $\forall p \in (t\bullet \cap \bullet t) : \left|M_0(p)\right| \leq (C(p) - |X(t, p)| + |X(p, t)|)$ and

  $\forall p \in \bullet t,\ (p, t) \in I : \left|M_0(p)\right| < W_I(p, t)\}$.

  t is active if the correct number of tokens on the inputs and outputs are present.

$b(t)$ valid in $M \Leftrightarrow \; range(b(t)) \subseteq \bigcup_{p \in \bullet t} M_0(p) \wedge b(t)$ injective, i.e. *b(t)* can only bind identifiers which exist in M_0.

E, L are the *earliest* and *latest static* firing times defined above.

TI_0 is a set of initial transition instances each consisting of a transition, bindings, earliest static firing times and latest static firing times.

Occurrence

When an enabled transition instance $t_i \in \tilde{TI} = (t, b, \tilde{E}, \tilde{L})$ occurs the *HLTPN* moves from state $S = (id, M, TI)$ to state $S' = (id', M', TI')$. The firing time $\tau(t_i)$ is constrained to be greater than or equal to the earliest firing time of t_i and not later than the latest firing time of every other enabled transition instance. Therefore the new state *S'* is defined where:

- $id \rightarrow id'$, i.e. new identifiers are allocated as necessary where

  $id : ID \xrightarrow{p} O$ so that:

  $\forall x \in dom(id) : id(x) = id'(x)$, all existing identifiers are preserved.

  $\exists newid : Out(t) \rightarrow ID \backslash dom(id) : id' \bullet newid = b'$. The function *newid* transforms output variables to new identifiers;

- $M \rightarrow M'$. The marking *M* is transformed into marking *M'* as follows:

  $\forall p \in P \backslash \bullet t \bullet \Rightarrow M'(p) = M(p)$ for all places that are neither input nor outputs of transition *t* the marking remains the same.

  $\forall p \in \bullet t \backslash t \bullet \Rightarrow M'(p) = M(p) \backslash b(t)(In(t))$ for all places that are only inputs of transition *t* the identifiers associated with the binding variables, defined on the connecting arc inscription, are cut.

  $\forall p \in t \bullet \backslash \bullet t \Rightarrow M'(p) = M(p) \cup newid(F(t)b(t)(Out(t)))$ for all places that are only outputs of transition *t* the identifiers, defined on the connecting arc inscriptions, are added.

$\forall p \in (t\bullet \cap \bullet t) \Rightarrow$

$M'(p) = M(p) \backslash b(t)(In(t)) \cup newid(F(t)b(t)(Out(t)))$.

For all places that are both an input and output of transition *t* the identifiers associated with the binding variables, defined on the input connecting arc inscription, are cut and the identifiers defined on the output connecting arc inscriptions are added;

- $TI \rightarrow TI'$. The transition instances *TI* are transformed to *TI'* as follows:

$$TI' = \{(t', b'(t), \tilde{E}', \tilde{L}') \in T \times b_{in}(t) \times \boldsymbol{Q}_0^{+} \times (\boldsymbol{Q}_0^{+} \cup \{\infty\}) \mid$$

t' active $\wedge\ b'(t)$ valid $\wedge\ G(t')(b'(t) \bullet id') = TRUE\}$ where:

t active in M', $b(t)$ valid in M' and

$$(\tilde{E}', \tilde{L}') = \left\{ \begin{array}{c} (max(0, \tilde{E}'' - \tau(t_i)), max(0, \tilde{L}'' - \tau(t_i))), (t', b', \tilde{E}'', \tilde{L}'') \in TI \\ (E(t'), L(t')), otherwise \end{array} \right\}$$

After the token movement the transition instances are determined and the dynamic firing times updated. For transition instances that are enabled before and after the occurrence of t_i the dynamic firing time is shifted left by the value $\tau(t_i)$.

Differences between various *time* Petri net models arise from how transition instance t_j is treated if it is enabled before and after the occurrence of transition instance t_i. In [25] transition instance t_j is regarded as being newly enabled, i.e. the dynamic firing interval is reset to the static firing interval, whereas in [88] transition instance t_j continues to be enabled and the firing interval is shifted by $\tau(t_i)$. In the chosen model transitions can continue to be enabled, as the first behaviour can be obtained by inserting a place with a single initial token as both input and output to the transition. Thus by default a transition has multiple transition instances with multiple dynamic firing intervals, that is one for each enabled instance. In figure 39 transition *t1* will occur twice within the interval (1,2), i.e. it is certain at time 2 that two tokens will reside on the output place. This is in contrast to the interpretation

in [25] in which a transition instance will occur once within the interval (1,2) and then again within the interval (2,4).

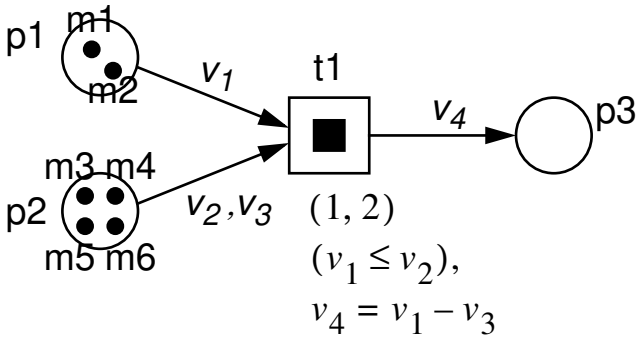

FIGURE 39. A high level *time* Petri net

Table 3 shows a mapping of global identifiers to objects. This mapping is represented by the function *id*. Table 4 shows all the enabled transition instances with bindings of global identifiers to variables and earliest and latest dynamic firing times.

Identifier	Object
m1	2
m2	2
m3	1
m4	1
m5	3
m6	3

TABLE 3. The mapping of global identifiers to objects for figure 39

$t1_i$	v_1	v_2	v_3	$\bar{e}$	$\bar{l}$
1	m1 (2)	m5 (3)	m3 (1)	1	2
2	m1 (2)	m6 (3)	m3 (1)	1	2
3	m1 (2)	m5 (3)	m4 (1)	1	2
4	m1 (2)	m6 (3)	m4 (1)	1	2

TABLE 4. Valid bindings and dynamic firing intervals for all transition instances of t1 at time = 0 for figure 39

$t1_i$	v_1	v_2	v_3	$\bar{e}$	$\bar{l}$
5	m1 (2)	m5 (3)	m6 (3)	1	2
6	m1 (2)	m6 (3)	m5 (3)	1	2
7	m2 (2)	m5 (3)	m3 (1)	1	2
8	m2 (2)	m6 (3)	m3 (1)	1	2
9	m2 (2)	m5 (3)	m4 (1)	1	2
10	m2 (2)	m6 (3)	m4 (1)	1	2
11	m2 (2)	m5 (3)	m6 (3)	1	2
12	m2 (2)	m6 (3)	m5 (3)	1	2

TABLE 4. Valid bindings and dynamic firing intervals for all transition instances of t1 at time = 0 for figure 39

Figure 40 shows the same net after the occurrence of enabled transition instance $t1_4$. Here it can be seen that a new identifier was created (m7) and the identifiers bound in transition instance $t1_4$ have been removed.

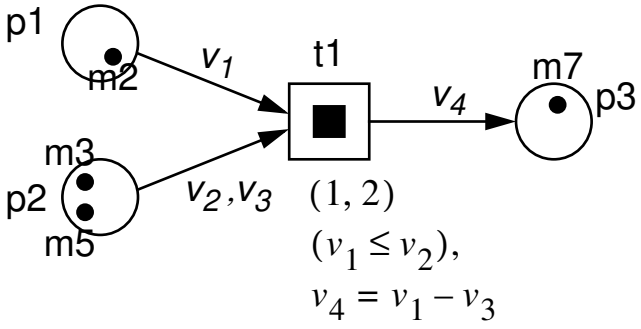

FIGURE 40. A high level time Petri net after the occurrence of transition instance $t1_4$ which is assumed to have occurred at time t = 1.5.

Table 5 shows a mapping of global identifiers to objects. This mapping is represented by the function *id*. Table 6 shows the only remaining enabled transition instance with its binding of global identifiers to variables and earliest and latest dynamic firing times. This remaining transition instance is obtained by removing all transition instances which also had a binding involving one or more of the same identifiers used in the binding of transi-

tion instance $t1_4$. Note that the transition instance $t1_7$ will fire in the interval *(0, 0.5)*.

Identifier	**Object**
m2	2
m3	1
m5	3
m7	1

TABLE 5. The mapping of global identifiers to objects for figure 40

$\mathbf{t1_i}$	$\mathbf{v_1}$	$\mathbf{v_2}$	$\mathbf{v_3}$	$\bar{e}$	$\bar{l}$
7	m2 (2)	m5 (3)	m3 (1)	0	0.5

TABLE 6. Valid bindings and dynamic firing intervals for all transition instances of t1 at time = 1.5 for figure 40

Remarks on High Level Time Petri Nets

Experience using high level time Petri nets has shown that it is sufficiently expressive to *easily* describe complex details of realistic systems. The objective of this kernel language was to define the smallest required set of extensions to Petri nets. Analysis techniques for this class of Petri nets is not the subject of this monograph however the following observations can be made. It is possible to simply transform high level time Petri nets to time Petri nets, enabling an analysis of time related behaviour to be carried out [19], [25]. High level time Petri nets can also be transformed into place-transition nets by simply ignoring net inscriptions and token data types. This enables *approximate* analysis of the high level time Petri net by analysing the corresponding place-transition net. For example, if a reachability analysis of the place-transition net shows that an undesirable state (token marking) can not be reached, then the high level time Petri net will also never enter that state. All sequences of transition occurrences in the high level time Petri net are present in the place-transition net but the contrary is not true. If the place-transition net can enter an undesirable state then the high level time Petri net may also be able to enter this state. This can be prevented by the value of tokens, transition guards or the time constraints of the

net. Standard analysis techniques such as place and transition invariants on the underlying place-transition net [79] also give meaningful results. Occurrence graph based analysis for high level Petri nets is currently an area of intense research [1], [70] with results showing that it is possible to detect dead and live locks, siphons, traps etc. for nets of modest size.

5.2 Combining Object Oriented Concepts and Petri Nets

Object oriented can be interpreted in many different ways. To some it is another name for *abstract data type* where data and functions are encapsulated into objects (modules, packages etc.) to enhance modularity and information hiding. Others regard *object oriented* as a way to share code and to organise a large system. Individual programs and their data are organised into a tree structure and objects inherit all attributes of higher level objects. This type of inheritance makes it possible to share code and data. During execution, the search for an attribute begins somewhere in the hierarchy and proceeds to the top. The first attribute found is the one that is used, therefore an attribute at a lower level hides those defined at higher levels and reflects a specialisation of the attribute.

Object oriented programs usually operate with hundreds or thousands of objects. Many of these objects have common properties which can be described with *classes* (templates). Classes themselves may also be collections of objects but always define templates used for the creation of objects i.e. they specify the number, and optionally, type of attributes that an object has over its lifetime and the methods and interfaces used to manipulate it. The class concept is a common mechanism for the construction of object oriented systems.

In [21] and [107] object oriented programming languages based on prototypes are described which do not use a class concept to create new objects but instead use a copy and modification mechanism. This is very flexible but highly dynamic, in that a program is essentially self modifying, defining and creating new types from existing objects during execution.

For hardware / software codesign this flexibility can not be supported in so far as a piece of hardware can not copy itself. At most it can modify its behaviour, albeit in predefined ways. Also prototype based programming assumes the presence of a garbage collector implying that memory allocation and deallocation can not be implicit.

In standard object oriented languages like Smalltalk [59] and C++ [103] an object consists of a set of procedures (methods) and local variables. There are two types of procedures: direct (those defined in the object's class and perhaps overriding a procedure defined in a superclass) and inherited (defined in a superclass of the object's class). There is one thread of control which is started by executing a creation procedure of a single class. Smalltalk and C++ also allow the creation and referencing of global variables.

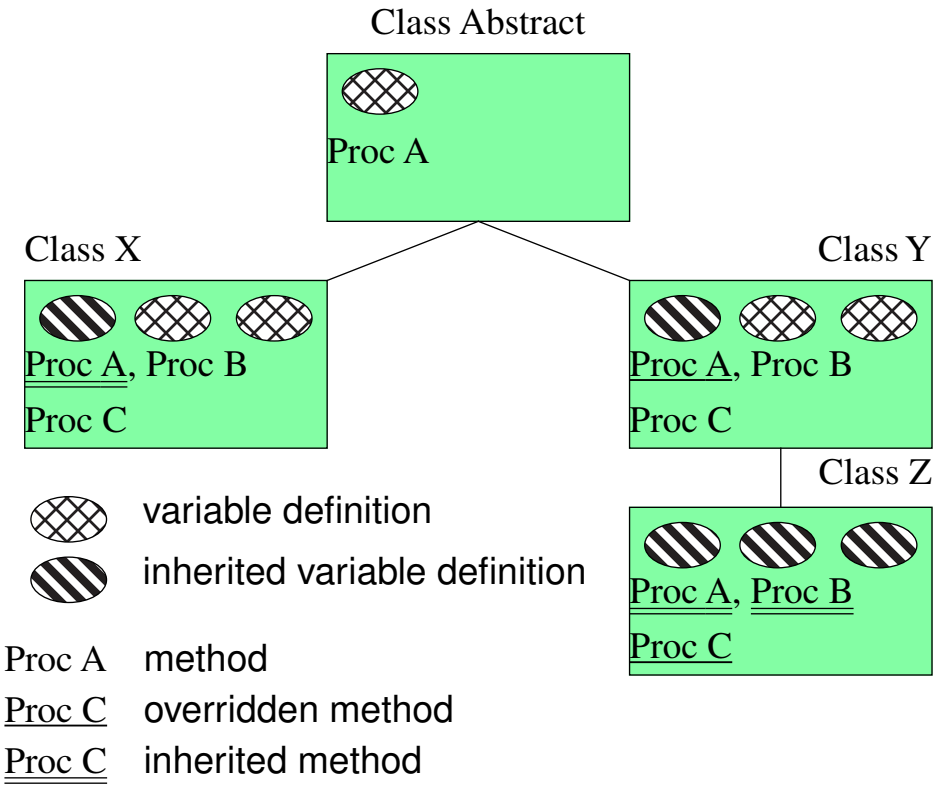

FIGURE 41. A typical class hierarchy showing inherited data definitions and inherited and overridden methods (procedures).

The execution of a sequential object oriented program can be thought of as a series of messages between objects as shown in figure 42.

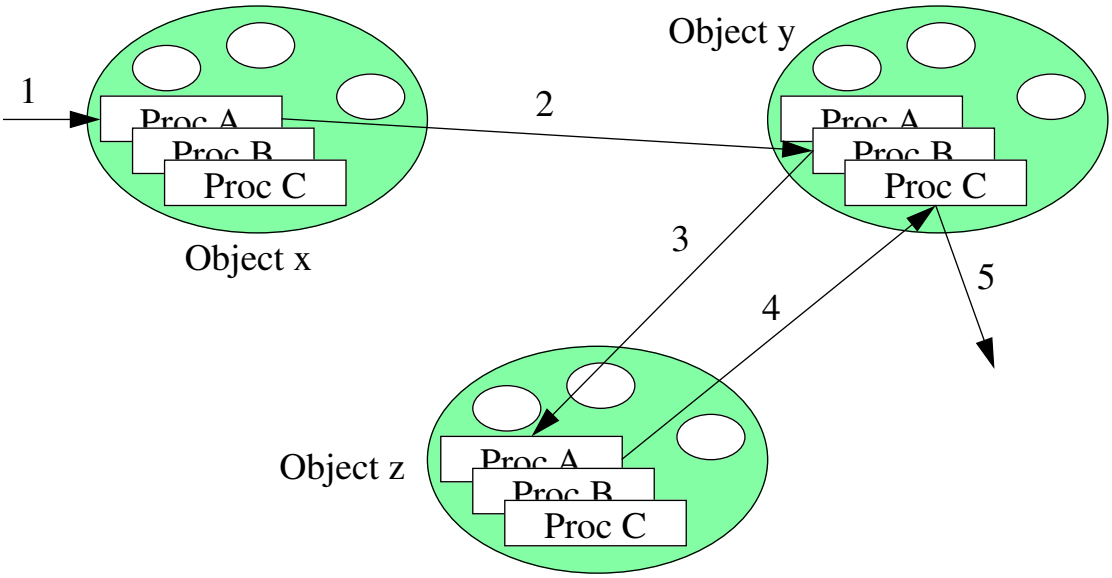

FIGURE 42. A sequential message sequence

The absence of both powerful structuring primitives and composition mechanisms in Petri nets is still a major weakness [69]. Various forms of graphical folding mechanisms [38], [93] have been proposed but only aid in information hiding and not in the composition of nets. Modularisation including composition proposals include those described in [17], [47] (which also presents module refinement techniques) and Hierarchical Coloured Petri Nets [70].

Recently there have been attempts made to combine the structuring techniques of object oriented programming with Petri nets. Allowing behaviour to be described with a Petri net enables complex concurrent behaviour to be modelled together with the data that it manipulates. In a high level Petri net data is modelled as tokens residing on places. Therefore classes can define functions, executed within transitions, and a Petri net where (token) data and behaviour are encapsulated. However as discussed above this is not enough, as it is also necessary to support code (net) reuse and sharing. This, in an

object oriented context, implies some form of inheritance and polymorphism.

5.2.1 Existing Object Oriented Petri Net Approaches

A number of proposals have been made to merge object oriented concepts with Petri nets. Some approaches have only addressed the definition of token types, preferring to retain a traditional Petri net style with tokens as passive data and the Petri net defining how data evolves. Other approaches support multiple levels of activity by allowing token types to be instances of Petri nets allowing complex systems such as operating systems and layered protocols to be modelled.

An early proposal to integrate Petri nets with object oriented concepts was PROT Nets [9]. PROT Nets can be decomposed into subnets with places as boundary elements. Token data can be manipulated using a general purpose programming language. PROT Nets do not support inheritance and have a strict token passing regime limiting each transition to have at most one input and output place with the same type (excluding the null type).

In [109] a two level model (the SimCon Object Model) is proposed specifically for information systems modelling. As Petri nets do not support a pure data oriented system view it includes a data model that comprises of simple objects and complex container objects - a structured set of simple objects accessed via a reference structure called *complexes*. Thus the SimCon Net model is a Petri net formalism where places hold complexes and transitions generate, delete, transfer and modify them as specified in an object algebra. This approach allows information modelling and the ability to define and verify constraints, however data and functions are not encapsulated together.

The coloured Petri net formalism [70] has also been integrated with the object oriented language Beta [24] as described in [35]. Beta is used to declare token types and to inscribe transitions. The Petri net itself defines the global control and is not object oriented.

Lakos and Keen have proposed LOOPN [81], [82] and LOOPN++ [80]. LOOPN net supports two separate class hierarchies - one for token data and the other for subnets. The derivation of complex subnets from simpler ones is supported by inheritance of subnets. It also provides global subnet access functions which allows access to the subnet state, i.e. dead, live etc. LOOPN++ improves upon this model by providing a single class hierarchy for tokens and subnets, allowing tokens to be not only data but also subnets. It also supports substitution places and synchronous interaction between subnets.

Lakos has further proposed Object Petri Nets [77], [78] which is a refinement of the concepts of LOOPN++. Object Petri nets have been proven to be equivalent to Hierarchical Coloured Petri Nets (HCPN) [70] and therefore benefit from the analysis techniques developed for HCPN. They extend HCPN's by integrating object oriented concepts with facilities to specify object interfaces as synchronous and asynchronous.

OBJSA [12] integrates abstract data types with the concurrency and synchronisation of Petri nets. In this model an object's life cycle is defined by a state machine. Objects execute concurrently and can be synchronised by transition fusion. OBJSA does not support inheritance but the derivative Language CLOWN [11] does, by the addition of method redefinition and attribute addition, and by requiring that the ST pre-order relationship be maintained between a class and its subclasses, thereby ensuring similarity of behaviour.

CO-OPN [28] is a formalism similar to OBJSA but extends the use of algebraic data types to communication structures. Here objects are defined by algebraic data types and the methods by transitions. Unlike OBJSA, CO-OPN objects may fire more than one transition concurrently. The firing of a transition requires only the internal state of an object whereas external transitions must be synchronised with the transitions of other objects. Objects can be parameterised but do not use inheritance.

5.3 Object Oriented Time Petri Nets

Most systems described in the previous section only consider token data to be object oriented. The approach chosen in this monograph is similar to that taken in LOOPN++ [80] where not only the token data is object oriented but also the Petri net itself. Conceptually every element is an object and can be used in every context. Hence token data are objects that may also contain a Petri net enabling multi level systems to be modelled e.g. an operating system controlling a number of processes. Two major differences to LOOPN++ are the addition of a concept of time and the concept of system instantiation, where a system is instantiated to represent a particular configuration in a step preceding the start of execution. For the types of systems targeted by the approach described in this monograph, time is a critical component and must be able to be expressed. Care has also been taken to ensure that suitably refined models can be implemented in hardware where the implementation constraints are most severe.

In the object oriented approach taken, every object is an instance of a particular class, which can be considered to be its type. An extension to the high level time Petri nets defined above is the addition of a type system, i.e. the values of tokens on places are constrained to be of a particular type. In this approach type is synonymous with class, therefore class refinement also implies type refinement. This an important mechanism in refining a model so that it can ultimately be efficiently implemented on a heterogeneous target architecture. For example, a filter can be described as a data flow graph transforming integer data to integer data. A refinement may involve defining the number of bits used to represent this data and how an overflow is handled (truncation, rounding or exception).

5.3.1 Class Definition

A class is a template for *class instances* or *components*. A class is a node in an inheritance tree whereby it may have at most one parent class i.e. single inheritance and $(0 \ldots n)$ subclasses i.e. classes for which it is the parent.

A class can be defined as a 7-tuple:

$Class = (Parent, HLTPN, Intf, Intf_{RW}, Arc, Comp, FUN)$ where:

- *Parent* can be empty indicating that it is the root class or another class. Classes form a *class hierarchy tree*;
- *HLTPN* is a high level time Petri net as defined before;
- *Intf* are directed interfaces graphically represented as ;
- $Intf_{RW}$ are undirected interfaces graphically represented as (read-write, write only and read only). These interfaces allow the state of one component to be inspected and perhaps changed by the occurrence of a transition in another component. Read-write interfaces allow the state of another component to be read and written, write only interfaces allow the state to be overwriten without being able to read the existing state, while a read only interface can only read the state of another component but may not modify it. These mechanisms can be also be implemented with directed interfaces;
- *Comp* represents *component holders*. A class can contain component holders that either refer to other classes or to (instantiation) functions. During class instantiation the presence of a component holder will either cause a component of the referred class or the component returned by executing the instantiation function to be inserted. Thus a system consists of a tree of components.

  Figure 43 shows a class definition of class *X* containing two class holders. One class holder refers to a class which will be instantiated when class *X* is instantiated. The other component holder holds a function call which will be executed, returning a component when class *X* is instantiated. The resulting component is also shown containing two *embedded components*, one a component of class *Y* and the other a component of class *Z*, returned by the instantiation function;

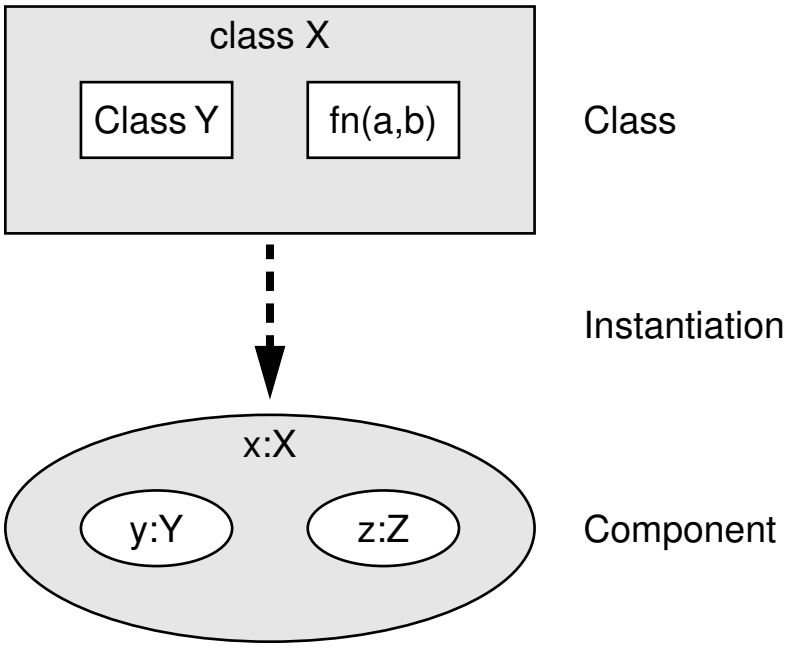

FIGURE 43. The instantiation process showing a class with two component holders, one referring to a class and the other to an instantiation function.

- $Arc \subseteq (T \times Intf) \cup (Intf \times P)$
  $\cup (T \times Intf_{RW}) \cup (Intf_{RW} \times T) \cup (P \times Intf_{RW}) \cup (Intf_{RW} \times P)$
  $\cup (Intf \times Intf) \cup (Intf_{RW} \times Intf_{RW})$

  where T and P are the set of all transitions and places of the *HLTPN*.

  Figure 44 shows graphically a possible connection of two components using directed interfaces. Defining arcs in this way ensures that components can be directly connected to one another with unambiguous semantics and without any intermediate ‘adapter’ or a fusion of elements. It is also possible to have more than one component input (to a place) connected to a single output. In this case all input places will receive tokens with the same values. Conversely having multiple (transition) outputs connected together driving an input is also possible;

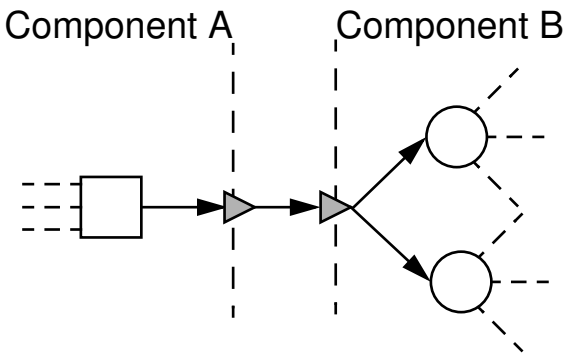

FIGURE 44. Two components interconnected with directed interfaces

- *FUN* is a set of functions where each function transforms input parameters (arguments) into a result. Functions returning a boolean value can be used as guards in transitions to possibly prevent a transition from being enabled. Functions are executed when a transition occurs to define the output token values.

5.3.2 Inheritance

Object orientation is more than simply encapsulating data and functions into objects. Inheritance is an important concept enabling code sharing, reuse, and together with polymorphism, is an important mechanism for refinement and abstraction. Inheritance is achieved by defining class hierarchies where subclasses inherit attributes from their parents. This section describes the semantics and rationale of inheritance on an attribute by attribute basis.

Functions

A function is associated with a class and maps input parameters to the function result. A function is defined by its name, input parameter types and result type called its *signature*. Therefore two functions are considered *equivalent* if their signatures are the same. A function is selected for execution either explicitly, or implicitly where a match function compares the function's signature to the actual parameter objects. Input parameter types

are compatible with the actual parameter objects if, for every actual parameter object its type is of the same class or a subclass of the corresponding input parameter type.

A function defined in a superclass is *overridden* when a subclass contains an equivalent function (with the same signature). Overridden functions may still be referenced.

Arcs

Arcs are always inherited from parent classes. Therefore the number of input or output arcs of an inherited element can never be less than that of the element defined in the parent class, i.e. arcs can only be added to an inherited element in a subclass. Arcs directly connected with transitions are inscribed with a name and optionally a weight function. The inscription *in[2]* is equivalent to the inscription in_1, in_2 but more flexible as a simple change in the weight function will act to effectively insert or delete names. Inherited arcs with a weight function must also have a weight function which may be changed but not deleted. If an arc has no weight function associated with it any inherited arc may not be subsequently associated with one. There is a fundamental difference in how a function refers to an input parameter if it is a n-tuple, i.e. (*in[2]*) or a single parameter. The arc inheritance rule ensures that subclasses can reuse functions defined in their parent class, and ensuring that it can also be effectively reused.

Places

In the high level time Petri nets used, places are typed with a class, ensuring that only tokens with values which are instances of the given class or subclass reside on the place. The choice of class depends on whether the place is inherited from the parent class or not. The type of inherited places can only be further refined, i.e. the place type class must be either the same or a subclass of the type class defined in the parent class. If it is not inherited it can be associated with any class.

Places can also constrain the maximum number of tokens that can reside on them, called place *capacity*. An inherited place can not have a capacity greater than that defined for the place in the parent class. This is important as capacity is directly related to the amount of memory required to implement the token storage. Ideally the underlying Petri net is bounded where the maximum number of tokens per place is defined, enabling transitions to occur without considering the capacity of their output places. However for reactive systems where events / data may enter the system with unforeseen distributions it is often necessary to define the behaviour of the system in an overload situation. Defining place capacities near the event inputs is one way of handling this situation.

Transitions

Inherited transitions are not constrained in their firing intervals, guards or actions. It is not desirable to constrain firing intervals as a subclass may implement an extended functionality with modified timing. However constraining the time behaviour of components as a whole may be appropriate and constitute a valid refinement.

Guard functions are boolean functions on the input token data providing additional enabling criteria. No inheritance rule is necessary for guards as the type of input places may be refined and additional inputs inserted.

Similarly, actions are functions on the input token data and define the output token data values. Again the number of inputs and outputs may increase and existing input and output places may have had their type refined, therefore no inheritance rule is necessary for actions.

Component Holders

Class definitions may contain references to other classes or function calls called *component holders*. A class defines a hierarchy of Petri net components (place, transitions and arcs), interfaces and *embedded components* defined by *component holders*. Embedded components are defined with

component holders either directly, via the class, or by an instantiation function. Direct class references may be refined in a similar manner to the refinement of place types, i.e. the class reference may be the same or a subclass of the embedded component class defined in the parent class.

Similarly embedded components may be defined with component holders having instantiation function calls. Here the result type must be of the same class or a subclass of the result class of the function defined in the parent class. Instantiation functions are used to select a component of a particular class and to initialise its state (initial token distribution) during the instantiation process. This could be for example assigning a unique node identifier for a system of equivalent communication nodes or determining whether a node acts as a slave or master in a distributed system.

5.3.3 Example

Figure 45 shows part of a class hierarchy used below as an example to show how a set of classes can be defined to implement a simple system.

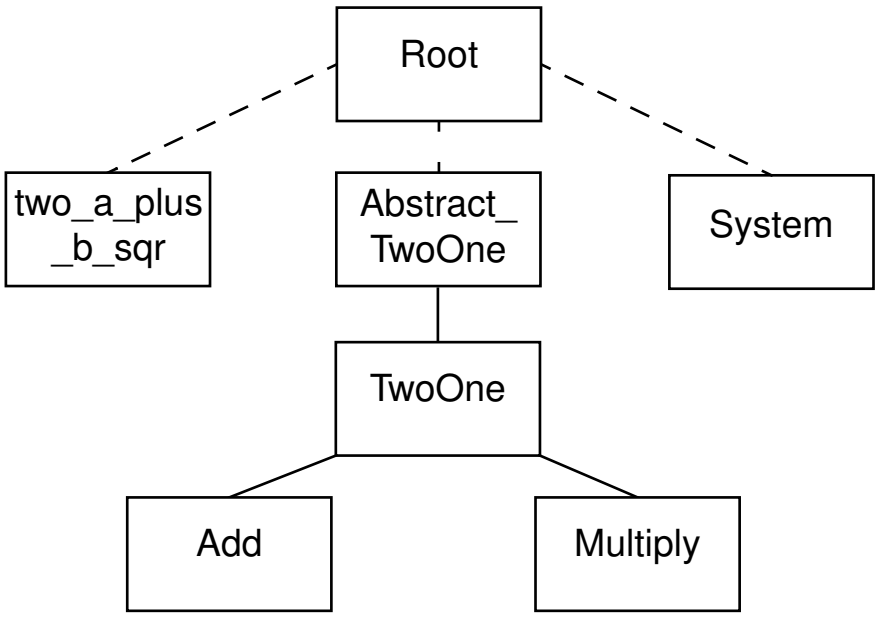

FIGURE 45. A class hierarchy tree. The dashed lines indicate an indirect (having intermediate classes) parent-child class relationship, the solid lines a direct relationship.

Figure 46 shows the graphical representation of the class *Abstract_TwoOne* having two inputs and a single output. The elements left, right and out are directed interfaces. The places *lP* and *rP* are annotated with a type, in this case *Number*. The type is used to ensure that only token values which are instances of class *Number* or a subclass of *Number* can reside on these places. Polymorphism is supported by dynamically selecting functions to be executed based on the type of the tokens involved.

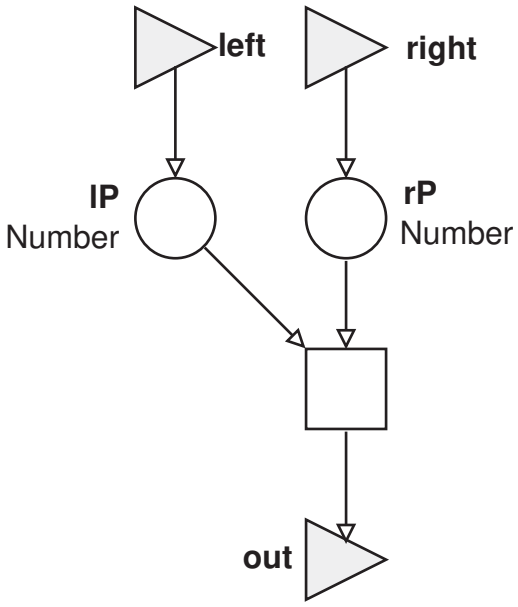

FIGURE 46. Graphical representation of Class 'Abstract_TwoOne'

Associated with this class is the following function called from within the only transition. As the class is marked as being *abstract* it can never be instantiated, but defines a function that must be overridden in its subclasses.

```
FUNCTION fn (a, b : Number) : Number
"an abstract function"
{
reportError('should never execute')
}
```

Classes are arranged in a class hierarchy. Subclasses inherit attributes from their parent. Figure 47 below shows the class *TwoOne* which is a subclass of

the class *Abstract_TwoOne* (the darker shading of elements indicate that these objects are inherited). The class has been extended with an undirected read - write interface and arc (both with a lighter shading). As a result of this extension the transition is also extended. Therefore any token value of a connected place can be used, in addition to the existing inputs, to define both the existing output and the new output connected via the undirected interface. Also of note is the refinement of the place type. Both places labelled *lP* and *rP* will only accept tokens of type *Real* or a subclass thereof. The places inherited from the parent class were typed with the *Number*. As *Real* is a subclass of *Number* this is a valid refinement.

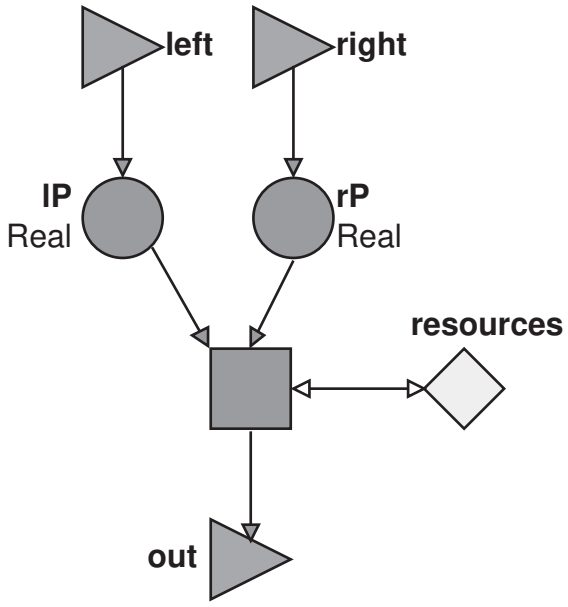

FIGURE 47. Graphical representation of Class 'TwoOne' showing inheritance and extension

Figure 48 below shows a concrete (intended to be instantiated) class *Add* - a subclass of *TwoOne*. In *Add* the function *fn* has been redefined to actually implement the required function.

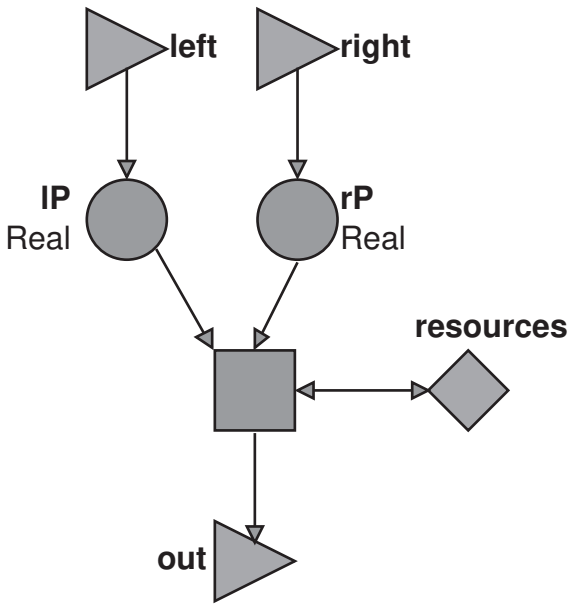

FIGURE 48. Graphical representation of Class 'Add'

```
FUNCTION fn (a, b : Real) : Real
"an add function"
{
RESULT := a + b
}
```

Similarly the class *Multiply* has been defined as a subclass of *TwoOne* where its function *fn* implements a multiply operation.

Classes can also contain component holders, which are references to classes or function calls, in addition to functions, interfaces and Petri net elements. Figure 49 shows the class *two_a_plus_b_sqr.* This class contains two references to the class *add* and a single reference to the class *multiply.* Furthermore interfaces have been defined to enter the input token data and to obtain the result. Connections between interfaces are used to compose the required behaviour from the components of classes *Add* and *Multiply.*

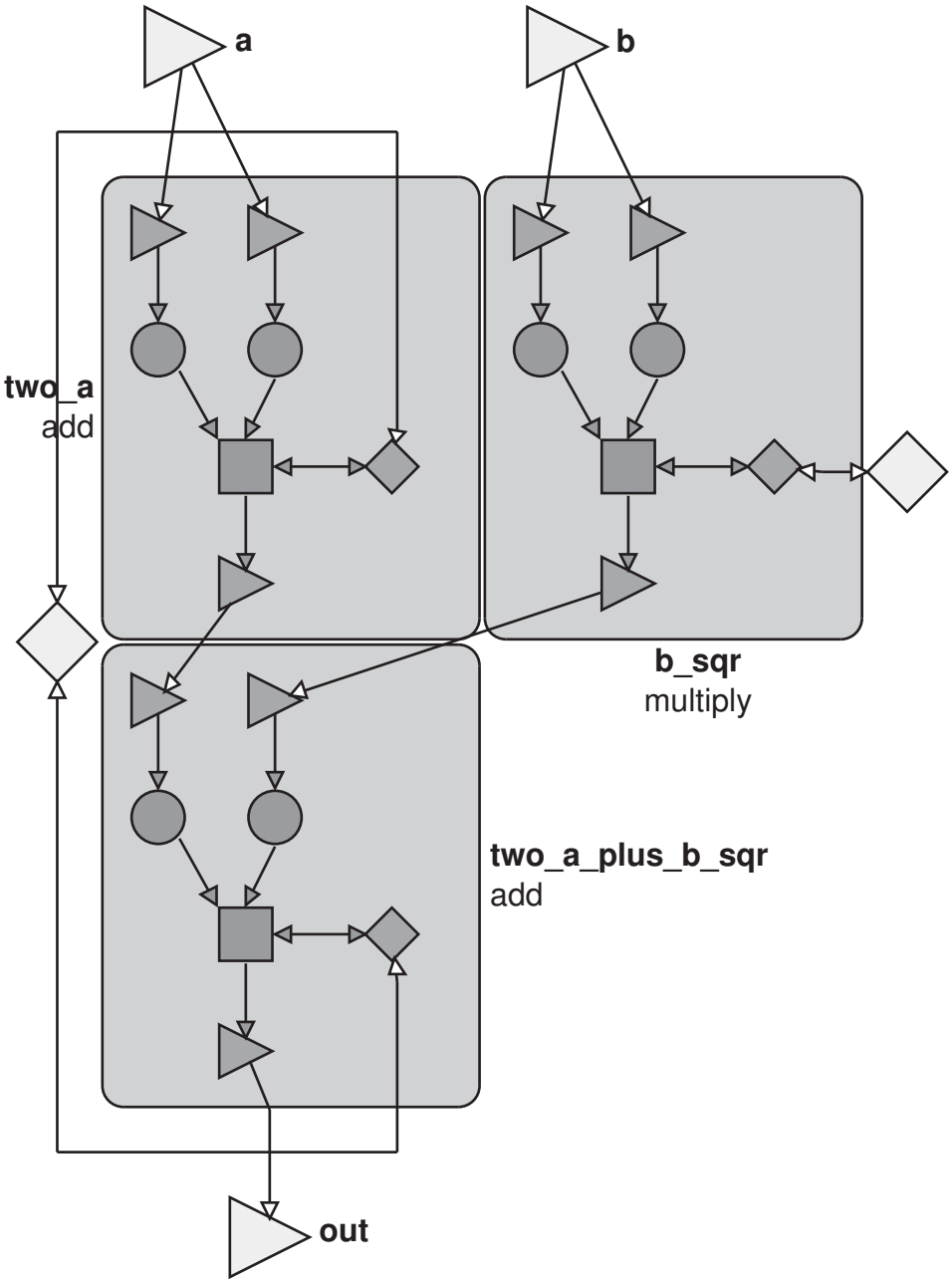

FIGURE 49. Graphical representation of compound Class 'two_a_plus_b_sqr'

Figure 50 shows an instance of the class *System* in which additional Petri net components are used to provide inputs to and store outputs from the component *two_a_plus_b_sqr*. This class defines a test environment for

two_a_plus_b_sqr. Instances or components have, in addition to the static structure defined by its class, a state defined by the token marking and dynamic firing intervals for each transition instance. Upon instantiation a component has an initial state.

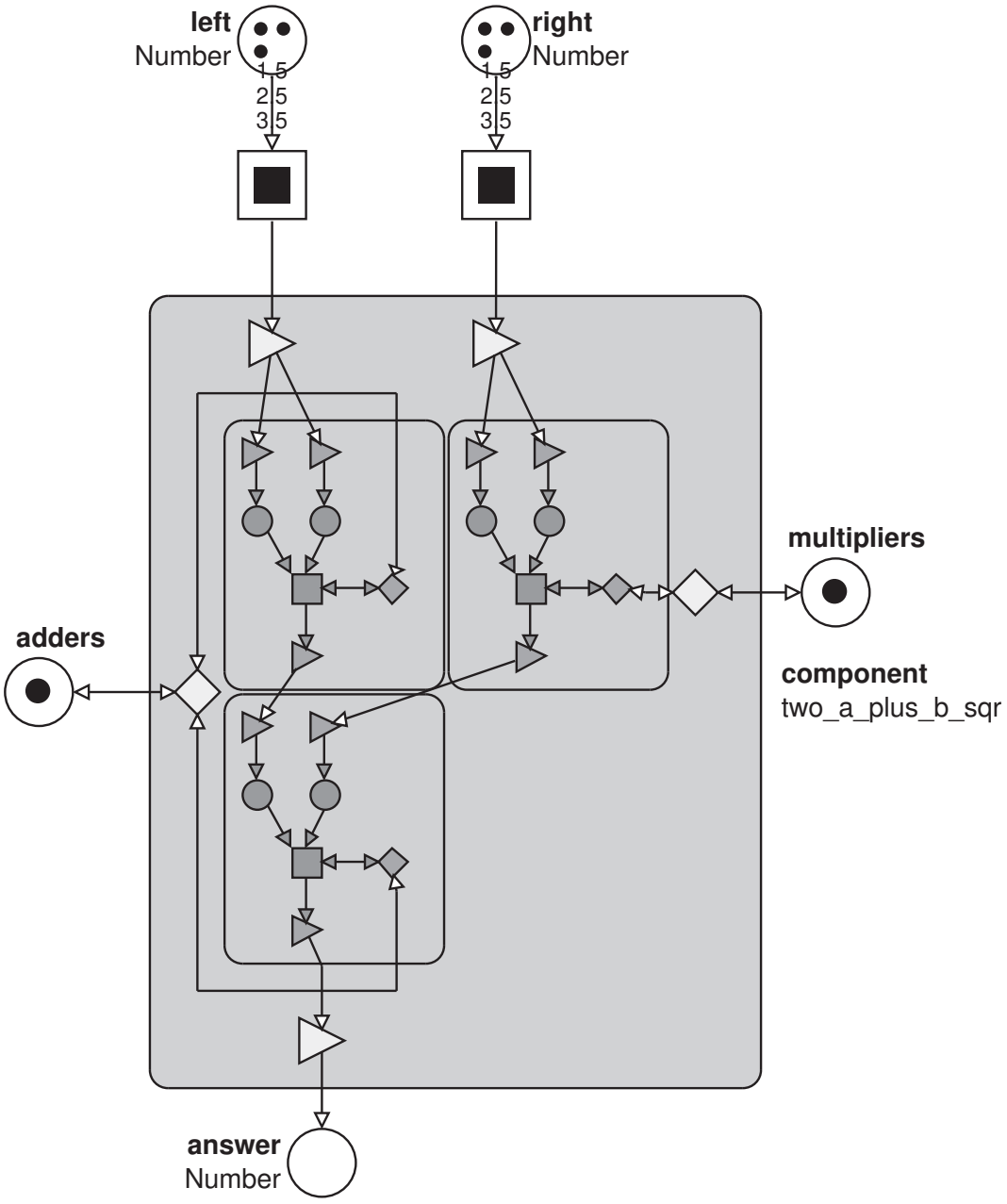

FIGURE 50. Graphical representation of class 'System'

5.3.4 Components

In this monograph class instance is synonymous with component. Figure 51 shows the instantiation hierarchy of class *System*. This can be compared with figure 45 on page 106 which shows the class hierarchy of the classes instantiated in *System*. In figure 51 all leaf nodes are Petri net elements as Petri nets define the semantics of these components. The names are used to refer to the individual components and are unique within a class and its instances.

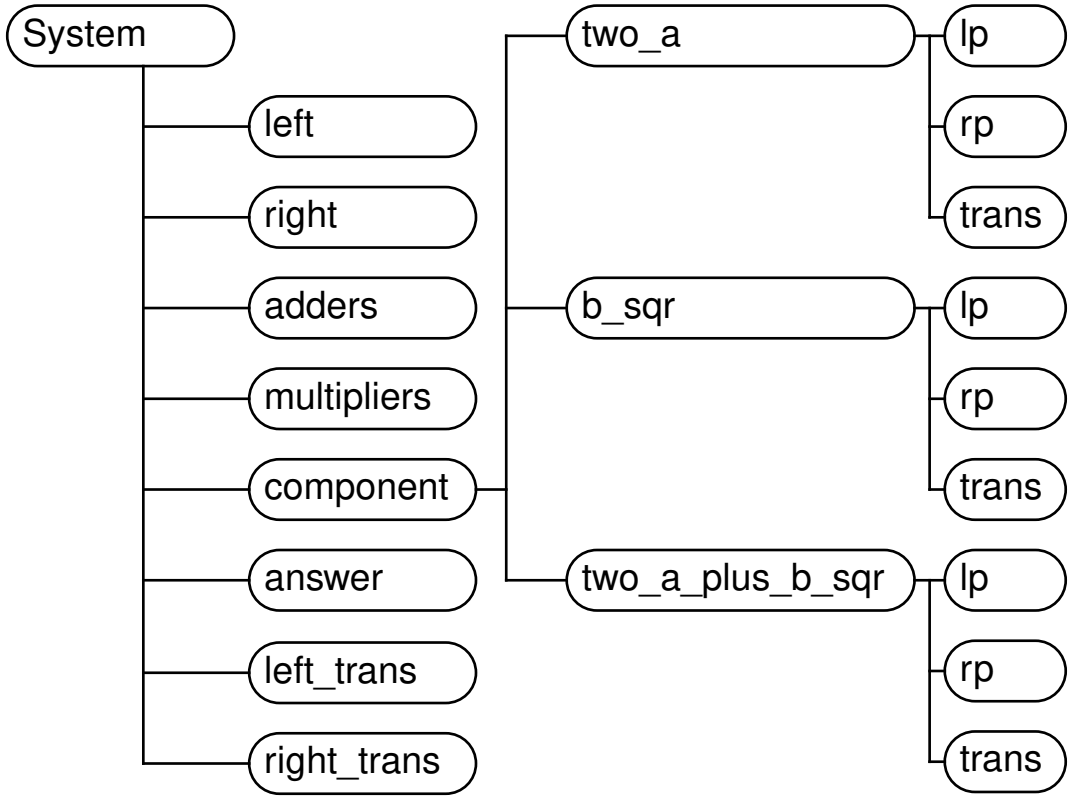

FIGURE 51. Instantiation tree of 'System' (without interfaces and arcs)

Components named *two_a* and *two_a_plus_b_sqr* are instances of the same class *Add*, i.e. only the state of the two components is unique. Thus these two components can share the same code (Petri net). Component *b_sqr* is an instance of the class *Multiply*. Classes *Multiply* and *Add* share the same abstract class and have a number of embedded components in common. In

this case only the function *fn* is unique. Subcomponents inherited without change can reuse the code (Petri net).

5.3.5 Configuration

Design exploration is an important requirement for embedded system design. This activity is supported by allowing class component holders to contain function calls or direct references to a class. When a class is instantiated the presence of a component holder will either cause a component of the referred class or the component returned by executing the instantiation function to be inserted. Instantiation function calls may be parameterised with objects defined in higher levels of the (incomplete) component hierarchy tree. In this way it is possible to define an object or function which can be used as a parameter by instantiation function. The result of the function execution can then be dependent on this parameter object.

The upper rectangle in figure 52 represents class *System* in which the three component holders contain function calls. Each function call requires a boolean object as parameter, which in this case is provided by executing a simple function *fast*. The upper elliptical object represents a class instantiation (component) of the class with the *fast* function returning *true* while the lower ellipse is the result of the *fast* function returning *false*. Note that the embedded components *fast_input* and *slow_input* may have completely different component hierarchies.

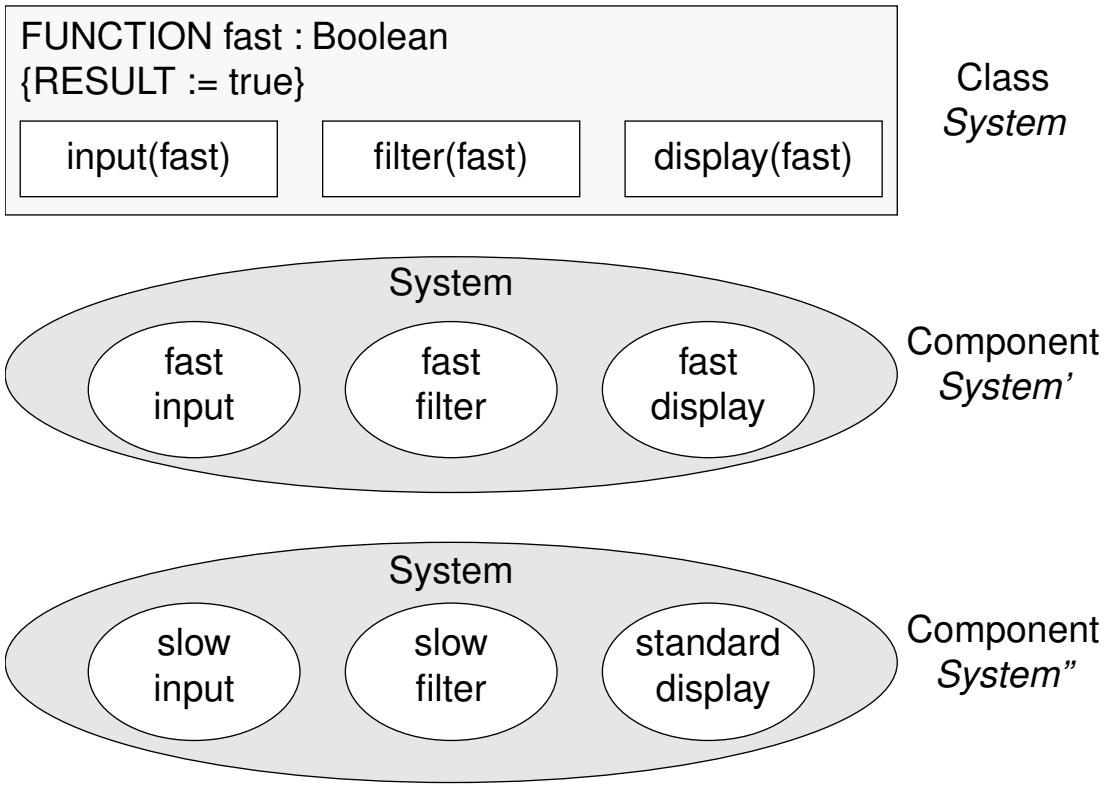

FIGURE 52. Two possible system configurations obtained by changing the result of the function *fast* used as a parameter in the instantiation function calls.

5.4 Summary

The kernel language provides facilities for encapsulating behaviour and state within components described with classes. Classes define how inheritance properties are propagated and provide mechanisms for configuration, reuse and sharing. A system is a composition of communicating components at every level of abstraction. Refinement is supported by substituting less refined components with components containing an increasing amount of detail. The kernel language is novel in that it is based on a formal model (Petri nets); it is homogeneous (does not rely on an external implementation language) and it is general enough to support other more specialised formalisms. This is the subject of the following chapter.

CHAPTER 6 *Support for Other Formalisms*

In the previous chapter a kernel language is described which is suitable for modelling real time systems. However many designers will choose to adopt other methods and notations depending on their working environment application domain and experience. Allowing systems to be modelled with already familiar formalisms reduces the risks, costs and difficulties of introducing new methods and avoids the reticence of many engineers to undergo retraining! In addition, the nature of many real time systems is such that no single formalism is suitable for modelling all aspects of these systems.

One goal of the approach presented in this monograph is to allow designers the flexibility of choosing the most appropriate set of formalisms for the particular system to be modelled. At the same time it was acknowledged that a formal approach was needed. In particular, care needs to be taken when describing how these formalisms are combined within a single system model. Many formalisms have well known implementation transformations and analysis algorithms and it is very desirable that these continue to be applicable in a heterogeneous modelling environment. The underlying kernel language provides the formal base upon which other formalisms are built, where these are described in terms of the kernel language.

Another goal is the ability of users themselves to add support for other formalisms rather than relying on the tool developer to modify the tool. The approach chosen is based on graph grammars, in which the user creates a set of kernel components for each element of a formalism and a set of graph grammar productions describing how these components are assembled and

disassembled. As all components are instances of a class it follows that any formalism assembled from components is also fully object oriented. Thus all the benefits of code reuse, abstraction and refinement provided by the object oriented kernel language are also applicable to all derivative formalisms.

Figure 53 shows the model architecture used to implement the ideas presented in this monograph.

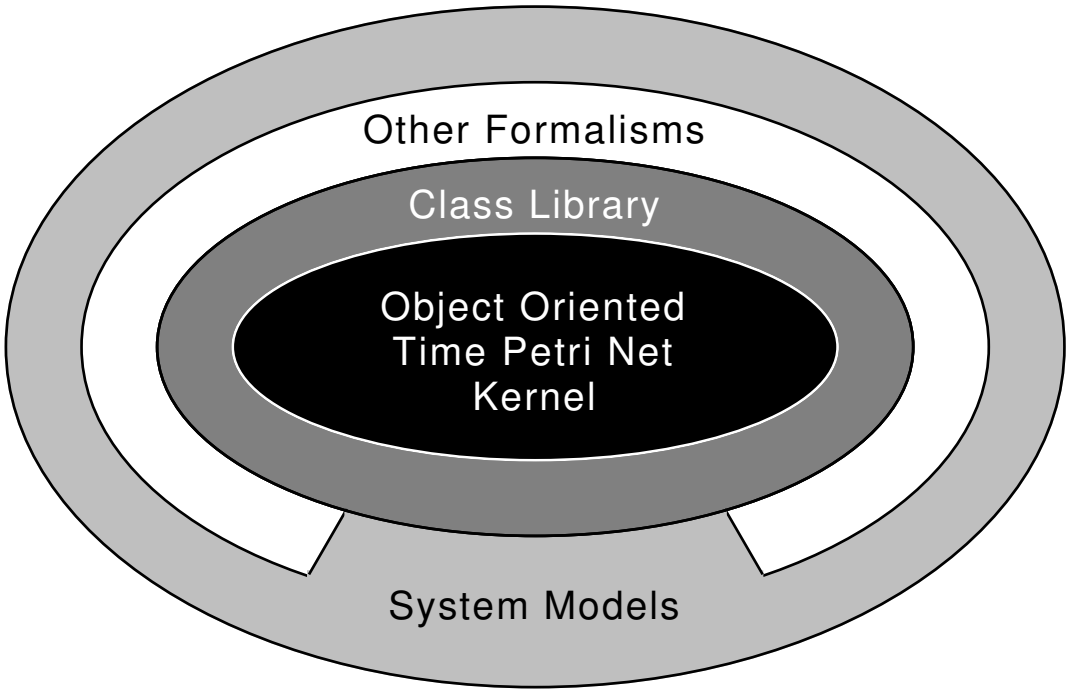

FIGURE 53. The Model Architecture

6.1 Other Approaches

A meta editing facility is present in a number of other systems that support multiple specification views. ARIES [71] and VIEWPOINTS [92] support multiple views of a specification while in PRISMA [91] multiple views of a system are captured with multiple formalisms. Consistency between formalisms is maintained through the coherence of the underlying logic theories. However not all properties are preserved between the various views of a system.

Cabernet [49], [96] offers meta editing based on a formal kernel language. This language, like the approach taken in this monograph, is based on high level time Petri nets. This formal language is also exploited to provide timing analysis [56]. Furthermore Cabernet allows the user to add basic tool functionality in addition to supporting other formalisms. The added functionality can be in the form of a debugger, tracing tool etc. and is defined with a subset of the kernel language, used to connect tool primitives which are executed to provide the added functionality required. A main difference to the approach taken in this monograph is the reliance on Petri net methods for abstractions that preserve temporal properties proven for specifications at a higher level of abstraction [48]. This approach, although rigorous, does not offer the flexibility of a general object oriented approach.

Other approaches allow systems to be defined with multiple formalisms. Ptolemy [72] is a unified framework that integrates specification, synthesis, simulation and evaluation. It allows subsystems to be modelled with different formalisms and has mechanisms to integrate them into a whole. An *event horizon* is defined for each domain over which streams of data are transferred to and from other domains. Domains in Ptolemy are formalised by the tool developer using the C++ language. This is in contrast to systems based on a simulation backplane [23] which impose a top level computation model through which all domains interact. Here the coupling between domains is adhoc, depending on the implementation of the participating tools.

6.2 Describing Formalisms with Graph Grammars

The approach taken in this monograph is based on graph grammars [61], [90]. Graph grammars are similar to string grammars in that they are both based on *productions* that describe how to derive elements belonging to the defined language. A production of a graph grammar describes how a subgraph can be replaced with another subgraph to obtain a new element of the graph language. Graph grammar productions specify the candidates for substitution, the subgraphs to insert and the connections between the inserted subgraphs and the rest of the graph.

Productions describe activities that need to be executed within a tool implementation. It is possible to interpret a graphical representation of a graph grammar production [96] or to use or define a language in which these productions are written. As graph grammar productions operate by inserting object oriented time Petri net components it seems appropriate to use the same language to describe the productions. Thus productions are normal functions operating in an object oriented environment, i.e. they are polymorphic and belong to the actual component classes into which a production will insert components. The resulting approach does not require the user to learn a new graphical or textual language but simply to use the same modelling language.

6.2.1 A Graph Grammar Notation

A notation to describe graph grammar productions is described in [90]. Here productions are written using a Y notation where the bottom left of the Y indicates the subgraph to be substituted by the application of the production, the bottom right indicates the new elements to be inserted into the graph and the top (often called the *embedding*) indicates the connections to be preserved. Dashed arcs represent conditions that must be fulfilled for the subgraph to be substituted.

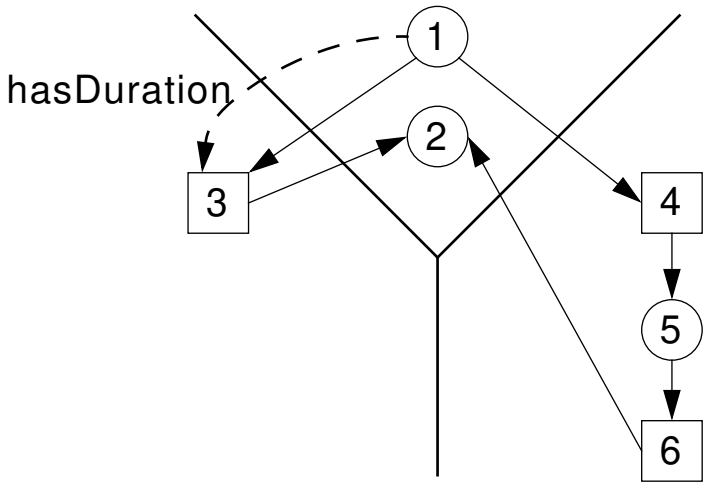

FIGURE 54. A graph grammar production

Figure 54 shows a graph grammar production that can be applied to model *timed* Petri nets with *time* Petri nets i.e. to nets having transitions with a firing duration. This can be compared with figure 36 on page 83. In this figure the types of the nodes are indicated by their shapes and labels. If more than one subgraph matches the graph in the bottom left segment then any one is chosen. In the case of an implementation in an interactive modelling environment, the user chooses the subgraph and the production to apply which is verified to be applicable to the chosen subgraph. The bottom right segment shows the replacement subgraph. Therefore in the figure 54 node 3 will be replaced by nodes 4, 5 and 6 and arcs from 4 to 5 and from 5 to 6. The connections to the top segment show that all inputs to node 3 will now be inputs to node 4 and all outputs from node 3 are now outputs to node 6. Figure 55 shows the application of this production to node *d*.

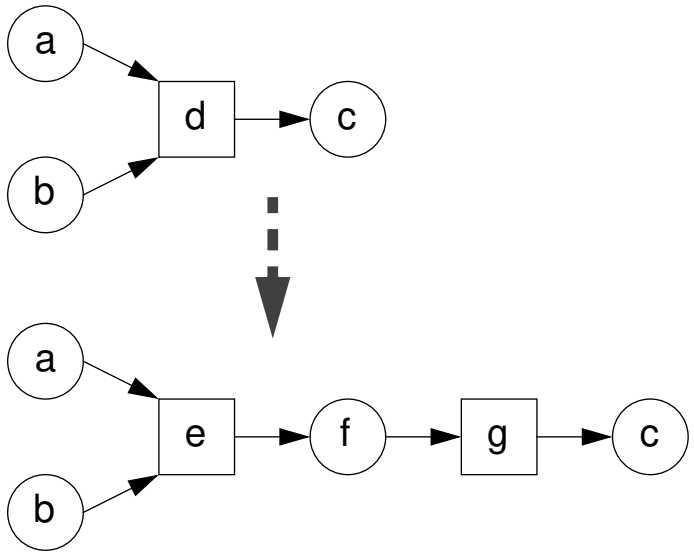

FIGURE 55. Result of the application of the production in figure 54

The graphical representation of a graph grammar production represents the following substitution algorithm:

1. all subgraphs that correspond to the subgraph in the bottom left of the Y diagram are formed, where the type of an element is given by its graphical shape and inscription;
2. dashed arcs are edges of the graph grammar and represent conditions that must be met by an equivalent subgraph or relationships between elements. Subgraphs that do not meet these conditions are ignored;
3. if more than one subgraph remains after the first two steps, one is either chosen randomly or via user interaction. Elements having labels appearing only in the bottom left segment are removed while elements with labels appearing only in the bottom right segment or displayed in bold are inserted. Elements with labels appearing in both the bottom left and bottom right segment are not changed, only being used for identification purposes;

4. the connections with the top segment of the Y diagram indicate how the new subgraph is connected to the rest of the graph. Existing connections to be deleted are indicated with arcs between the top segment and the bottom left segment while new connections are indicated by arcs between the top segment and the bottom right segment.

6.2.2 Describing a Formalism

The following steps illustrate how a new formalism is supported. These steps use the expressiveness of the kernel language in which a formalism's elements and editing rules are described:

- the syntax rules of a new formalism are represented as graph grammar productions. Productions define how a formalism's elements are edited to create models, e.g. the insertion of an element or the connection of two elements. These productions are called Syntax Graph Grammar (SynGG) productions [96] and are described completely in terms of the new formalism;
- an object oriented time Petri net component class is associated with every element in a formalism. These component classes may be already existing classes where an element of a formalism is directly represented by a kernel language component or may be new classes representing other behaviour. The resulting class hierarchy represents the elements of a formalism and should reflect that elements may have similar behaviour, which can be inherited. Care must also be taken in how communication between the proposed formalism and its environment is defined. Often a class will represent a part of a model described wholly by a single formalism and in which only classes representing the formalism's elements may be instantiated. These classes can be defined abstractly and, through subclassing, have the ability to be refined and reused. Thus all formalisms are object oriented and models may use the refinement techniques afforded by the object oriented approach;

- complementing every SynGG is a Semantic Graph Grammar (SemGG) production which describes the semantics of the syntax defined by the SynGG. These semantics are defined in terms of the kernel language (i.e. object oriented time Petri nets) and involve the component classes defined to represent the formalism's elements.

  The productions defined by the SemGG are more complex than those represented by the SynGG as they describe how kernel components are composed to represent a formalism's semantics including the syntax (editing rules) represented by the SynGG.

  Functions are then written representing the productions defined by the SemGG and are associated with a particular class in the object oriented environment, i.e. they are defined for the class in which a production can be executed. These functions represent the conditional arcs and the actual subgraph selection and replacement mechanisms;
- to complete the description of a formalism functions are created that define how a formalism's element is graphically represented (shape, weight etc.) In addition functions are created that define the interface to the editor. This allows a formalism specific editor to be generated in which actions cause previously defined productions to be executed.

The productions for Synchronous Data Flow (SDF) [84] illustrate the above steps. SDF graphs are widely used in digital signal processing systems because their representation exposes the natural parallelism of the modelled algorithm and imposes minimal constraints upon the execution order. Analysis techniques exist for SDF graphs and efficient implementations can also be generated. SDF graphs consist of nodes which represent operations and arcs which represent the data flow between nodes. Nodes can have any number of inputs and outputs. For an operation to be executed a predefined number of data needs to be present for every input arc and a predefined number of output data is generated for every output arc. SDF graphs also constrain the number of outputs connected to a particular input to 1.

A formalism can be contained within a component representing the subsystem model. This component has kernel interfaces through which the formalism can communicate with other parts of the model described by the kernel

language or other formalisms. Therefore the class that defines the component can also define the editing facilities (productions) that are appropriate for the formalism. In the case of SDF graphs it defines productions to insert SDF nodes and interfaces (used to communicate with other components) and to connect SDF nodes with one another. For each new formalism the following points need to be defined:

- the semantics of the individual elements including, connections of a formalism, in terms of components of the kernel language;
- productions to insert and delete these elements;
- productions for connecting elements;
- the appearance of the individual elements in the formalism;
- the transformations required to visualise the state and animation activity of individual elements.

In the case of SDF graphs, each node can be defined by a class which defines the number of inputs, outputs and function of the node. Figure 56 shows the productions that insert a component node into an SDF graph. A SDF node class is defined as a subclass of *Abstract_SDF_Node*. It defines the number of inputs and outputs and the function executed when sufficient data tokens are present on the inputs. The bottom left segment of the Y diagram is empty so a SDF node can be inserted in any context and, as the top segment is also empty, no connections need to be preserved.

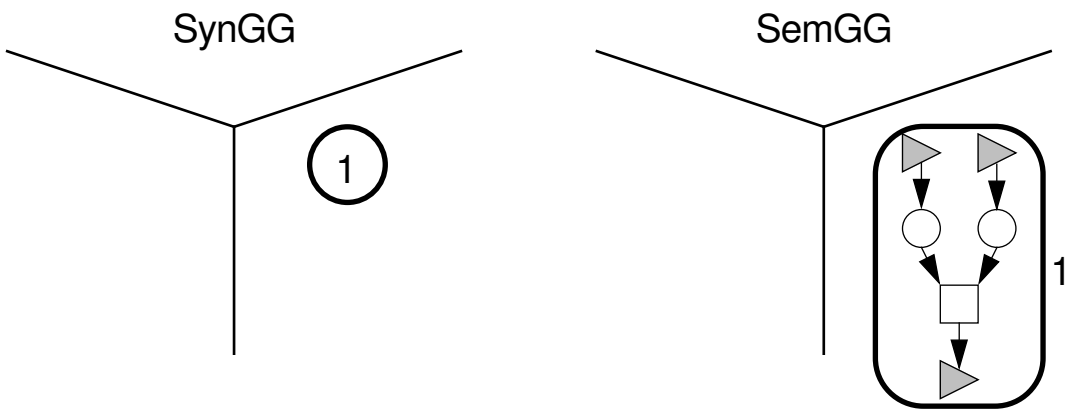

FIGURE 56. ***insert_node*** **syntax graph grammar (SynGG) and semantic graph grammar productions (SemGG) where inserted elements are bold**

In an interactive environment the elements defined in the bottom left segment of the Y diagram are explicitly identified by the user. Thus for the above production an editor button is defined representing the production that, when pressed, only prompts the user to position an icon representing the node in the editor window, as no elements need to be identified. Once the icon has been placed, a production, in this case the *insert_node* production, is executed which effectively inserts a node into the SDF graph. This production is described using the following function written in the kernel language:

```
FUNCTION insert_node (parent: Top_SDF): Root
"insert a node into myself if not successful return nil"
{RESULT := insertAnyComponentNamedInto (parent,
'Abstract_SDF_Node')}
```

The function *insertAnyComponentNamedInto()* presents a dialogue, containing all the subclasses of the class named *Abstract_SDF_Node*, from which a class can be selected. As all SDF nodes are subclasses of this abstract class any SDF node can be instantiated within the parent component. Also visible from the function header is the type of the parent component, i.e. SDF nodes can only be inserted into a parent of class or subclass of *Top_SDF*. Inserting a component always extends the class of *parent* by inserting a component holder referencing the selected class.

Figure 57 shows the productions used to insert an arc between two SDF nodes. It shows that the source and destination of the arc needed to be identified and that an input interface can only be connected with an output interface. The semantics of SDF graphs allow node inputs to be connected only with a single output. This is reflected in the conditional arc (not part of the formalism) in the SynGG labelled *canInsertArc*. However many unique inputs can be connected with the same output.

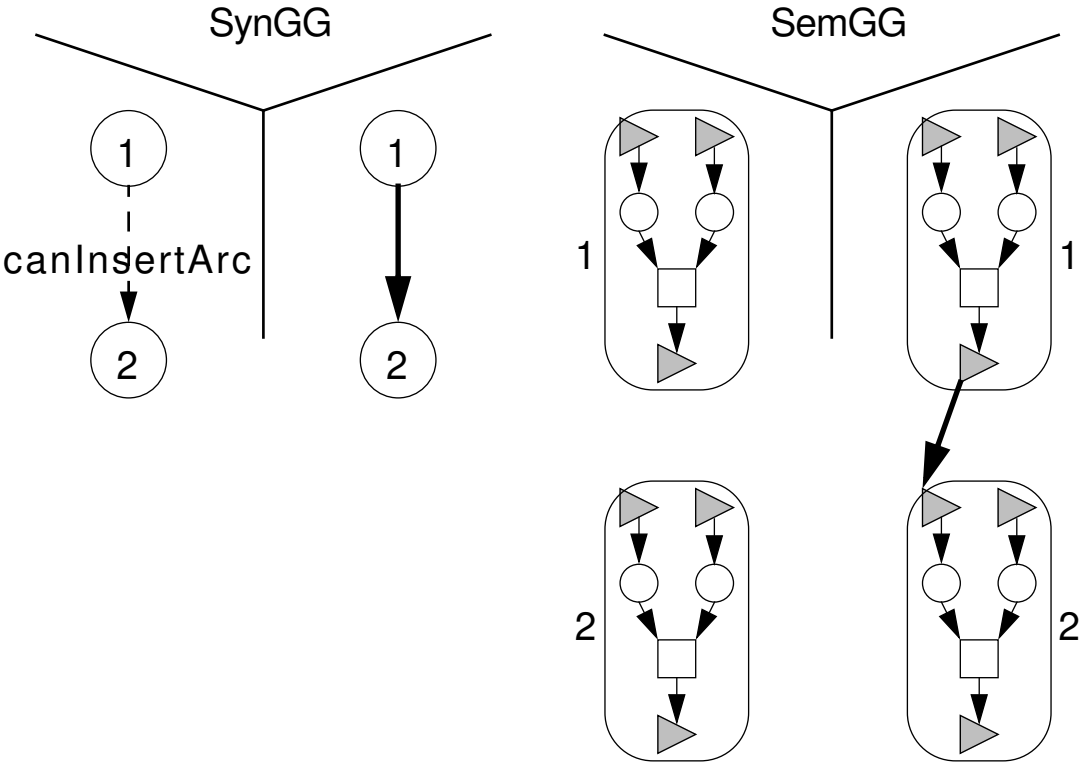

FIGURE 57. ***Insert_Arc*** **productions**

Two functions test the applicability of the selected elements to be used in the production. In the editor the user presses the arc button and is prompted for a node interface. Once the user has selected a node, a function is called to test the applicability of the initial selection. Note that this function has the same name as the conditional arc but in this case, as only one interface has been identified, has just two parameters - the parent and the selected interface.

```
FUNCTION canInsertArc(parent:Top_SDF;from:Root):String
"can I insert an arc into parent"
"an empty string means yes"
{
IF (className (from) = 'ToutPin') THEN
      "ToutPin is the class of a directed interface"
      {RESULT := ''}
ELSE
      {RESULT := 'must connect to an interface'}
}
```

The above function tests to see whether the class of the selected component *from* is an interface. If so it returns an empty string and the user is prompted for a destination node which is again tested with a function. This function has a signature that is almost identical to that of the previous function, only having an additional parameter for the second interface identified by the user.

```
FUNCTION canInsertArc(parent:Top_SDF;from,to:Root):String
"can I insert an arc into parent"
{RESULT := 'can only connect interfaces together'}
```

If *from* and *to* are not of type *ToutPin* then the above function will execute and return an error message.

```
FUNCTION canInsertArc(parent:Top_SDF;from,to:ToutPin):String
"can I insert a connector into parent"
{
IF (size(connectedNodes (to)) = 1) THEN
      {
      IF (name(from) = name(to)) &
         (name(PARENT(from)) = name(PARENT(to))) THEN
            {RESULT := 'can not connect the same node
interface with itself'}
      ELSE
            {RESULT := ''}
      }
ELSE
      {RESULT := 'destination is already connected'}
}
```

If the types of the two selected objects are interfaces then the above function will execute ensuring that the same interface on the same node has not been selected twice and that the *to* interface is not already connected. Finally, if the output and input have been identified, then the actual production to insert a connection is executed. The production function inserts an arc of class *SDF_Arc* between *from* and *to*.

```
FUNCTION insertArc(parent:Top_SDF; from,to:ToutPin):SDF_Arc
"insert a connector into myself if not successful return nil"
{RESULT := insertArcNamedInto (parent, 'SDF_Arc', from, to)}
```

Having defined the production functions, the appearance of each formalism element needs to be described including how state information and animation is presented. In the case of SDF no hierarchy is present therefore each node is represented by a circle in which only the interfaces are visible. Arcs keep their default representation and the colour of a node indicates whether it is active or not. Data is usually represented by displaying the contents of input node places on the corresponding input arcs of each node.

6.3 Example: StateCharts

The following example describes the productions used to implement support for the StateCharts formalism. The approach taken is informal and illustrates how a non trivial formalism can be supported using object oriented time Petri nets. This is in contrast to the informal non object oriented Petri net based description in [96] and the formal algebraic description in [86].

StateCharts [62] are a popular extension to standard finite state machines. They extend conventional state-transition diagrams with hierarchy, concurrency and communication. StateCharts are a more compact way of describing finite state machines and can be used in conjunction with other methods such as functional decomposition and data flow to provide a complete design methodology.

Figure 58 shows a typical StateChart in which states are represented by rounded boxes and transitions by directed arcs. In figure 58 state *A* represents a *parallel* or *AND* decomposition into two states *B* and *C*, i.e. being in state *A* also means being in state *B and C*. States *B* and *C* are further decomposed with a *sequential* or *OR* decomposition into states *D*, *E* and *F* and states *G* and *H* respectively. States *A*, *D* and *G* are initial states as indicated by the ●→ arc. Transitions are labelled with input and/or output events. The arrival of event *a* when the system is in its initial state will cause the transition from state *D* to state *E* to occur. Similarly if the system is in state *A* and event *k* arrives the transition from state *A* to state *J* will occur, thus also leaving states *B* and *C* and either state *D*, *E* or *F* and *G* or *H*.

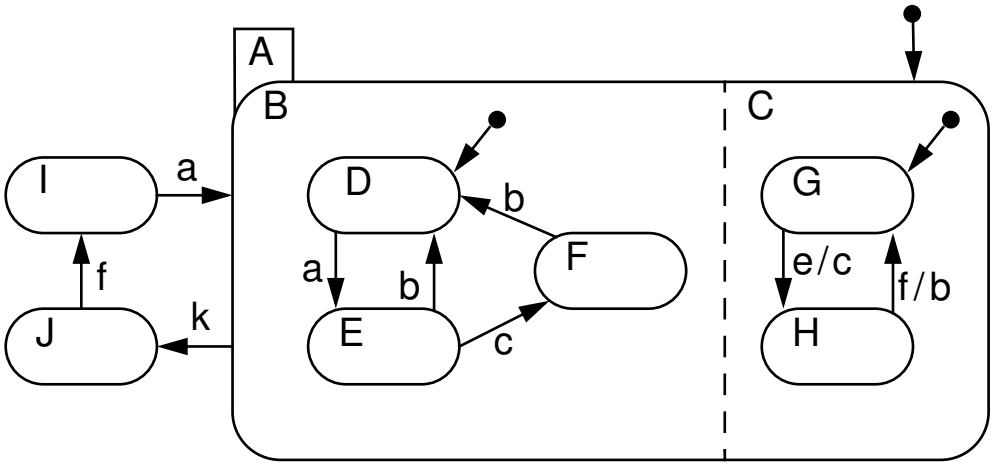

FIGURE 58. A typical StateChart

6.3.1 Modelling Finite State Machines with Petri Nets

Before discussing how StateCharts are supported it is helpful to consider how finite state machines (FSM) can be modelled with Petri nets. See also 'Finite State Machines' on page 26.

Finite state machines can be considered to be a subclass of Petri nets with the added constraint $|\bullet t| = |t\bullet| = 1$. States are modelled with a place and

state-transitions with a transition. The active state is marked with a token and in all the places that represent states the total number of tokens is always equal to 1.

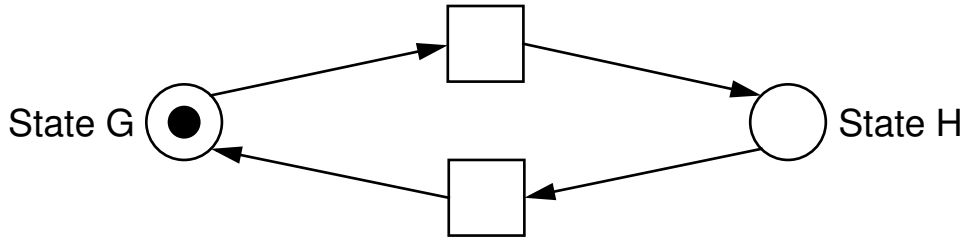

FIGURE 59. A Petri net representation of state *C* in figure 58.

Figure 59 shows a basic Petri net FSM representation. However the question of events also needs to be considered, i.e. a state-transition can occur when an event is present, optionally producing an output event. Figure 60 shows the Petri net extended with event input and output places. This model is sufficient if the state machine is completely specified, i.e. for every possible state, state-transitions exist that consume every possible event. However this requirement is often too severe and most FSM implementations assume that the system will remain in its present state if no explicit state-transition consumes an available event.

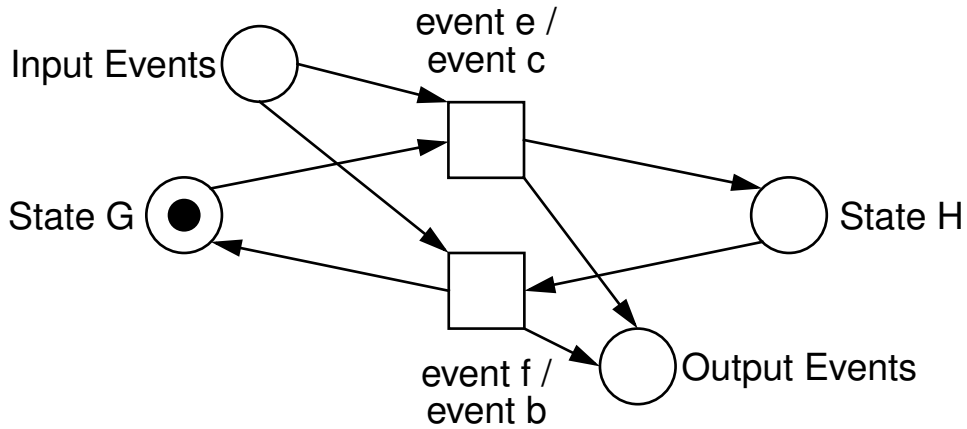

FIGURE 60. A Petri net finite state machine representation including input and output events.

Figure 61 shows the same Petri net FSM representation as in figure 60 but with additional transitions to consume events not explicitly defined in state-transitions.

As can be seen from figure 61 the Petri net representation now has little relationship to the equivalent system described with a FSM graphical representation. StateCharts are considerably more complex, making a direct Petri net implementation impractical.

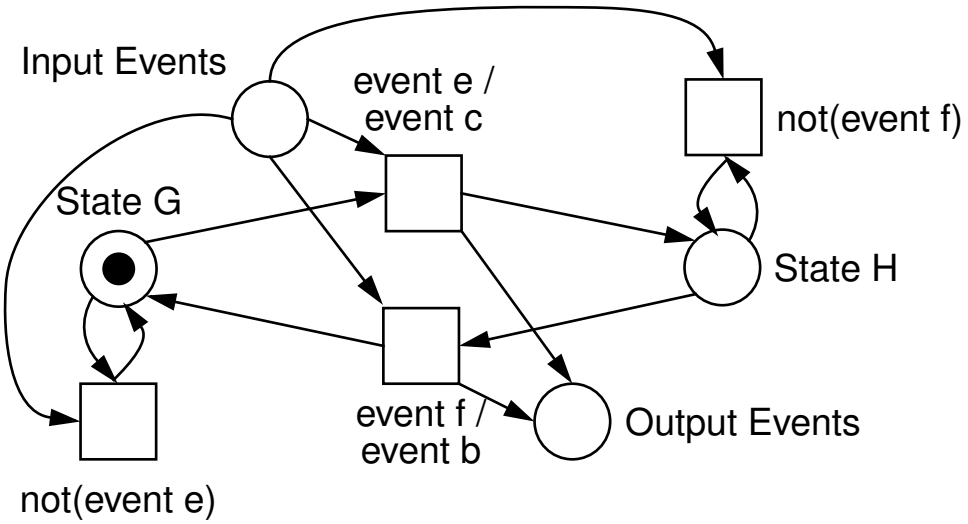

FIGURE 61. A Petri net finite state machine representation, including input and output events and provision for incomplete specification.

6.3.2 Supporting StateCharts

Although StateCharts are based upon standard finite state machines there are a number of semantic issues that still need to be addressed, as they have never been formalised. Harel states in [62] *The StateCharts formalism turns out to be quite a challenge when it comes to providing formal semantics, much more so than simple finite-state automata*. To this date there is no common standard which completely describes its semantics. For many environments which support StateCharts only the simulator defines the semantics.

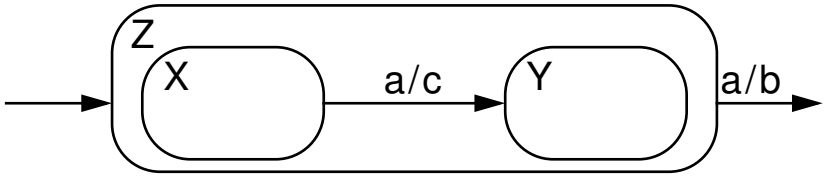

FIGURE 62. An ambiguous StateChart

Figure 62 shows a situation that can lead to ambiguous results between StateCharts implementations. If the system is in state *Z* and *X* and event *a* is present it is not clear which events will be produced, *b* or *b* and *c*? In this monograph internal events have priority over external events and internal events are visible in each subsequent 'step' where they are propagated from the top down to all active states. Therefore in figure 62 event *a* is seen by both states *Z* and *X* in the same delta step, resulting in the transition from *X* to *Y* occurring to produce *c* and the transition which leaves *Z* to produce *b*. This decision is reflected in how the implementation of the state event mechanism is implemented in the kernel language. Other behaviours can be implemented by suitably modifying this mechanism. Thus the kernel language is used to provide an unambiguous definition of the required semantics.

StateCharts [62] also define a number of constructs that significantly reduce the ability to reuse existing states, such as allowing a state-transition to reference any state in an *AND* decomposition and allowing state-transitions to cross hierarchy boundaries. In this example constructs that reduce modular-

ity are not supported, however it would be straight forward to introduce, for example, interface elements enabling state-transitions to cross state boundaries, similar to those described in [14].

StateCharts are adequate for modelling control aspects of embedded systems. However they do not have any capability for modelling data and functions. This has been addressed in a number of tools (StateMate, BetterState, SpeedChart etc.) through the use of either a general or special purpose programming language or by the addition of a data flow formalism. In the approach taken in this monograph a functional or data flow approach can be included, either as an integral part of StateCharts, effectively extending the formalism, or as a parallel formalism communicating with a StateCharts component over a defined interface via the StateCharts event mechanism.

6.3.3 Defining the Syntax Graph Grammar

The syntax rules of a new formalism are represented as graph grammar productions. These productions define how a formalism's elements are edited to create models, e.g. the insertion of an element or the connection of two elements.

The following production shows how a StateCharts component is embedded within another formalism. Figure 63 shows that a StateCharts component can only be inserted within a Petri net component. This is made explicit as only the semantics of coupling a StateChart to a Petri net have been considered. It may be desirable to extend this in the future if a requirement arises for other formalisms to encapsulate state machine behaviour.

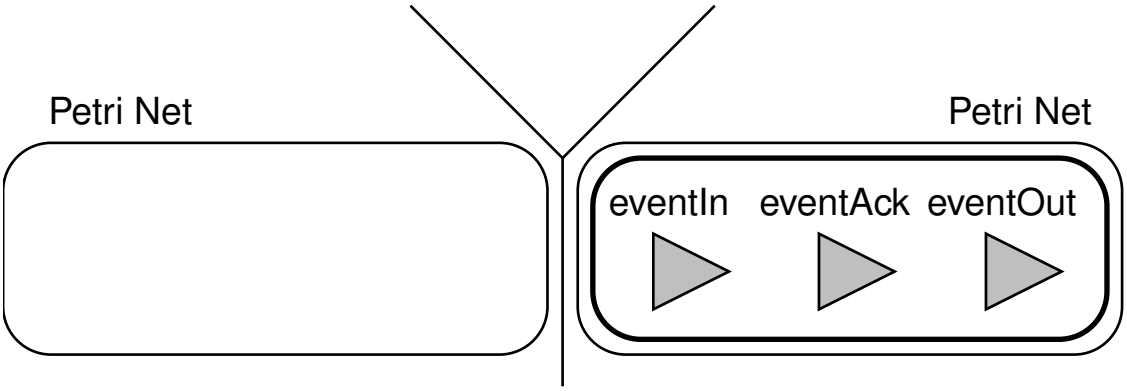

FIGURE 63. ***Create_State_Machine*** **syntax graph grammar. The shaded triangles represent interfaces through which events are communicated and synchronised.**

Once a StateCharts component has been inserted it is possible to insert an initial state. Always ensuring that a StateChart has only one top or initial state enables this state to be reused in other contexts, such as being embedded within another state. As a consequence productions for inserting state-transitions are only required within sequential states and not in the top level. The conditional arc in figure 64 ensures that only a single initial state can be inserted.

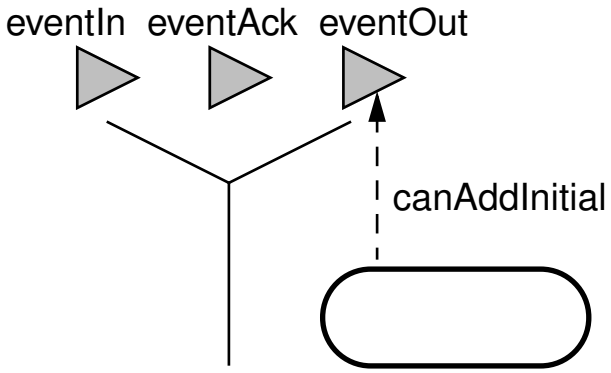

FIGURE 64. ***Add_Initial_State*** **syntax graph grammar**

Different productions are available depending on whether a *sequential* or *parallel state* is being edited. In both cases it is possible to insert both parallel and sequential states. However in *parallel states* no productions have

been defined to add state-transitions as this would reduce the possibility of reusing the involved states. Figure 65 shows the syntax graph grammar production for inserting a state into a sequential state.

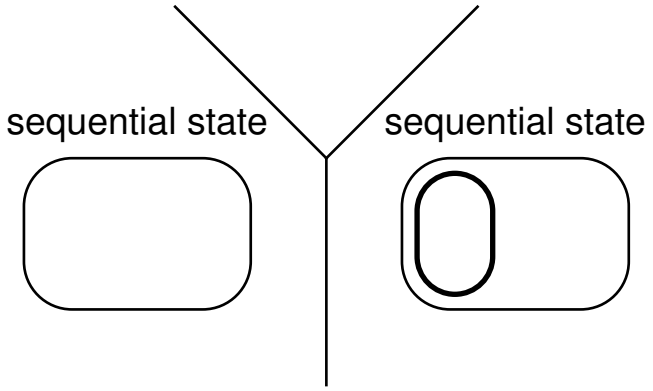

FIGURE 65. ***Insert_State_Into_Sequential*** **syntax graph grammar**

State-transitions can only be inserted within sequential states and may not cross hierarchical state boundaries. StateCharts, as defined by Harel, allow state boundaries to be crossed at a cost of reducing the modularity of the component states. It is proposed in [14] that interfaces be defined via which state-transitions are connected that cross hierarchical boundaries. This effectively means defining an extra element in the StateCharts formalism and inserting productions for connecting states to interfaces and vice versa.

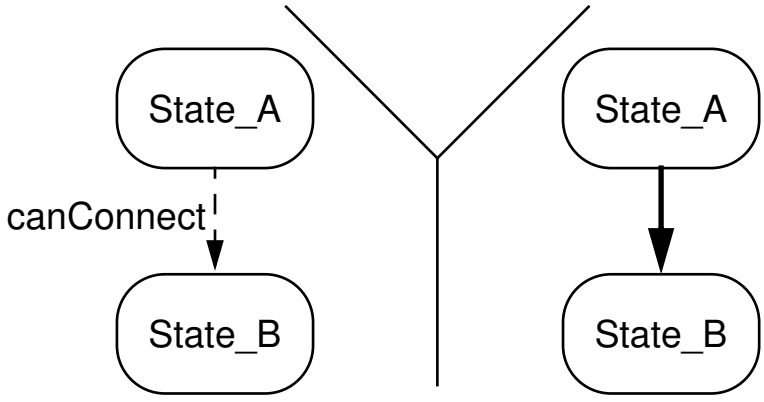

FIGURE 66. ***Insert_Transition*** **syntax graph grammar**

6.3.4 Defining the Class Hierarchy

Once the syntax of a formalism has been defined using syntax graph grammar (SynGG) productions kernel classes representing the elements of a formalism need to be defined. A class will use the kernel language to define the behaviour of each element and its interfaces, which can be partly derived from the SynGG productions. Mechanisms inherent to a formalism also need to be represented and may require the definition of specialised classes or the extension of classes representing elements of the formalism.

As an object oriented time Petri net component class is defined for every element in a formalism, they are modular and can be instantiated any number of times. In addition, unlike non object oriented approaches, subclasses can be created which refine and reuse the behaviour of a class. The definition of an appropriate class hierarchy is very important as only a *good* structure will allow all the benefits of object orientation to be utilised.

StateCharts Event Mechanism

The semantics of StateCharts can be viewed as being synchronous or asynchronous. In the synchronous view an external event causes the system to enter a defined state. It is assumed that intermediate internal events and transitions that occur due to internal events are used only to define this state. Thus the system can be considered to react to an external event by perhaps changing state in a single synchronous action. In the asynchronous view an external event may trigger a cascade of state-transitions (steps) and the system will process another external event only after all internal activity has ceased. Similarly to most implementations asynchronous semantics are assumed in the following example. StateCharts also define a broadcast mechanism for events (communication) where an event is visible in all active states. Due to this event a set of transitions may occur, optionally producing a set of internal events. These internal events are then only visible to all states in the next step. In order to implement event propagation in StateCharts a mechanism is needed for broadcasting the current event set and for collecting the generated events produced by state-transition occurrences.

StateCharts Class Hierarchy

Figure 67 shows part of the class hierarchy that is used to implement StateCharts. Subclasses of *Top_State_Chart* define a StateCharts component that can be embedded within other formalisms. It also defines interfaces over which events can be exchanged and synchronised and ensures that external events are only processed when no internal events have been produced.

The class *FSM_Trans* represents a state-transition and is marked as an abstract component class, i.e. when instantiated it automatically creates a subclass which is monomorphic (it can not be further subclassed) and is therefore a leaf node in the class hierarchy tree. This is used when an element represents a specific behaviour, which is obtained by inheriting the general behaviour from its superclass and refining one or more attributes. For example, all state-transitions respond to a specific set of events and optionally generate another set of events. These events can be regarded as a specialisation of a general class, representing a state-transition that responds to no events. The monomorphic class attribute will ensure that no further (inappropriate) refinement of a state-transition class is possible.

Abstract_Parallel and *Abstract_Sequential* are abstract classes which must be subclassed to be used. As StateCharts support the hierarchical composition of states, each such hierarchical state is represented by a class having fields which are again references to states. *Leaf* is one such subclass and is designated as being monomorphic. It is intended to be used for all states that will never be further refined, i.e. leaf states. As all hierarchical states are defined with classes, they can be further refined by subclassing where new states and state-transitions can be added and all inherited state-transitions can be modified. See 'Other Formalisms in an Object Oriented Environment' on page 143. Therefore a (sub) system can be modelled abstractly, refined and reused. These inheritance characteristics are provided, by default, by the kernel language.

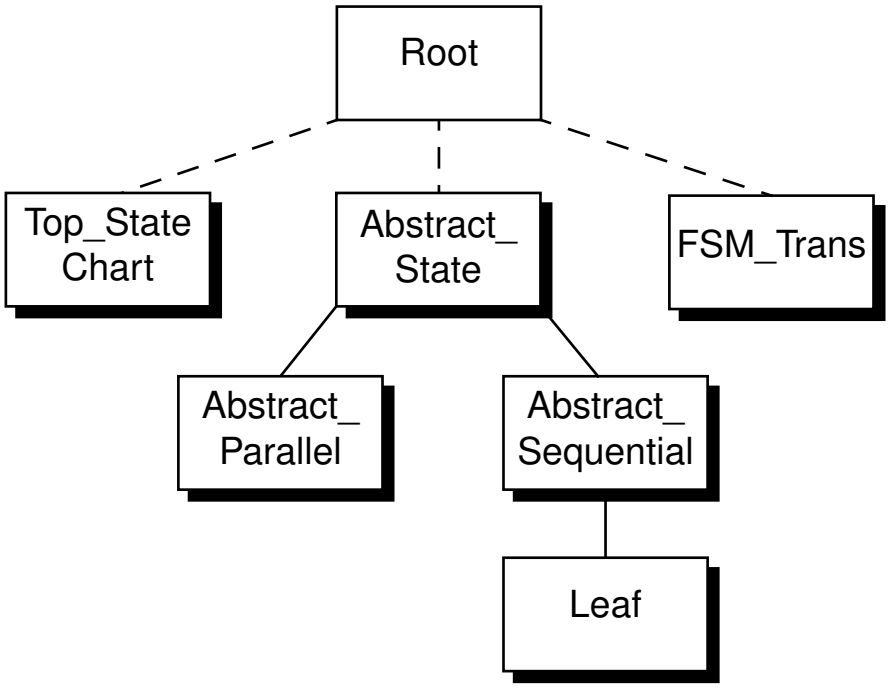

FIGURE 67. The StateCharts Class Hierarchy Tree

6.3.5 Defining the Semantic Graph Grammar

Complementing every SynGG is a Semantic Graph Grammar (SemGG) production which describes the semantics of the syntax defined by the SynGG. These semantics are defined in terms of the kernel language, i.e. object oriented time Petri nets and involve the component classes defined to represent the formalism's elements.

Functions are written representing the productions defined by the SemGG and are associated with a particular class in the object oriented environment, i.e. they are defined for the class in which a production can be executed. Functions represent the conditional arcs and the actual subgraph selection and replacement mechanisms.

Embedding and Event Mechanism

The following production shows how a StateCharts component is embedded within another formalism. Figure 63 shows that a StateCharts component can only be inserted within a Petri net component. This is made explicit as only the semantics of coupling a StateCharts component to a Petri net have been considered. It may be desirable to extend this in the future if a requirement arises for other formalisms to encapsulate state machine behaviour. In the semantic graph grammar production (figure 68) a net is introduced that distributes and collects events. The Petri net ensures that external events are only processed when no internal events have been produced. The transition *init_Complete* is used to make certain that the system is ready to accept its first event. In this example all internal events are also made externally available through an interface. Of note is the small net that ensures that an initial state will be defined.

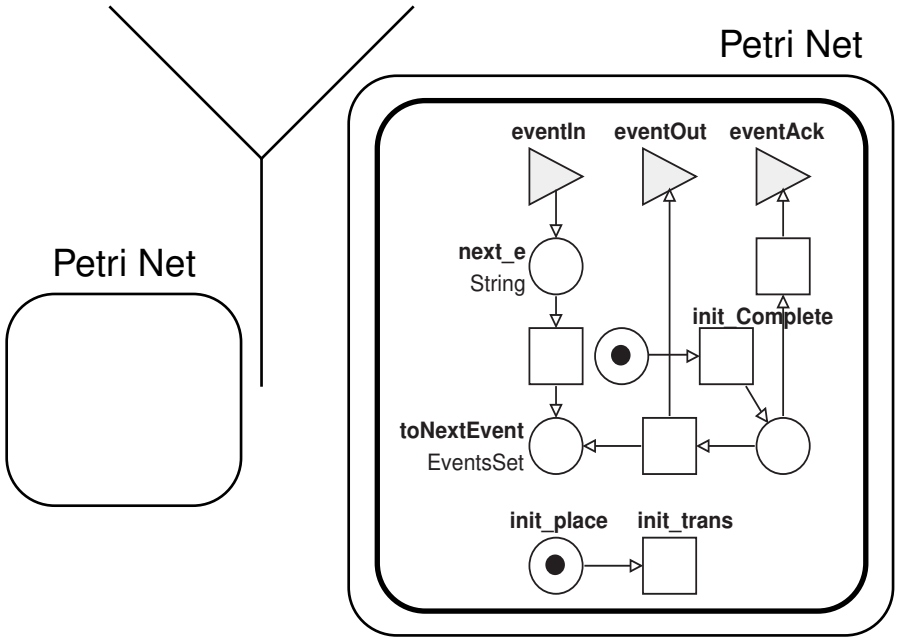

FIGURE 68. ***Create_State_Machine*** **semantic graph grammar. Note that the component is inserted within a Petri net component and will communicate with the Petri net through the interfaces.**

Inserting States

Once a StateCharts component has been inserted it is possible to insert an initial state. This initial state can either be defined by an existing class (reuse) or be an instance (component) of a new class, which can then be extended by the application of further productions.

The insertion of an initial state consists of inserting a state component, connecting it to the event distribution net and connecting the *setState* interface to the initial state net. The inserted state is either a subclass of *abstract_sequential* or *abstract_parallel*. Here an undirected arc connects the *readEvents* interface with the transition *init_Complete* ensuring that the first event will be accepted only after the set of valid events is available to be read.

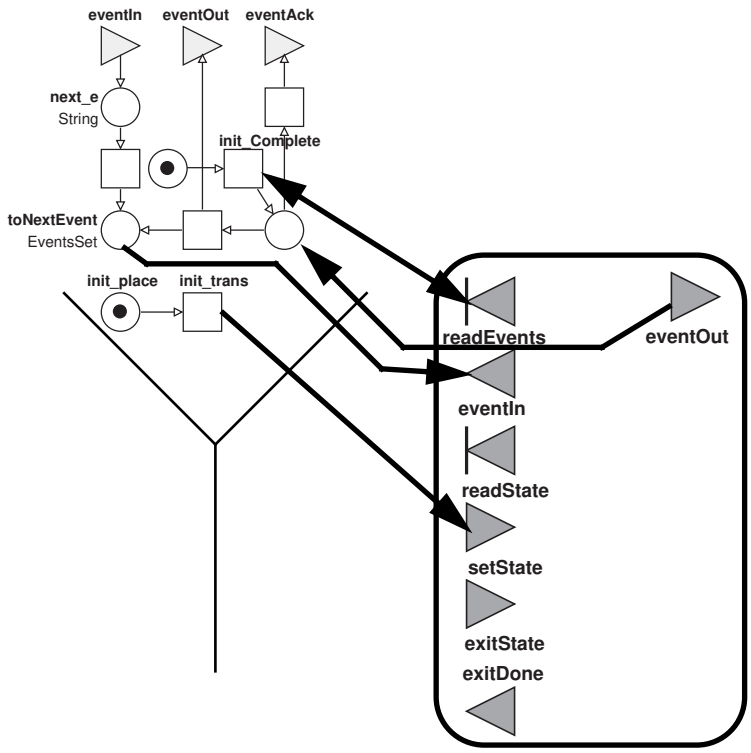

FIGURE 69. ***Add_Initial_State*** **semantic graph grammar**

Different productions are available depending on whether a subclass of *Abstract_sequential* or *Abstract_parallel* is being edited. In both cases it is possible to insert both parallel and sequential states. However in subclasses of *Abstract_parallel* no productions have been defined to add state-transitions as this would reduce the possibility of reusing the involved states. Figure 65 shows the syntax graph grammar production for inserting a state into a sequential state. The semantic graph grammar production in figure 70 shows that each state consists of a non trivial Petri net implementing the event distribution and collection mechanism and state change mechanism.

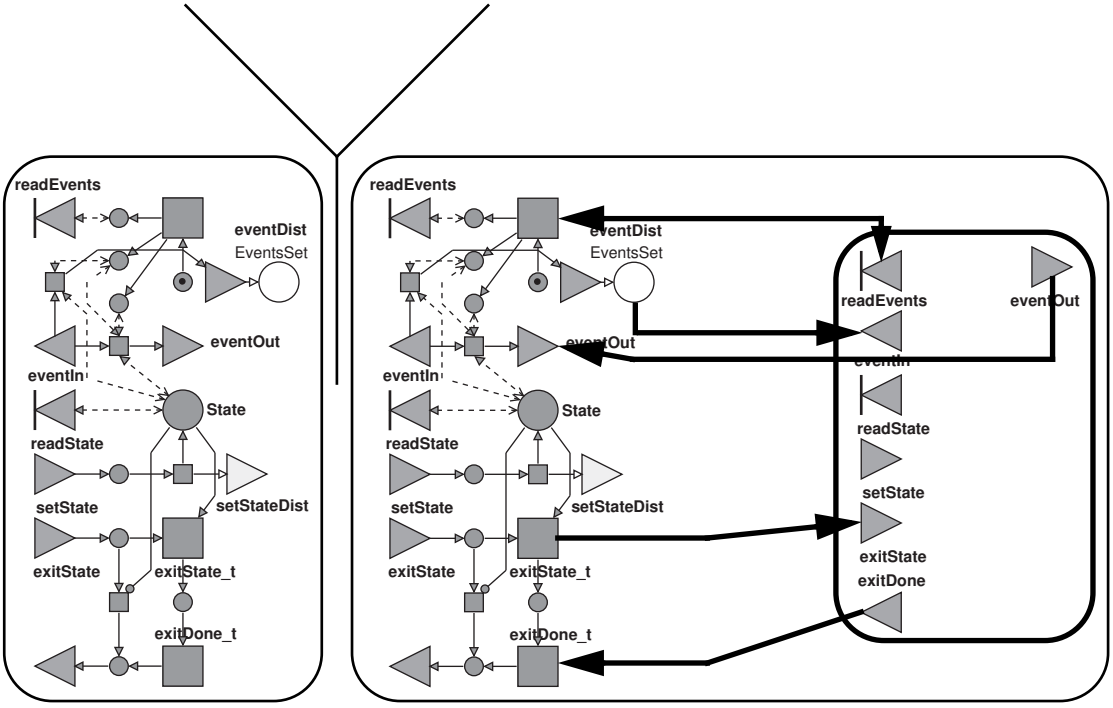

FIGURE 70. ***Insert_State_Into_Sequential*** **semantic graph grammar**

Inserting a state within a parallel state is virtually identical to inserting a state within a sequential state except that elements are inserted to multiply, synchronise and collect events to and from all parallel states.

Inserting State-Transitions

Figure 66 shows the syntax graph grammar for inserting a state-transition. Figure 71 shows a state-transition modelled with a kernel component. This component is connected to the event distribution mechanism and will allow the state-transition it represents to occur when an event is present that is included in the list of events stored within it. The occurrence of a state-transition can be defined as a series of steps:

1. enabling: the input state is active and the input event is a member of the set of events that the state-transition defines;
2. exit state: a message (token) is sent to the input state in order to exit. This message is rippled to all states contained within the input state;
3. enter state: a message is sent to the output state to become active;
4. event generation: when the output state acknowledges that it is active the internal events defined within the state-transition component are sent to the event collector and will become visible at the next step.

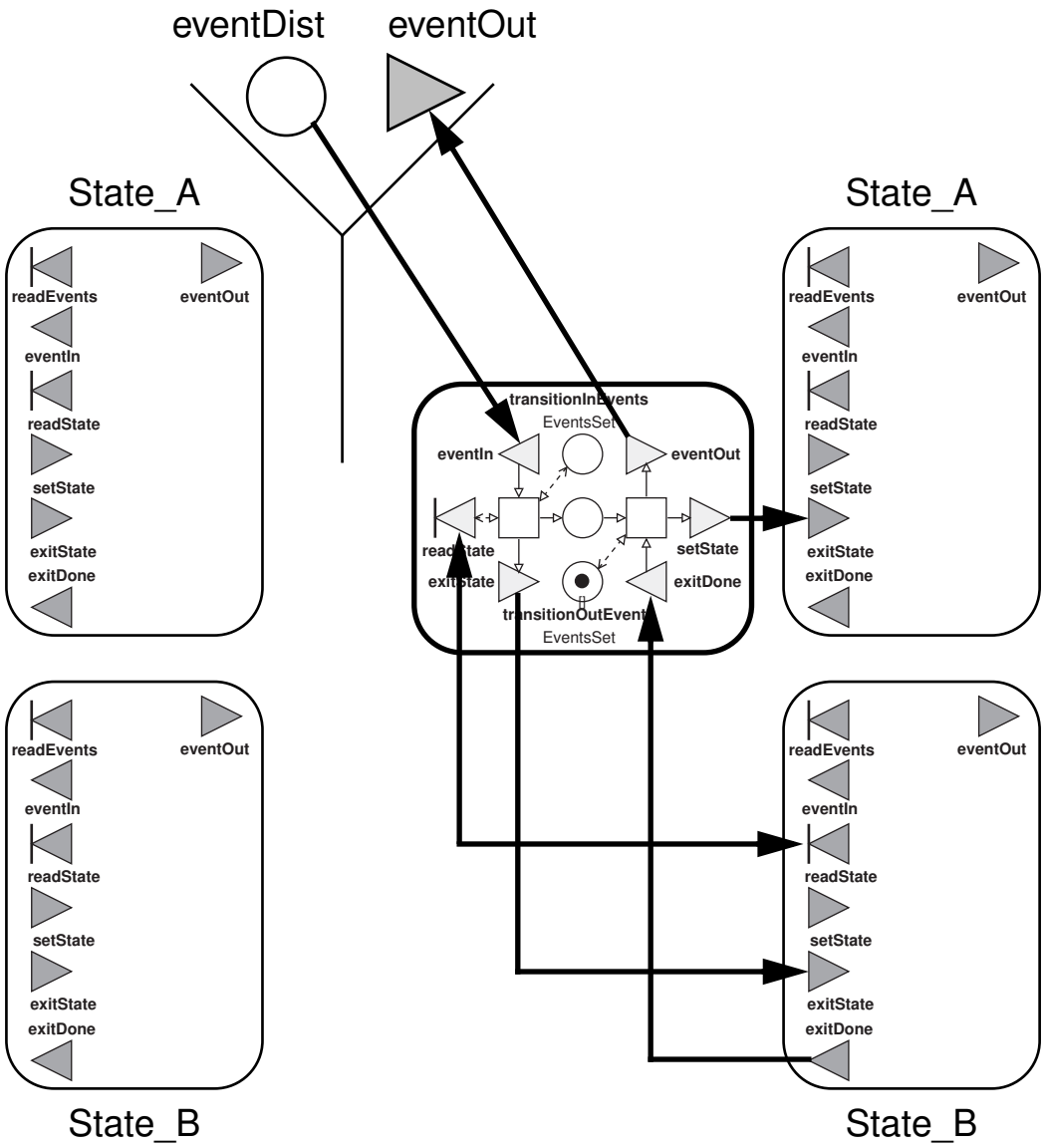

FIGURE 71. ***Insert_Transition*** **semantic graph grammar**

6.3.6 Defining the Graphical Representation

To complete the description of a formalism functions are created that define how a formalism's element is graphically represented (shape, weight etc.) These functions are associated with the component classes that represent the element and are called by the editor when displaying. In addition state information and animation productions for each formalism element need to be described. In the case of StateCharts each state is represented by a rounded box which can be made transparent to show the internal states. State-transition components are represented as arcs whose colour indicates whether it is active or not. An active state is indicated by a double lined rectangle.

Finally functions are created that define the interface to the editor. This allows a formalism specific editor to be generated in which actions cause previously defined productions to be executed.

6.4 Other Formalisms in an Object Oriented Environment

As described previously in this chapter other formalisms are defined with the kernel language. This kernel language is object oriented facilitating the creation of class templates which are instantiated in a number of contexts. Any element of a formalism described as a component of a class can be subclassed. As with the components of the kernel language, a subclass may be extended and refined in a number of ways (see 'Components' on page 112). This is an important feature of the object oriented approach and when applied correctly results in considerable gains in productivity [22]. In object oriented systems results are reused from either subclassing or embedding.

The subclassing of a formalism's elements allows refined elements to be used in place of the standard one. The kernel language inheritance rules ensure that the refined element will always be able to be used in the correct way. Subclassing may be prevented by defining a class to be monomorphic.

This ensures that only suitable classes are able to be refined. Assuming that the functions representing SemGG productions allow a choice of component classes to be instantiated, subclassing is available to all formalisms. This enables abstraction and refinement to be used.

Embedding is a formalism specific feature and usually represents a hierarchical structuring. Formalisms that do not support hierarchy, such as SDF, cannot therefore make use of embedding at the formalism level. In contrast StateCharts do support hierarchy and can benefit from the advantages of code sharing.

The following figures show an example of subclassing and embedding using the StateCharts formalism. Three classes are used to build a state machine able to count to 4 and 8. Figure 72 shows a subclass of *Abstract_Sequential*. This class implements a simple two bit counter. It does not define a state-transition from state *b1* and state *b0* as this would prevent any subclass from implementing a larger counter.

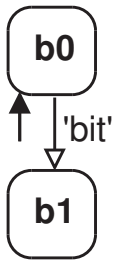

FIGURE 72. The Class *Bit_2* implementing a 2 bit counter

Figure 73 extends the 2 bit counter to 4 bits. It is a *subclass* of *Bit_2* and has been enhanced with the addition of two states (*b2*, *b3*) and 3 state-transitions. The states and transitions inherited from the class *Bit_2* are shown as being shaded. The state-transition from state *b3* to state *b0* produces an event named *'nibble'*. This event is broadcast throughout the state machine and in addition is made available through the external event interface from where it can be used by other parts of a model, modelled in perhaps different formalisms. Class *Bit_4* reuses the states and state-transitions defined in class *Bit_2*.

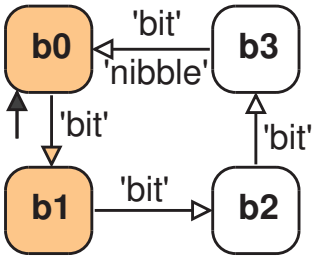

FIGURE 73. The Class *Bit_4* implementing a 4 bit counter

Finally in figure 74 the class *Bit_8* is shown to consist of two *embedded* components of class *Bit_4*, one of which is marked as the initial state. As the two component hierarchical states are instances of the same class they only differ in their dynamic state information. Ignoring the internal states of both components of class *Bit_4* the system can be regarded as only having 2 states (*b_0_3*, *b_4_7*) and 2 state-transitions that occur when event *'nibble'* is present. The state-transition from state *b_4_7* to state *b_0_3* produces the event *'byte'* to indicate that 8 bits have been counted.

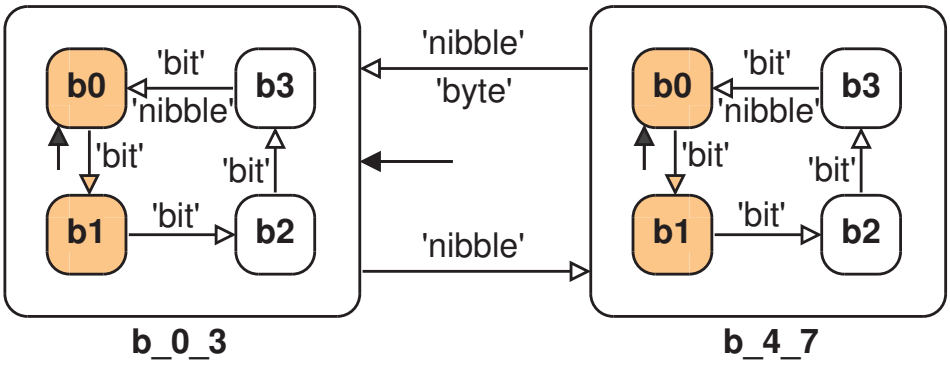

FIGURE 74. The Class *Bit_8* implementing a 8 bit counter

Figure 75 shows part of the class hierarchy used to implement the 3 classes above.

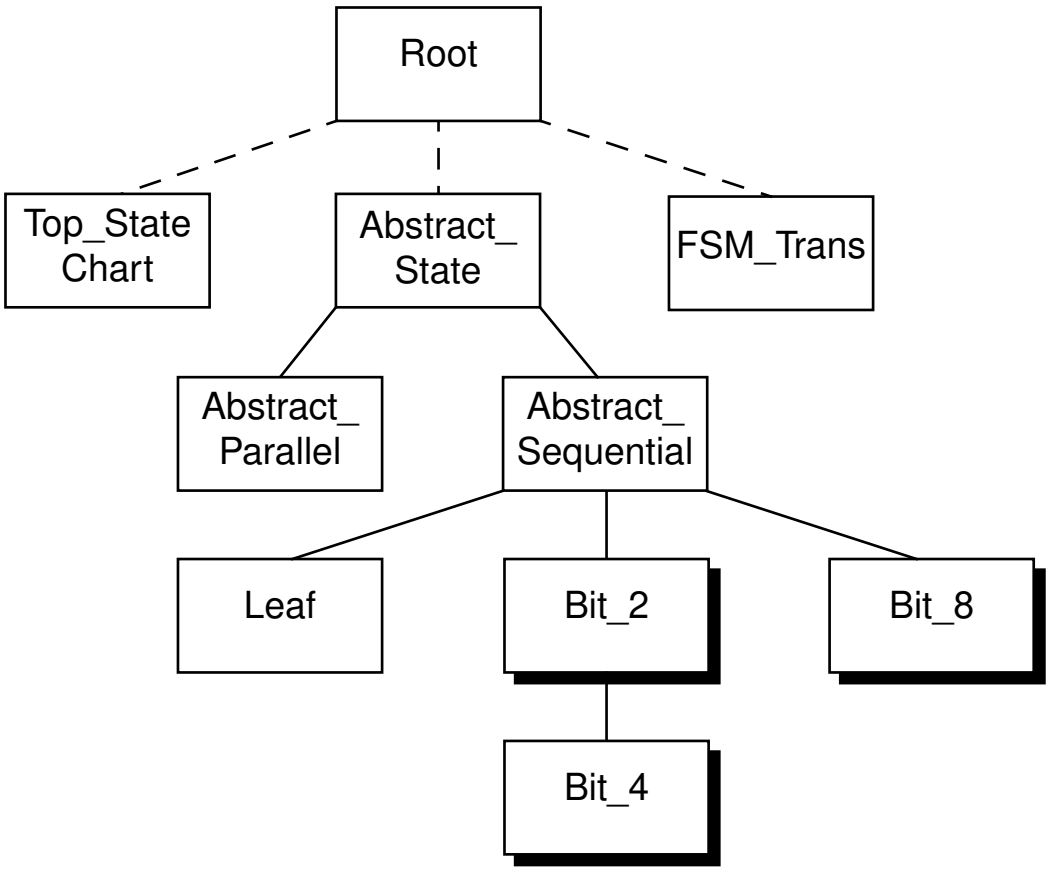

FIGURE 75. Class hierarchy tree for StateCharts classes implementing an 8 bit counter

Figure 76 shows part of the instantiation tree for a component of the class Bit_8. This component can be embedded within another class or used directly within a subclass of *Top_State* where the production will connect the event mechanism to the component. This figure does not show the kernel components used to implement the elements of the StateCharts formalism. The components marked *inherited* are defined in the parent class of the defining class.

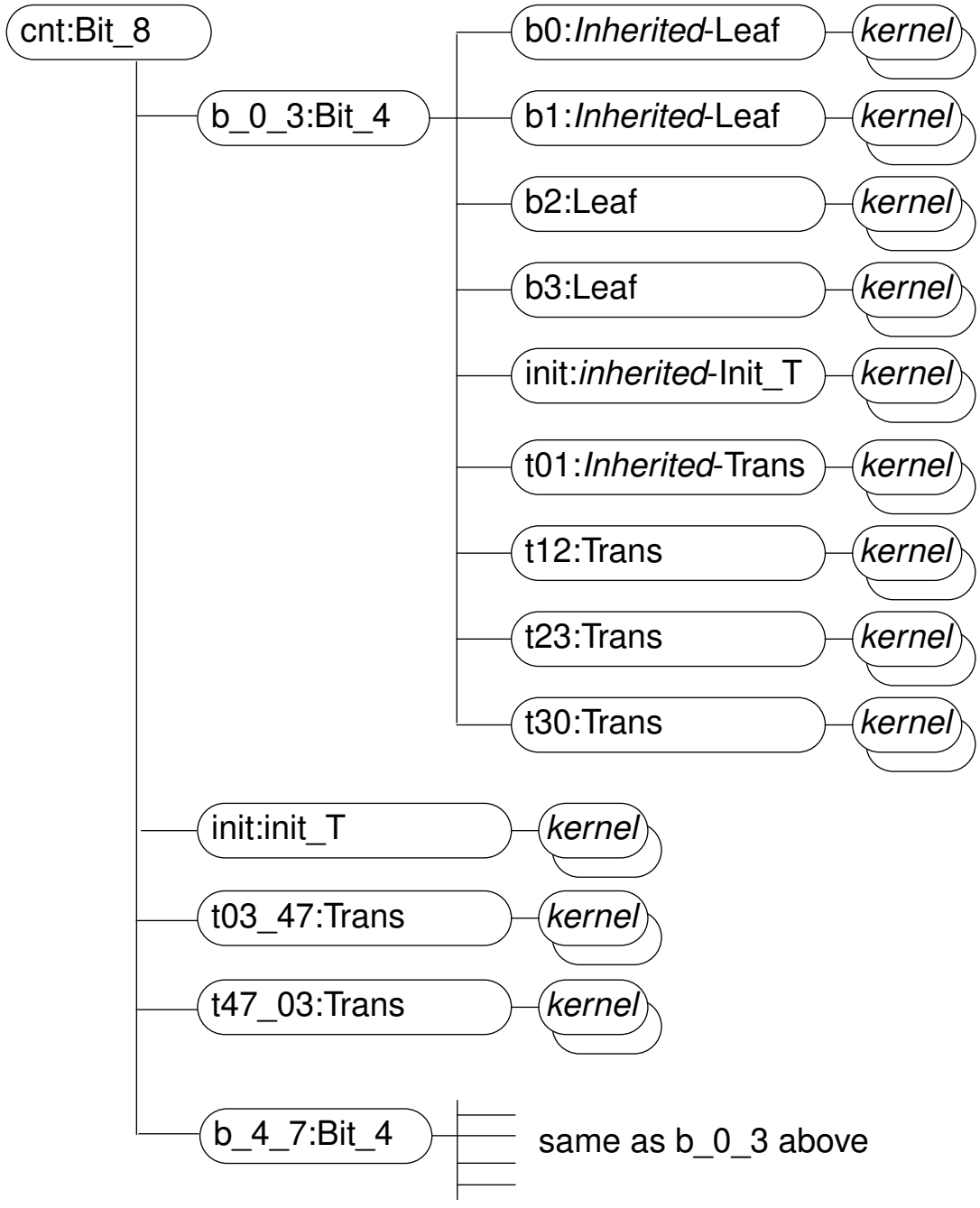

FIGURE 76. Instantiation tree of the class *Bit_8* (figure 74)

6.5 Summary

Unlike other approaches that use special languages to represent other formalisms this approach uses the same object oriented time Petri net kernel language to model other formalisms, including the productions for editing them. This is beneficial as only a single modelling language is involved in modelling both a system or a formalism. In addition all the benefits of the object oriented approach are applicable for models described with other formalisms and for the description of the formalisms themselves.

The following steps define a how a new formalism is supported:

- the syntax rules of a new formalism are represented graphically as graph grammar productions, called Syntax Graph Grammar (SynGG) productions;
- an object oriented time Petri net component class is associated with every element in a formalism. The resulting class hierarchy represents the elements of a formalism and reflects that some elements may have similar behaviour which can be inherited. All formalisms are object oriented and models may use the refinement techniques afforded by the object oriented approach;
- for every SynGG there is a Semantic Graph Grammar (SemGG) production which describes the semantics of the syntax defined by the SynGG. These semantics are defined in terms of the kernel language, i.e. object oriented time Petri nets and involve the component classes defined to represent the formalism's elements. Functions are then written representing the productions defined by the SemGG and are associated with a particular class in the object oriented environment;
- functions are created that define how elements of a formalism are graphically represented (shape, weight etc.) In addition functions are created that define the interface to the editor.

CHAPTER 7 *Example*

The evolution of a simple telephone model illustrates the approach described in the previous chapters. The familiar example of a telephone allows important features, such as support for multiple formalisms, configuration and refinement to be applied and demonstrated.

It is beyond the scope of this monograph to document the complete design of a telephone, however the major steps in the proposed approach have been carried out using the example to highlight important aspects. Two components of the telephone are refined to accurately model their required functionality while others have been left abstract. This example will not show those aspects of the traditional codesign approach, i.e. partitioning, interface generation, hardware and software synthesis etc. Instead emphasis will be placed on the highest levels of abstraction - system specification and modelling.

7.1 Requirements

- using a telephone, calls can be originated and received;
- the telephone will be connected to the standard telephone system;
- the telephone will allow the last number dialled to be redialled by pressing a single button.

These are representative of an informal requirements statement as many important questions are left unanswered. For example it neglects to define the scope of the telephone, whether the telephone is a member of a family of telephones (where similarity between various models can be exploited) or whether it must implement the basic functionality as cheaply as possible. It also assumes that the physical design does not place any restrictions on the implementation of the system in order to achieve the desired functionality.

The standard telephone system defines the interaction protocol between the telephone and the system and the electrical interface, including the maximum power that can be extracted from the telephone line which imposes a maximum power constraint on the system.

The above requirements imply that the telephone only implements the redial function and does not contain a memory for additional numbers. It neglects to define the maximum number of digits that can be stored and what occurs if this maximum number is exceeded.

For the purposes of this example it is assumed that further enquiries have shown that:

- the telephone must implement the required functionality as cheaply as possible;
- it should be able to be used in as many countries as possible.

In many countries there is a requirement for pulse dialling whereas in others there is not, as tone dialling is used. As pulse dialling capability adversely affects the resulting cost of the telephone a model will be created which supports two configurations - with and without pulse dialling.

7.2 System and Environment

The system and its boundary with its environment are defined in the first step of the system design process. Figure 77 illustrates this, where the tele-

phone includes a man machine interface including a keyboard, handset and a connection to the telephone system which is assumed to be a standard two wire connection.

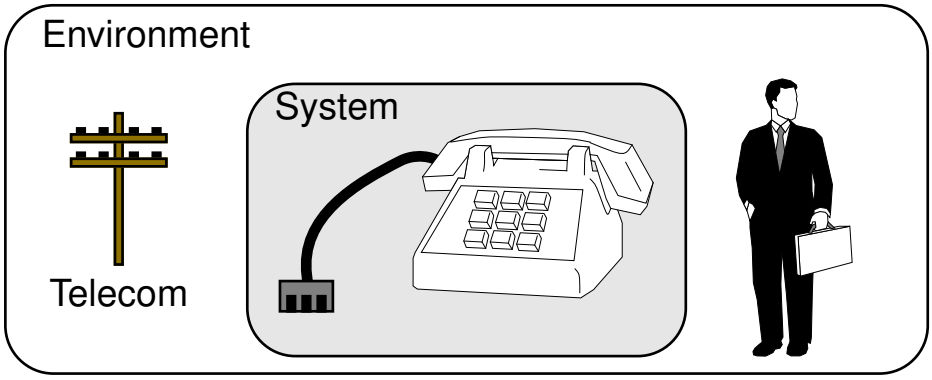

FIGURE 77. The (telephone) system and its environment

The interface components accessible to the user are: the *keyboard* consisting of the standard 12 keys and an additional key to initiate last number redialling; the *handset* containing a *speaker* and *microphone* which, when lifted, operates the *hook*; the *bell* used to make the ring tone audible and the *tone/ pulse dialling switch* which selects the mode of dialling and may be omitted for telephones only supporting tone dialling.

The model of the environment also includes a model of the telephone exchange, for this example referred to as *Telecom*. This model must include all the logic required to make a call including on / off hook detection, digit decoding, various tone generation (dial, busy, ring and finished) and a switching component allowing two telephones to be directly connected.

7.3 Scenarios

The next step in the design process is to formulate *scenarios* [99] that the system must implement. A scenario defines the messages that arise between components of the system and environment as a result of a particular event occurring.

Successful Call Scenario

- user A wants to call user B. User A lifts his handset and hears the dial tone. He enters user B's number and hears a ring back tone informing him that user B's telephone is ringing. User B answers the call by lifting her handset. The ringing and ring back tone stop and a voice connection is made between the two telephones. Either user A or user B can hang up to end the call.

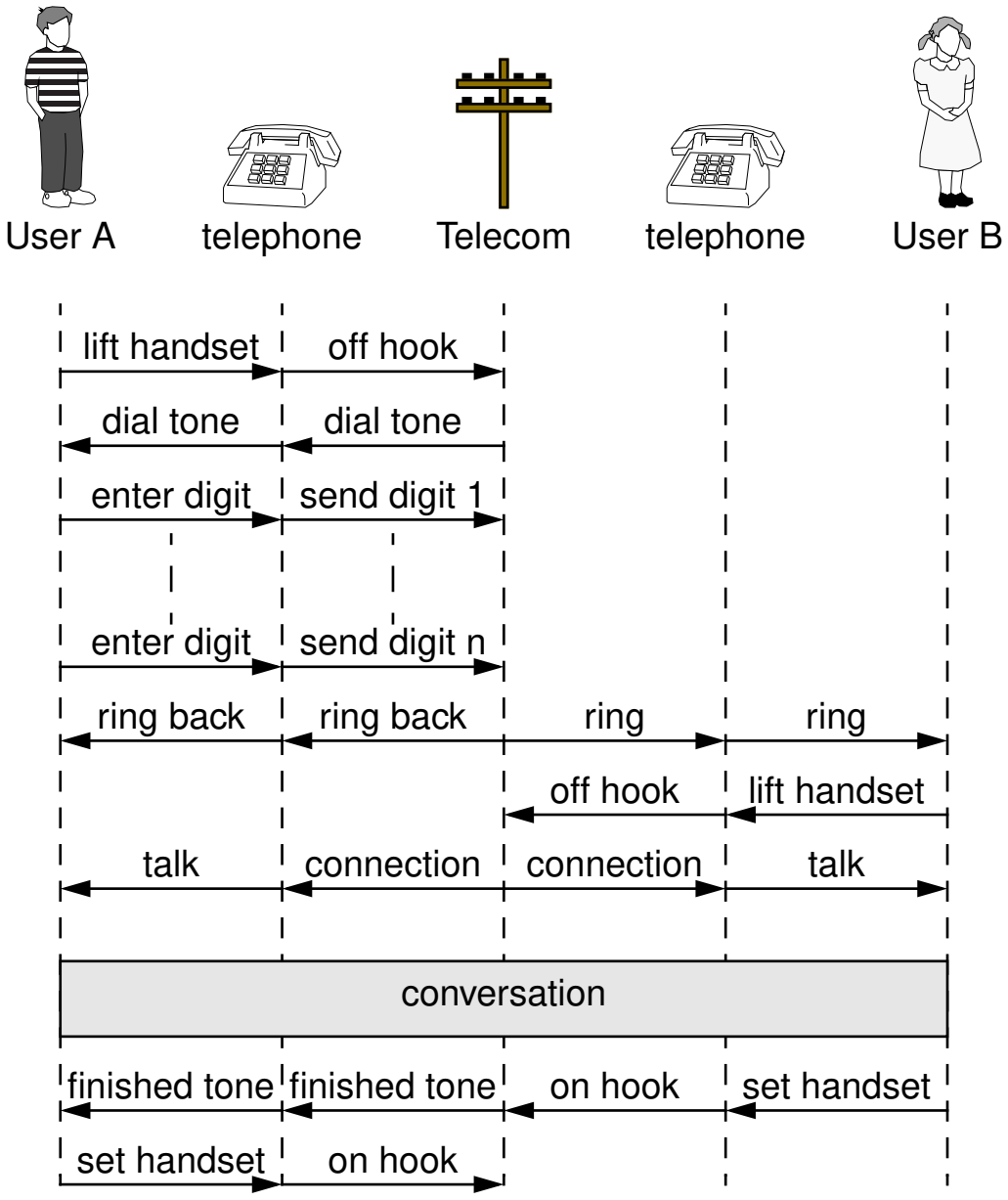

FIGURE 78. A successful call scenario

Figure 78 shows the messages that flow between the environment and the telephone during a successful call. In effect it defines a protocol that the telephone exchange and users execute during the course of a call.

Busy Scenario

- user A wants to call user B. User A lifts his handset and hears the dial tone. He enters user B's number and hears a busy tone informing him that user B's telephone is busy. User A hangs up to end the call.

Figure 79 shows a busy scenario where user B is not able to answer the call.

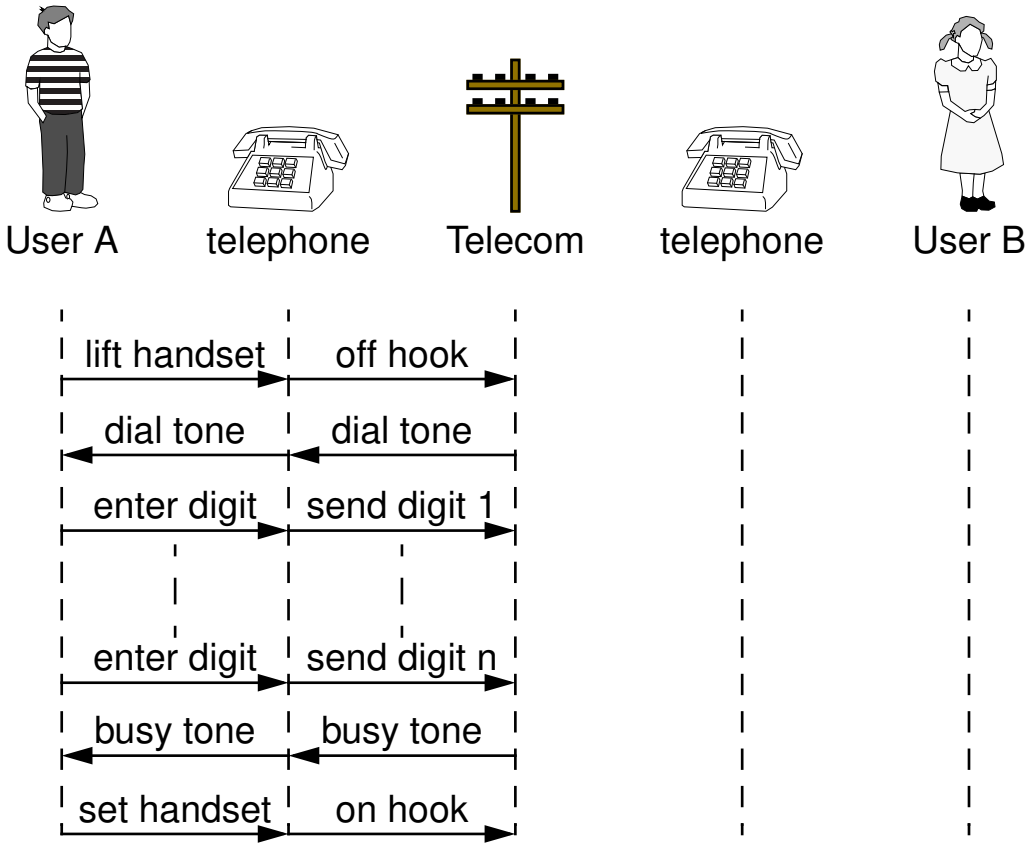

FIGURE 79. A busy scenario

Redial Scenario

- user A can call user B again by lifting his handset and pressing the redial button.

Figure 80 shows a redial scenario. In this case it is assumed that the redial memory contains a stored number.

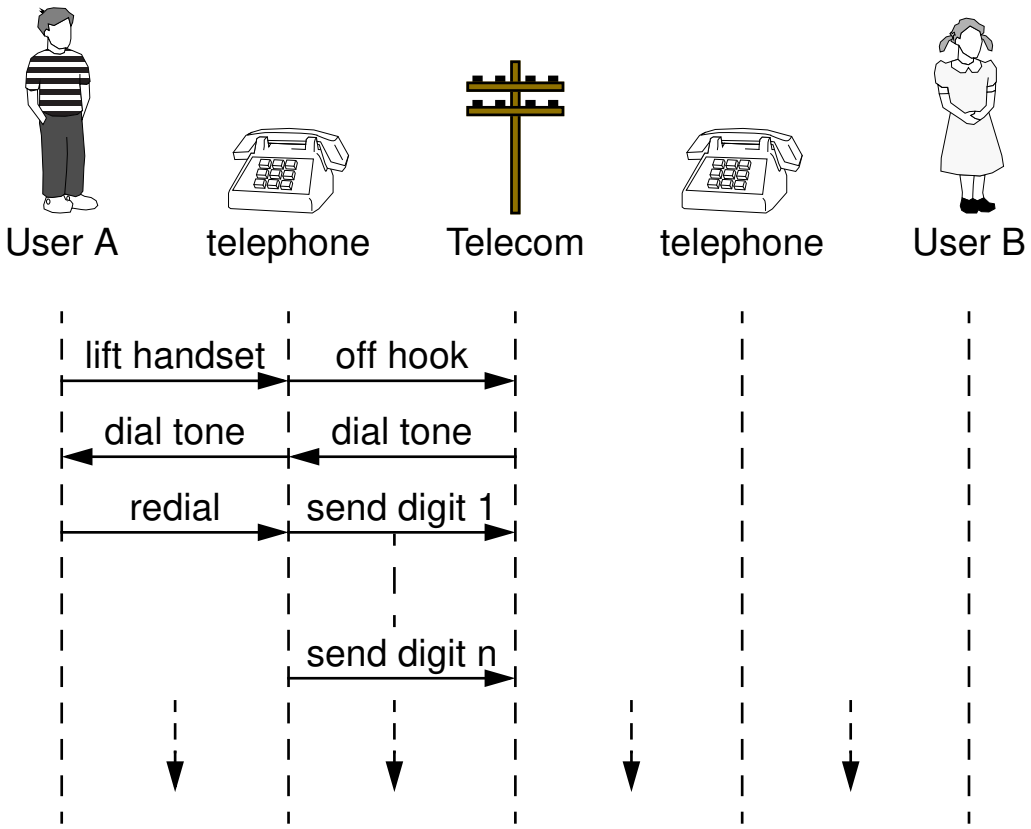

FIGURE 80. A redial scenario

Many more scenarios would normally be captured to describe a system, even one as simple as the telephone. Initial scenarios focus on system interaction leading to more detailed scenarios and are a method of describing a systems functionality. In particular, scenarios depicting exceptional circumstances are required. During the course of a design more detailed scenarios are defined as an aid to document the required functionality of, not only the whole system, but also that of individual subsystems.

7.4 Environment Model

The approach presented in this monograph is optimised for embedded system design. Embedded systems have a complex interaction with their environment and to successfully design them it is essential that the behaviour of the environment is completely understood. The process of defining the system / environment boundary, in addition to creating a model of the environment, leads to a better understanding of how the system must behave in order to implement the required functionality.

During the design process the environment model may be further refined or successively replaced by the real environment. A further benefit of creating a model of the environment is that it may be used for system validation tests.

Figure 81 shows an abstract representation of a complete telephone system. It consists of two instances of the telephone and a model of the environment consisting of two abstract representations of users and a model of the telephone exchange.

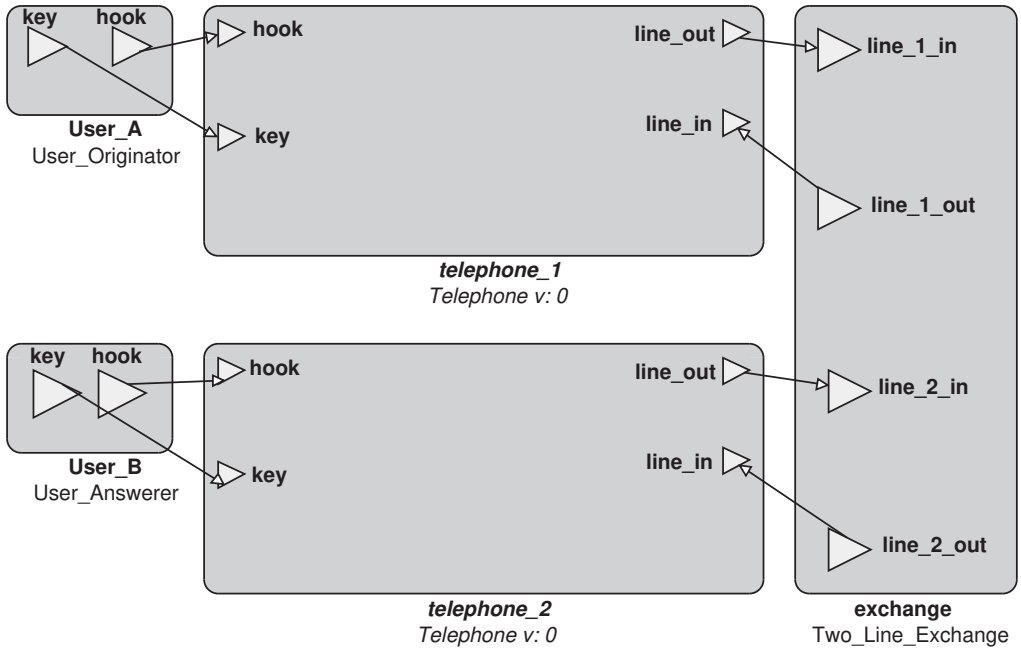

FIGURE 81. An abstract representation of a telephone system

Figure 82 shows a Petri net representation of two users. In this model user A originates a call by lifting the handset (transition *offHook*) and dialling a two digit number (transitions *dial_1*, *dial_2*) and finally terminating the call by replacing the handset (transition *onHook*). User B answers the call by lifting the handset (transition *offHook*) and replacing it after the call is completed (transition *onHook*). The inherent time model can be used to model the range of possible times within which the environment (users) operates, by initiating or reacting to events. The Petri nets below have time constraints modelling a large range of users from slow to fast.

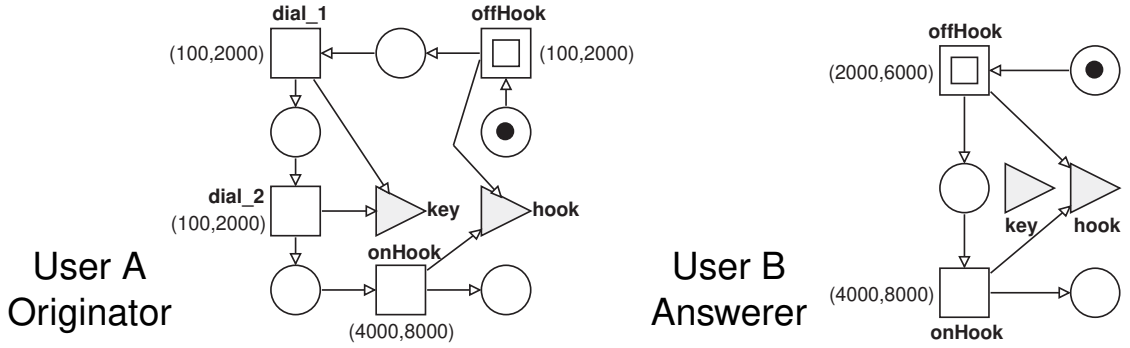

FIGURE 82. The Petri net representation of two abstract users, an originator and answerer showing time constraints.

Using the configuration capability it is possible to insert different user models enabling the telephone system to be tested using the various scenarios. The following two functions define which of two classes is instantiated to represent the originator or answerer.

```
FUNCTION originator : User_Originator
"select the type of originator to instantiate"
{IF systemKind = 'redial' THEN
    {RESULT := User_Orig_Redial.new}
ELSE
    {RESULT := User_Originator.new}}

FUNCTION answerer : User_Answerer
"select the type of answerer to instantiate"
{IF systemKind = 'redial' THEN
    {RESULT := User_Ans_Redial.new}
ELSE
    {RESULT := User_Answerer.new}}
```

Figure 83 shows the model of an abstract telephone exchange. This model consists of two line cards which are directly connected to a telephone, modelling the logic required for each telephone. The exchange switching component is a simple model of the switching part of a telephone exchange. In this case the exchange switching component can only switch these two telephones together. In figure 83 the three Petri net elements are used to insert events generated by both line cards into the exchange switching component (*switcher*).

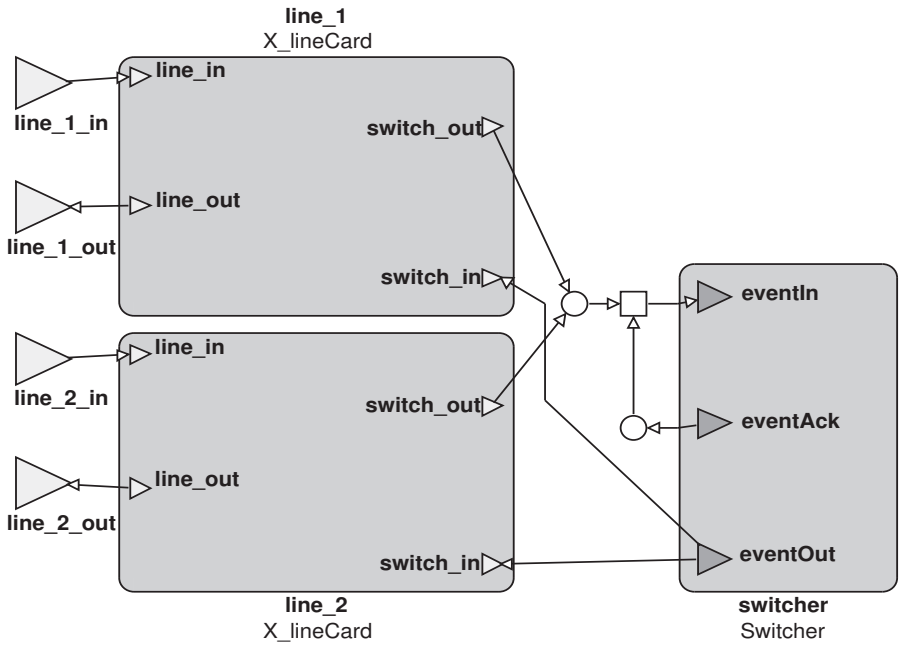

FIGURE 83. An abstract representation of a two line telephone exchange

Figure 84 shows the line card consisting of a controller modelled with a hierarchical state machine, a timer (to detect time-outs) and a line interface that detects the state of the telephone's hook switch and recognises the dialling digits. The state machine implements the call protocol by receiving events from the line interface (*offHook*, *onHook* and *digit*), the timer (*timeOut*) and the switcher (*busy*, *ring*, *connect* and *disconnect*). It also produces events to restart or clear the timer (*setTimer*, *clrTimer*) and events that configure the line interface (*ringTone*, *busyTone*, *silence*, *dialTone*, *finishedTone*

and *connection*). The additional Petri net elements serve to insert, filter and route events.

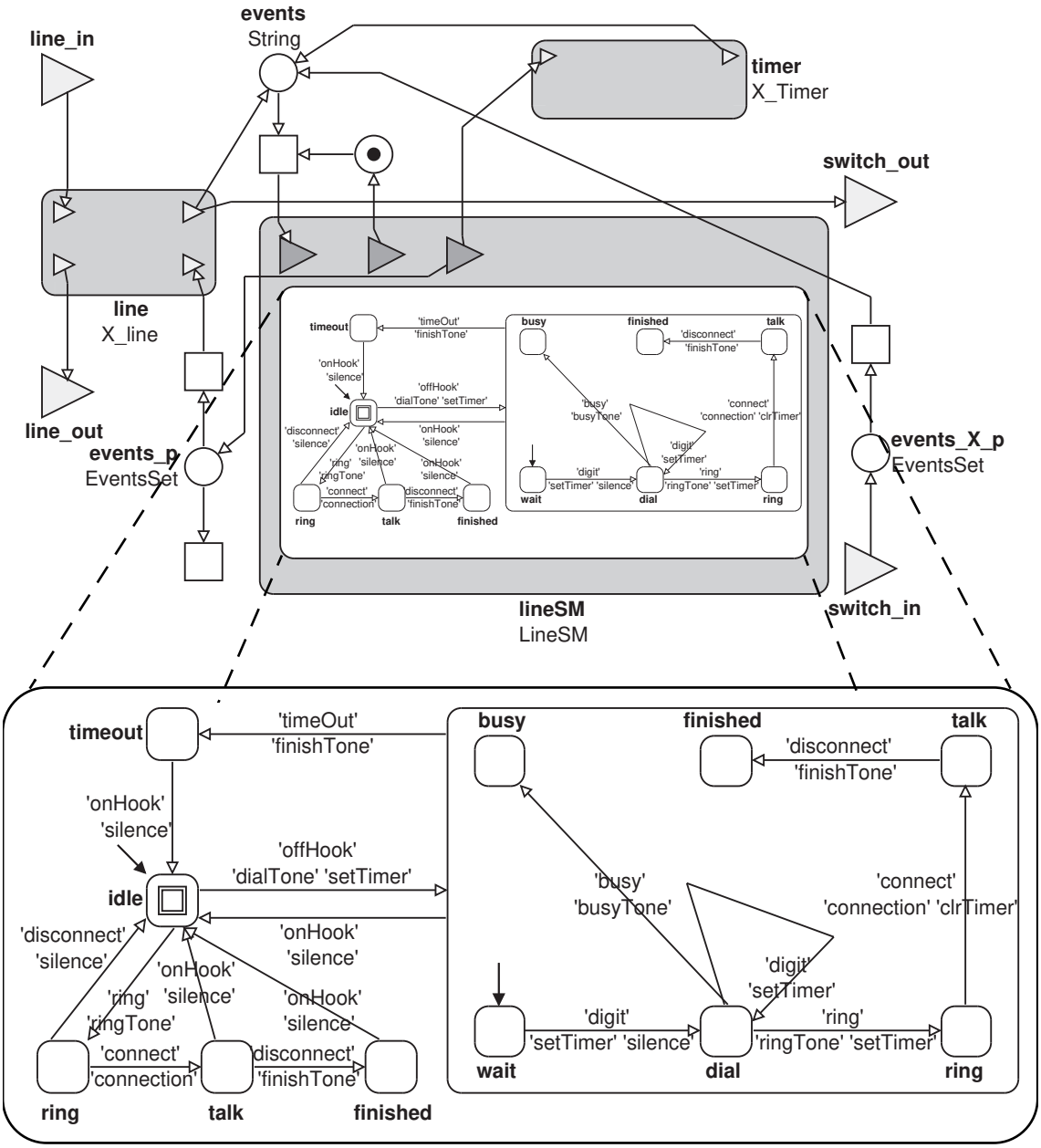

FIGURE 84. The model of an exchange line card

The exchange switching component *switcher* is shown in figure 85 and consists of a state machine which detects when both telephones are *off hook*. This is a very simplistic but sufficient model of a simple two line telephone exchange. The exchange switching component includes a state machine to

recognise the digits a user enters to make a call. Two state-transitions from state *first* to *last* model the possibility of the line being busy. The exchange switching model assumes that all calls only require exactly two digits to be entered. This does not imply that a telephone's redial memory need only store two digits as real telephone systems require more digits to be entered and highlights the requirement that the designer be aware in what regard an abstract (environment) model deviates from the reality.

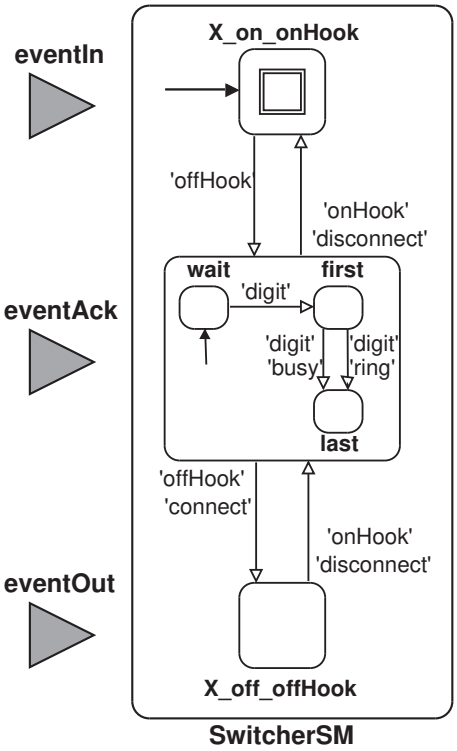

FIGURE 85. An abstract model of the exchange switching component

Often the creation of the environment model represents a significant, albeit necessary, proportion of the total modelling effort. The resulting model can be used for system validation tests and also be reused in subsequent projects, in this case for any further telephony projects.

7.5 System Structure

The structure of the embedded system can be derived by analysing scenarios and identifying candidates for concurrent execution, between which messages flow. In figure 86 the telephone structure is defined as consisting of a *dialler* and *line* component in addition to components representing the environmental interfaces. The telephone itself has a line interface to connect to the exchange and a *User_hook* and *User_key* interface enabling the user models to be connected together. The *dialler* and *line* components were required to model a minimal functionality. In the case of the originator, pressing a key causes a *digit* event to be produced from the dialler which is passed on by the line component. This model is very abstract but nevertheless is able to be embedded into the environment model and simulated. It does not implement redialling but was used to test the environment model using the successful call scenario.

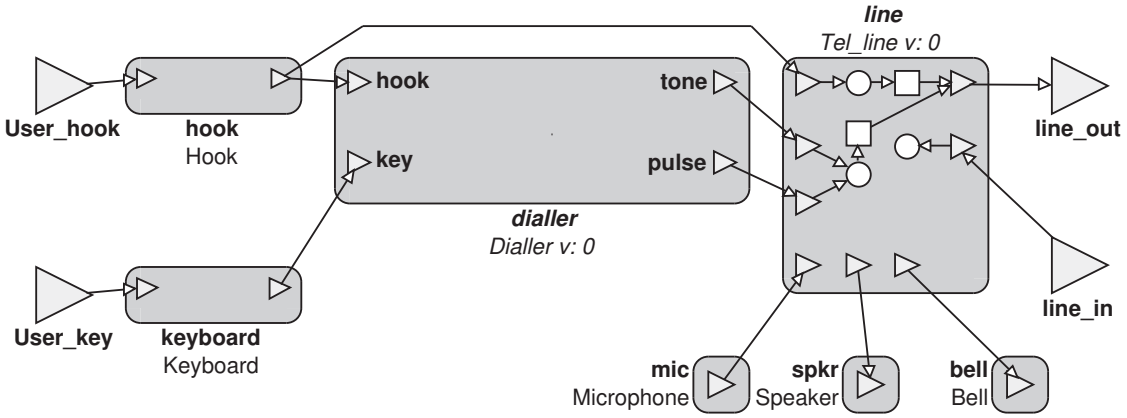

FIGURE 86. The initial structure of the telephone represented by the *Telephone version 0* class

Figure 87 shows a refined telephone system structure able to model all scenarios including last number redial. It includes a component *controller* that represents the last number redial functionality and number store. The *dialler* is a component of *Dialler version 1* class and differs significantly from the *Dialler version 0 class*. It can be seen that the hook component is now connected with the *controller* with the *dialler* only receiving digits from the

controller. An additional *ack* interface between the *dialler* and the *controller* has been inserted to ensure that both components remain synchronised.

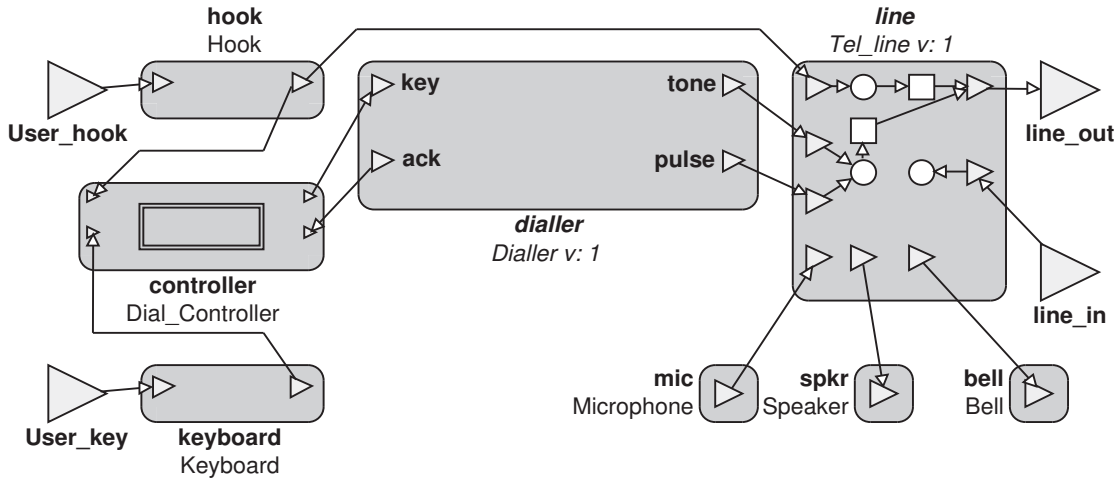

FIGURE 87. The refined structure of the telephone represented by the *Telephone version 1* class

7.6 System Model

Once the system structure has been defined the individual components can be modelled. Figure 88 shows the *Dial_Controller* component class in which a state machine implements the logic required to either redial the stored number, or to enter a new digit into the number store.

While modelling this component it was noted that it was possible to enter digits quicker than the pulse dialler could accept them. This is also true for last number redial where the digits are streamed to the dialler. A solution uses an acknowledge event from the dialler to indicate its ability to accept a new digit and an asynchronous read process. This sends a digit to the dialler when an acknowledge event is received and when a digit is able to be sent. The dial controller is modelled as having a controller which manages the number store where the three transitions *firstDigit_t*, *nextDigit_t* and *redial_t* accept events from the controller and manipulate the number store. The top transition models the asynchronous read process in which a digit is

sent when the *dialIndex* is not equal to the *insertIndex* and an acknowledge event from the dialler (*next* interface) has been received.

In this component class not all data types have been sufficiently refined. The place *key_p* is typed with the class *Keys* where each key is assigned a unique code, however the place memory is typed with the *Collection* class which models an unrestricted collection of objects. The indices to this store are also unrestricted integers. A refinement step would replace these types with restricted types which reflect the type of objects to be stored (digits) and the maximum size of the number store.

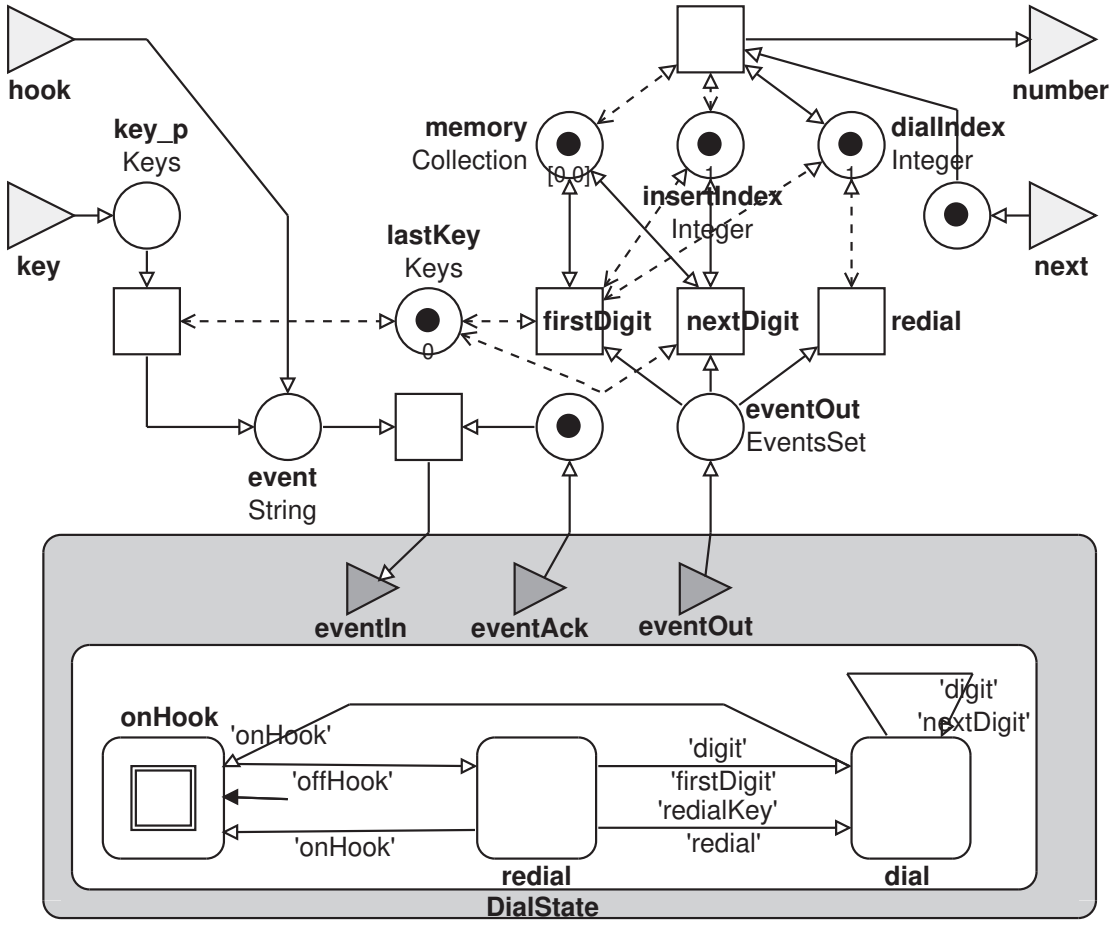

FIGURE 88. The *Dial_Controller* class

7.7 Refinement and Configuration

The refinement process and configuration can be illustrated with a description of the dialler. The refinement process involves the development, analysis and validation of components. This is carried out in successive abstraction steps, which increasingly include more detail. Configuration is used as the system has more than one possible form. In the case of the telephone, pulse dialling is assumed to be optional, requiring two types of dialler components - one with tone dialling and another with tone and pulse dialling. Configuration can also be used during design exploration to choose between particular compatible components which may have different (performance, implementation, power etc.) characteristics.

Figure 89 shows a further refined version of the telephone system. *Telephone version 2* consists of components having the latest versions of all component classes and contains instantiation code to configure it with the particular instance of the dialler - with or without pulse dialling. The line is considerably more complex but still abstract, translating dialling information (tone and pulse) into events to send to the telephone exchange model. Subsequent refinement would model the physical signals on the telephone line and include the routing of signals from the dialler and microphone and to the speaker and bell. This would also necessitate a refinement in the environment model to enable detection of tone and pulse dialling and to generate different tones such as dial, ringing, busy and finished tones.

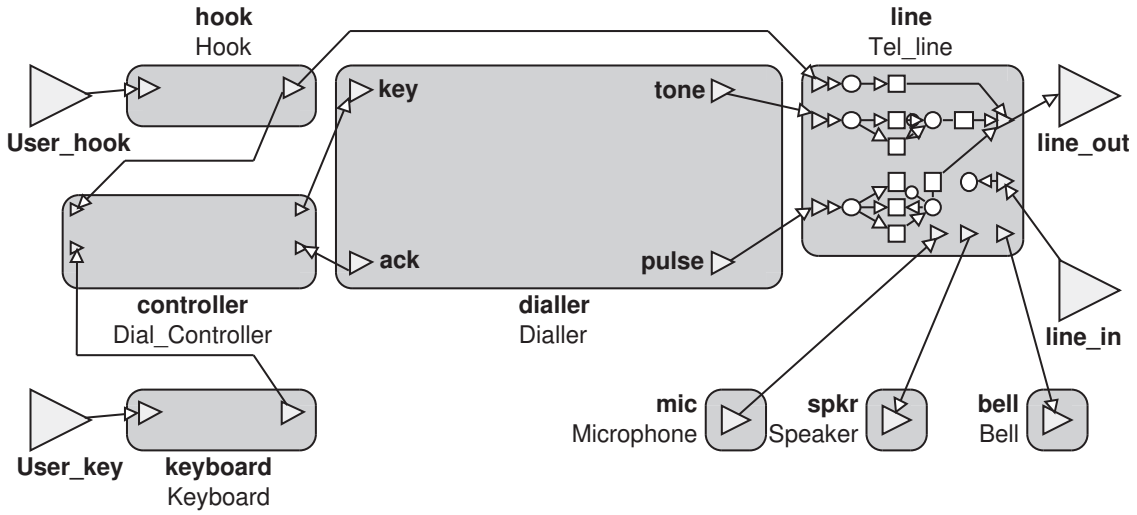

FIGURE 89. The *Telephone version 2* class

Figure 90 shows the *Dialler version 2* class implementing a dual tone multiple frequency (DTMF) encoder. It includes a component implementing the encoder and the interfaces from the controller and to the line component. When compared with the *Dialler version 1* class, in figure 87, it represents a significant refinement step from a very abstract model to one which models the complete functionality.

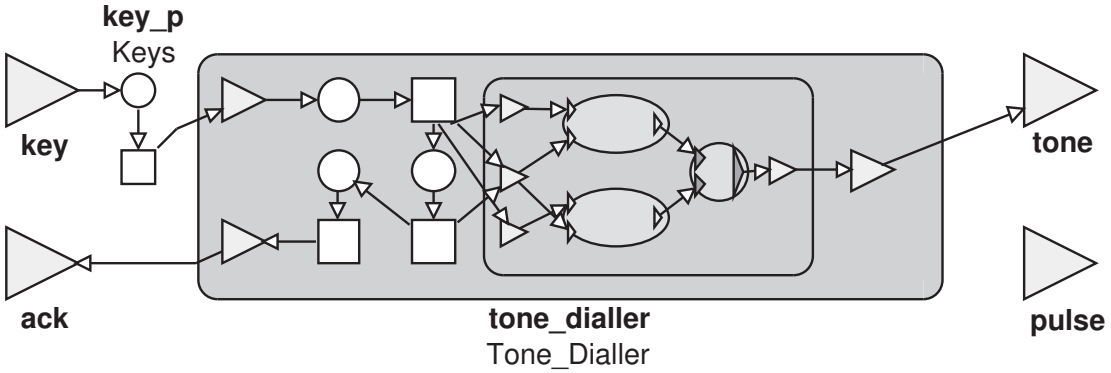

FIGURE 90. The *Dialler version 2* class implementing a DTMF encoder

Figure 91 shows the *Full_Dialler* class - a subclass of the *Dialler* class shown in figure 90. This class inherits components (dark shading) including the tone dialler component and inserts a pulse dialler component and a

switch for selecting between the two diallers. The *Telephone* class will either instantiate the *Dialler* class or the *Full_Dialler* class.

The *tone_dialler* implements a DTMF encoder. This is modelled with a small Petri net where the frequency selection and control of the tone generator is implemented. The tone generator is modelled using the synchronous data flow (SDF) -like formalism.

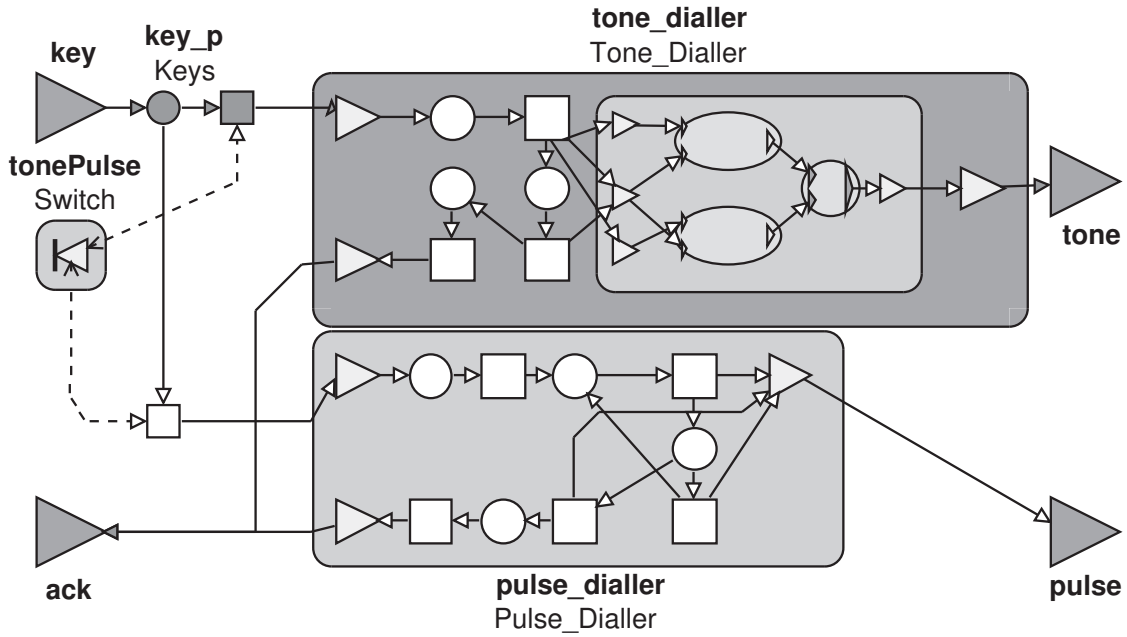

FIGURE 91. The *Full_Dialler* class modelling both tone and pulse dialling

Figure 92 shows the SDF-like model where the class *Prog_Oscillator* is instantiated twice to represent the two tone oscillators and an adder sums the outputs from the oscillators. The inputs *lowFreq*, *highFreq* and *start* are used for controlling the oscillators which, when started, produce a stream of tokens at a given sample rate. Strictly speaking these elements are not SDF nodes as their inputs are used for controlling the outputs and do not require a token to be input for every token generated on the outputs. It would be possible to model the oscillators with SDF nodes having only outputs, implying

that the frequency to be generated is somehow made known to the oscillators.

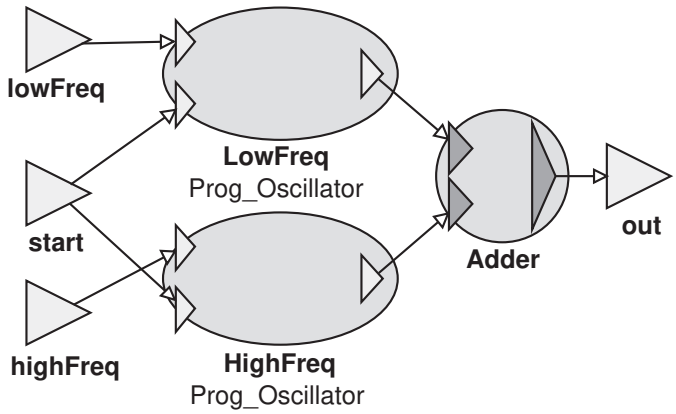

FIGURE 92. The *DTMF* class

Figure 93 shows the *Prog_Oscillator* class. Here it can be seen that the last transition which produces tokens through the *out* interface accesses control and state data. These can be changed by tokens arriving at the input interfaces. A further refinement would substitute less abstract place types in the *freq*, *freq_in_p* and *value* places.

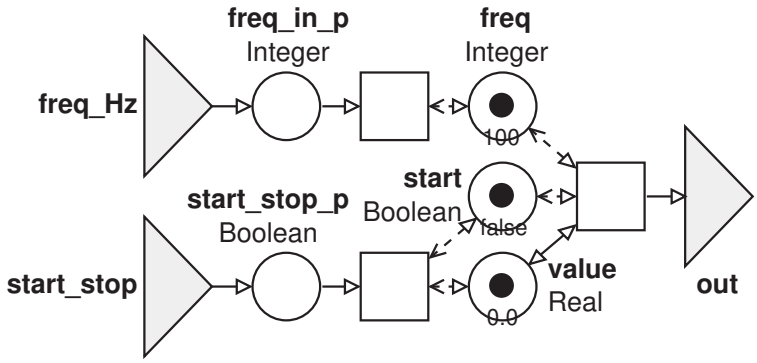

FIGURE 93. The *Prog_Oscillator* class

7.8 Summary

The previous sections describe only a small number of the steps required to design a telephone - the complete design being beyond the scope of this monograph. To complete the example all system components need to be further refined to a stage where any further refinement would imply an implementation technology. Once the full functionality has been defined, implementation options may be explored. This involves trial partitioning using the model's functionality and constraints to suggest implementation options. The result can then be refined, with the inclusion of architectural features. In addition any heterogeneous implementation suggestions need to be refined with the inclusion of interfaces, which ideally would also be described with a kernel language based formalism. From such partitioned models implementation code is produced and analysed to ascertain the quality of the proposed solution.

Even for such a simple system the amount of information required to fully describe the design is large and difficult to interpret and understand. The environment *Code Sign* has been created to implement the approach described in this monograph and was used to support the design of the telephone described in this example. *Code Sign* provides a way of conveying the design of an embedded system designed using the proposed approach. It includes capabilities that allow the object oriented class hierarchy to be navigated and to allow classes to be instantiated. The simulation capabilities allow the dynamic behaviour to be animated and more easily understood than a static paper based documentation.

CHAPTER 8 *Conclusions*

In this monograph a new approach for the design of embedded systems is presented. This approach is based on a formal kernel language augmented with object oriented concepts. The formal kernel language is a class of high level time Petri nets which offers the following benefits:

- intuitive graphical representation;
- simple model of concurrency and synchronisation;
- operational semantics that support simulation / execution;
- time model directly supporting the modelling of real time systems;
- proven analysis techniques.

The inclusion of object oriented concepts provides facilities for creating components that encapsulate behaviour and state and thereby supporting:

- abstraction;
- refinement;
- encapsulation;
- reuse and sharing.

Components are described with classes that define how inheritance properties are propagated and provide mechanisms for configuration, reuse and sharing. A system is a composition of communicating components at every level of abstraction. During the design of an embedded system classes are defined, refined, reused and configured. Refinement is supported by substi-

tuting less refined components with components containing an increased amount of detail. Abstraction and refinement allow complex systems to be modelled and successively refined towards an implementation.

The use of this class of Petri nets allows relevant aspects of hardware / software codesign to be addressed. These include:

- the use of application specific formalisms;
- the ability to parameterise or configure models;
- the support for design exploration.

Other approaches use a number of domain specific formalisms with which different aspects of a systems behaviour are modelled. These formalisms may be wholly formal methods or a combination of formal methods and informal ones. They often neglect the question of the semantics of the combination of formalisms within one specification.

Unlike these approaches that use special languages to represent other formalisms this approach uses the same object oriented time Petri net kernel language. This is beneficial as only a single modelling language is involved in modelling both a system and a formalism. In addition the semantics of combining multiple formalisms are completely given by the underlying kernel language. The flexibility supported by the formal language is available to all other kernel based formalisms as they are also object oriented and can be refined and reused in the same manner as the kernel language.

Having defined a flexible and powerful modelling language a methodology is presented for the design of embedded real time systems. This methodology does not replace existing traditional methodologies for hardware / software codesign but complements the strengths of these approaches by effectively adding an extra level of abstraction to better cope with the increasingly heterogeneous nature and complexity of today's systems.

The evolution of a simple telephone model in chapter 7 illustrates this approach. The familiar example of a telephone allows important features, such as support for multiple formalisms, configuration and refinement to be applied and demonstrated.

Appendices

A Implementation

The approach presented in this monograph is realised in the *Code Sign* tool. Code Sign has been designed and implemented by the author as an environment to test the practicality of concepts envisaged for the design of embedded systems. Code Sign was designed using the Booch method and implemented in Smalltalk (Visualworks 2.5). All the examples in this monograph have been modelled and simulated using Code Sign and from which figures in encapsulated Postscript format were generated.

The proposed approach is based on a kernel language. This kernel language is based on high level time Petri nets and object oriented concepts. It provides the capability of either directly modelling embedded systems (parts of or complete systems) or supporting other formalisms, in which embedded systems can also be modelled. The object oriented environment is used to create classes which describe how components (objects) are created (instantiated) and interact. Classes define how inheritance properties are propagated and provide mechanisms for refinement, configuration, reuse and sharing. The elements of high level time Petri nets are described with classes and form the primitive classes upon which all other classes are based.

The following sections give an overview of the key architectural components of Code Sign and their realisation.

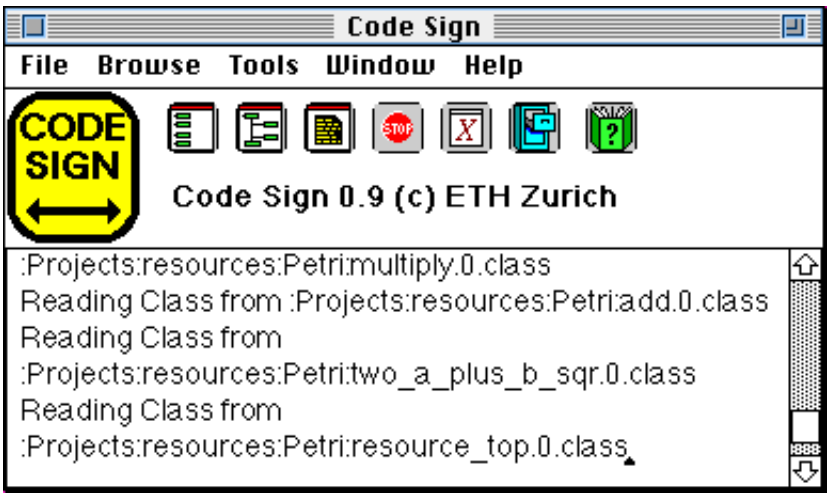

FIGURE 94. The Code Sign Launcher - provides access to all browsers and tools

A.1 Language

The textual representation of the Code Sign kernel language is used to provide permanent storage of models. It completely describes all aspects of all components, including the graphical representation used within a model, the textual functions and component inscriptions.

This language is compiled by Code Sign whenever a project or class is opened and whenever the user changes a function or element inscription. The resulting parse tree is used to create a Smalltalk object structure representing the project or class. Branches of the parse tree are stored in the Smalltalk objects representing functions and element inscriptions, and are used for presentation and code generation. Source code is never stored within Code Sign, being regenerated upon demand from the parse tree.

Similarly, from every project or class a textual representation can be generated and stored in the host file system. The syntax of this language is defined in the section ’Language Syntax’ on page 181.

A.2 Project

Code Sign supports the creation of projects as a means of grouping prerequisite projects and classes together, constituting a design. Projects sort classes with respect to their interdependencies, ensuring that classes are reopened in the correct order.

Figure 95 shows the graphical browser associated with a project. In this browser projects and classes are listed. The circular icon indicates that the corresponding entry is *folded* - effectively hiding prerequisite projects and classes. Classes and projects can be added to and deleted from a project either using the appropriate menu item or using the drag and drop capabilities of Code Sign.

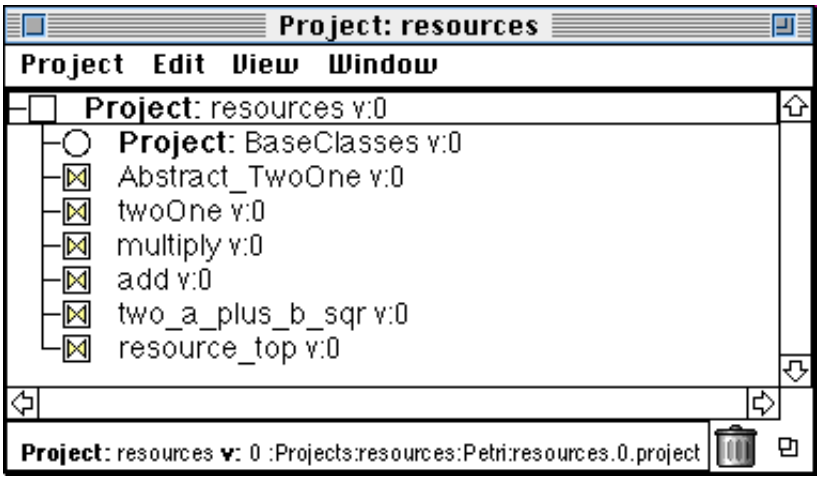

FIGURE 95. The project browser

The textual representation of a project is shown below:

```
PROJECT resources.0 DIRECTORY ':Projects:resources:Petri:'
DATE 16 June 1996 4:10:40
STATE unmodified
PREREQUISITES
{PROJECT BaseClasses.0 DIRECTORY
                    ':Projects:BaseClasses:'
CLASS Abstract_TwoOne.0 DIRECTORY
                    ':Projects:resources:Petri:'
CLASS twoOne.0 DIRECTORY
```

```
                    ':Projects:resources:Petri:'
CLASS multiply.0 DIRECTORY
                    ':Projects:resources:Petri:'
CLASS add.0 DIRECTORY
                    ':Projects:resources:Petri:'
CLASS two_a_plus_b_sqr.0 DIRECTORY
                    ':Projects:resources:Petri:'
CLASS resource_top.0 DIRECTORY
                    ':Projects:resources:Petri:'}
```

A.3 Class

Classes have a name and a version number providing a unique reference. In design exploration new versions of classes are created and modified. These modified classes are used to model further aspects of a design. If this is unsuccessful, back tracking facilities unwind the system design to a previous state (a set of classes with specific versions). A class is stored on the host file system as an individual file, which can optionally be placed under the control of an external version management system.

Classes define a structure of interconnected (embedded) components consisting of kernel language (primitive) and structured components. The class may directly reference another class, representing an embedded component, or indirectly using a function whose execution during the instantiation process returns the component to embed.

Functions are associated with a class and are therefore common to all components (instances) of the class. Functions are used for the configuration of components, to support the modelling environment (by implementing the graph grammar productions used to realise other formalisms) and as data transformations used by a model to implement its functionality.

Figure 96 shows the function browser opened on a particular project. It includes a class hierarchical list pane showing all the classes included in a project and their parent / child relationships. Classes in grey are not referenced by the project and either refer to primitive classes or indicate an error. Selecting a class in the class hierarchy list will generate a function list in the

pane below and selecting a function from this list will show its code in the editor pane, generated from its parse tree. This code may be edited and compiled by pressing the compile button. Any errors will be shown directly in the editor pane. Functions can be added or deleted from a class using menu functions. Changing a function's signature and compiling it will also result in a new function being added to the class.

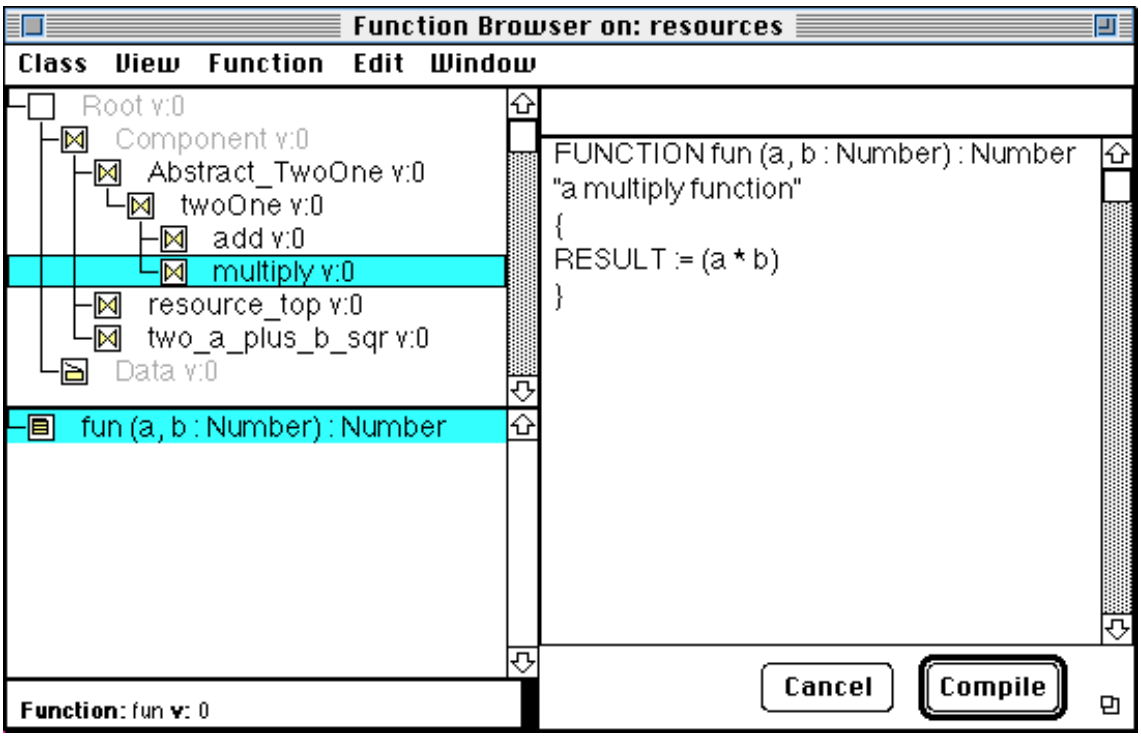

FIGURE 96. The function browser showing the class hierarchy, functions associated with the selected class and selected function code.

Figure 97 shows the class browser. In this browser selecting a class from the class hierarchy list will cause it to be instantiated. The instantiation hierarchy is shown in the pane below. There is an important distinction between the class hierarchy which shows the inheritance relationship between classes and the component hierarchy which shows the structure of a component. Selecting a component in the component hierarchy list will open an appropriate editor.

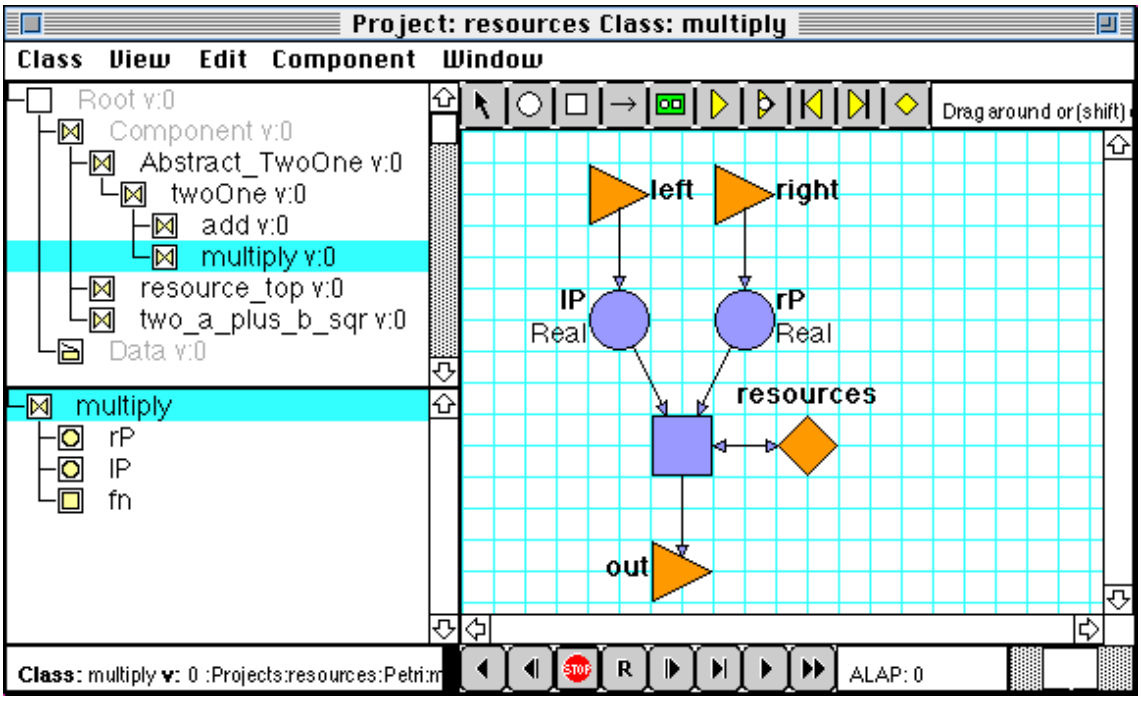

FIGURE 97. The class browser showing the class hierarchy, component hierarchy and graphical representation of the selected component

A.4 Component

Selecting a class in the class browser will instantiate it. This instantiation process is a transformation of Smalltalk class objects to Smalltalk component objects. The result of the instantiation process is a hierarchical structure of components, each of which is represented by a Smalltalk object. This instantiation process may either simply instantiate the class referenced as an embedded component or, alternatively, the result of a function execution and insert it the parent component.

Many traditional object oriented languages and environments exist where a class is edited by modifying its textual representation. In Code Sign editing operations are always carried out on components. Components have a static structure defined by the class and which is affected by editing. As a result of a class being edited all components (instances) of the class will be updated. Components also have state information which can be edited. This state

information is only valid for a particular simulator state and does not have a corresponding class representation.

Figure 98 shows the class browser, where a class with embedded components has been instantiated. In the component hierarchy list one such component is unfolded revealing its primitive kernel components.

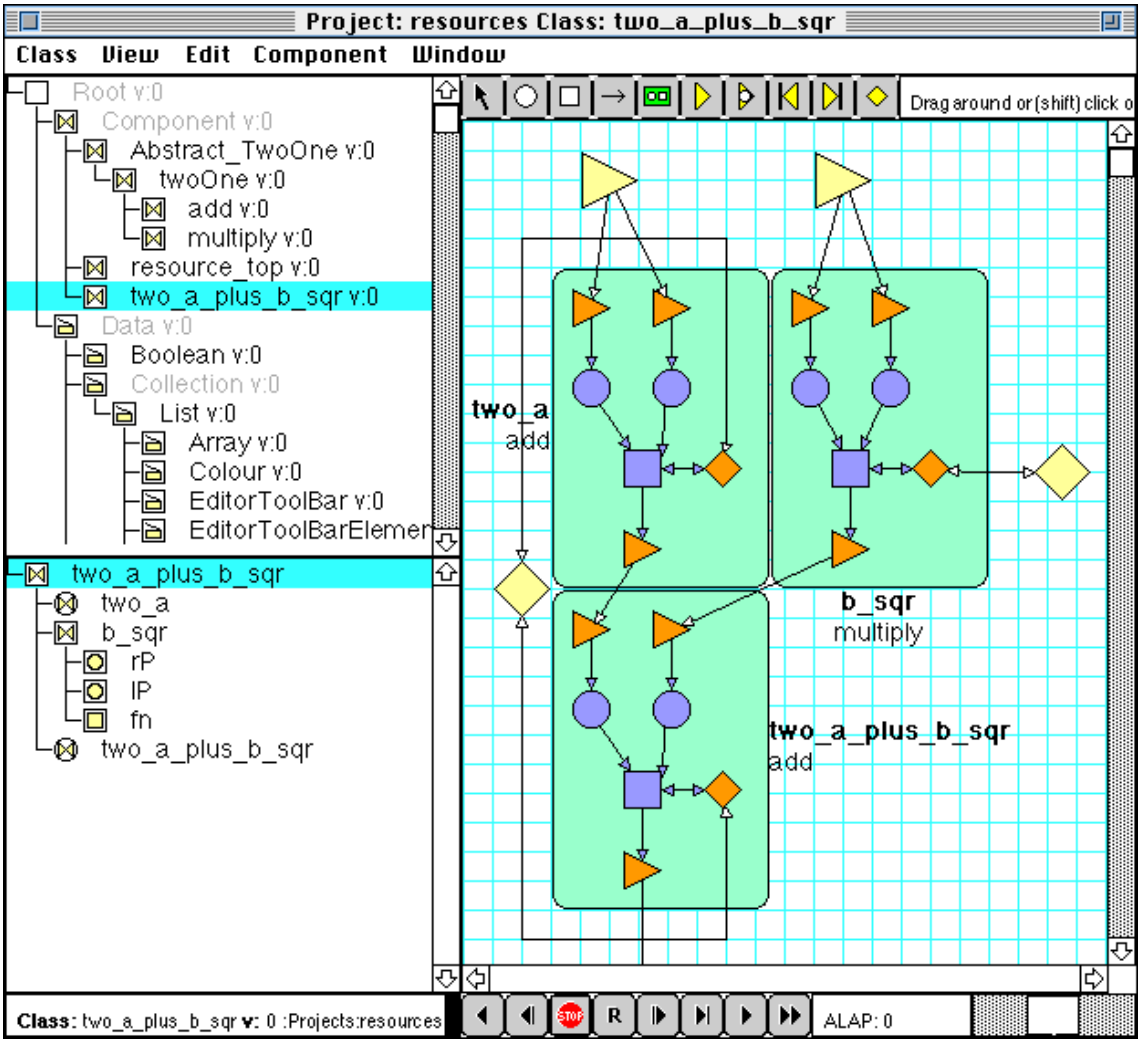

FIGURE 98. The class browser showing the class hierarchy, component hierarchy and graphical representation of a component containing non primitive components

Selecting a component from the component hierarchy list will cause an editor to be inserted into the editor pane. The particular editor used depends on the class of the selected component.

The graphics editor has special capabilities and can be configured by functions defined in the selected component class. These functions define editor buttons and associated operations. This is used in the support of other formalisms. Additionally each embedded component has functions which determine whether it is displayed and if so, its graphical representation. This also includes a graphical representation of the state of an object, such as whether a transition is enabled or if a particular state in a finite state machine is active.

Integrated into the class browser are the simulator controls. The simulator component of the class browser is very simple, in effect only maintaining a record of the simulation time and scheduling strategy. It operates by delegating responsibilities to the selected component which in turn may delegate responsibilities to its embedded components. Thus a component determines how it will react to a request from the simulator. Primitive components representing the high level time Petri net kernel implement the appropriate simulation functions and ultimately define the semantics of all structured components that have delegated their simulation functions. This strategy allows other formalisms to override the default simulation functions, perhaps implementing co-simulation facilities or defining new primitives.

Figure 99 shows the place editor. This editor includes sub-editors allowing the current tokens, token history, initial tokens and token type to be displayed and changed. It also allows the name, place capacity and text position to be changed.

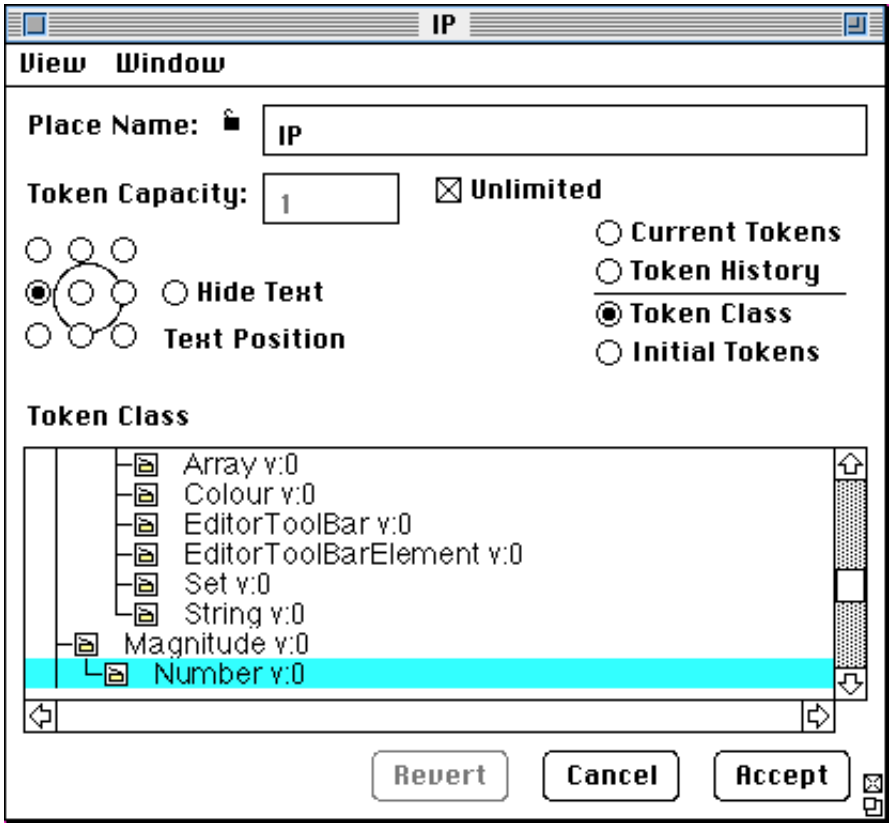

FIGURE 99. The place editor where place attributes and current tokens, initial tokens and token class are displayed and edited

Figure 100 shows the component configuration editor. In this editor a class can be defined which will be instantiated to represent the component. It is also possible to define a function which will be executed during the instantiation process to return the component.

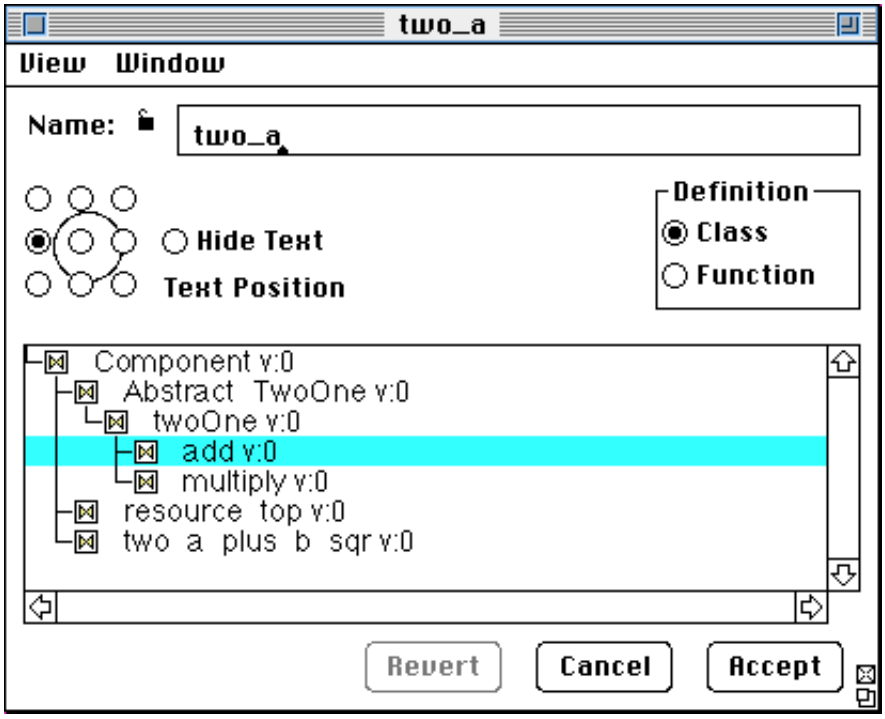

FIGURE 100. The component configuration editor where the name, text position and component class is displayed and changed

Most editors, including this one, have a revert button or menu function. If active this indicates that the component is inherited and its settings can be reset to the values defined by the parent class. All editors implement the inheritance rules allowing those actions which are applicable for the component.

In addition all editors can be opened outside the class browser allowing multiple windows to be open on multiple levels of system. All windows will always show the current state of a displayed component.

Code Sign is currently being used at the ETH for teaching and research. This research includes distributed simulation of Code Sign models, the efficient generation of C and VHDL, analysis techniques, partitioning and interface synthesis.

B Language Syntax

The kernel language presented in this monograph can be described textually. The Backus-Naur Form (BNF) is used to specify the syntax with the following notational conventions:

[]	Items contained within square brackets are optional.
*	An asterisk after an item indicates a sequence of zero or more occurrences of that item.
+	A plus sign after an item indicates a sequence of one or more occurrences of that item.
|	A vertical indicates two or more alternative production rules.
( )	Items in brackets are grouped, allowing a modifier such as an asterisk or plus sign to be applied to all group terms.
<terminal>	Terminal productions are shown in angle brackets.
‘symbol’	A item between apostrophes indicates a reserved word or symbol.
“comment”	An item between double apostrophes is regarded as a comment and has no semantic meaning.

B.1 General

```
<name>          : [A-Za-z][A-Za-z0-9_]*
<char>          : $[!-\~]
<number>        : [0-9]+
<string>        : \'(~[\']|\'\'|[\s\t\r])*\'
<binaryname>    : [! #%-& \* \+ \- / <-@ \\ ^ \|] +
<hex>           : \\[hH][0-9A-Fa-f]+
<dchar>         : \\[0-9][0-9][0-9]
<hchar>         : \\[xX][0-9A-Fa-f][0-9A-Fa-f]
<comment>       : \"(~[\"]|\"\"|[\s\t\r])*\"
<space>         : [\s\t\r]+
```

B.2 Class

```
Class                : 'CLASS' ClassDescr ParentName Dir
                       ClassInfos
                     | 'CLASS' ClassDescr ParentName
                       CollectionClassInfo Dir ClassInfos

ParentName           : ':' ClassDescr
                     | "nothing"
CollectionClassInfo  : '(' ClassDescr ',' Number ')'
                     | '(' ClassDescr ')'
Dir                  : 'DIRECTORY'  String
ClassInfos           : ClassInfo ClassInfos
                     | ClassInfo
ClassInfo            : Date
                     | State
                     | Field
                     | Place
                     | Transition
                     | Function
                     | Export
                     | Connect
                     | Ginfobox
                     | Ginfoline
Date                 : 'DATE' Day Month Year
                       Hours':'Minutes':'Seconds
Day                  : Number
Month                : Name
Year                 : Number
Hours                : Number
Minutes              : Number
Seconds              : Number
State                : 'STATE' Name
Field                : 'FIELD' Name ':' ClassDescr
                     | 'FIELD' Name ':' FunctionCall
                     | 'FIELD' Name ':' AbstractComponent

ClassDescr           : Name'.'Number
AbstractComponent    : '{' Class '}'
Ginfobox             : 'GINFOBOX' Name Number Name Number
                       Number Number Number
Ginfoline            : 'GINFOLINE' Name Number Real Real
                       EmptyNumberList
EmptyNumberList      : NumberList
                     | "nothing"
```

B.3 Export (Interface)

```
Export          : 'EXPORT' Name Name
```

B.4 Connection

```
Connect         : 'CONNECT' Name  Name',' Name Name','
                  Name
                | 'CONNECT' Name '(' Expr ')' Name','
                  Name Name',' Name
                | 'CONNECT' Name ':' ClassDescr  Name','
                  Name Name',' Name
                | 'CONNECT' Name ':' ClassDescr '(' Expr
                  ')' Name',' Name Name',' Name
```

B.5 Place

```
Place           : 'PLACE' Name ':' PlaceParam
PlaceParam      : ClassDescr
                | ClassDescr CollectionList
                | ClassDescr '(' Number ')'
                | ClassDescr '(' Number ')'
                  CollectionList
```

B.6 Transition

```
Transition      : 'TRANSITION' Name '(' TransParamList
                  ')'TransitionCond TransitionAct
                  TransitionDelay
TransParamList  : TransDecl ',' TransParamList
                | TransDecl
                | "nothing"
TransDecl       : Name Name
TransitionCond  : 'CONDITION' '{' Expr '}'
                | 'CONDITION' '{' '}'
                | 'CONDITION' '{' CommentList Expr '}'
                | 'CONDITION' '{' CommentList '}'
                | "nothing"
TransitionAct   : 'ACTION' InitialTransCodeBlock
                | "nothing"
```

```
TransitionDelay      : 'DELAY' '{' Expr ',' Expr '}'
                     | 'DELAY' '{' CommentList Expr ','
                       CommentList Expr '}'
                     | 'DELAY' '{' Expr ',' CommentList Expr
                       '}'
                     | 'DELAY' '{' CommentList Expr ',' Expr
                       '}'
                     | 'DELAY' '{' '}'
                     | "nothing"
InitialTransCodeBlock :'{' VarList CB '}'
                     | '{' CB '}'
                     | '{' CommentList '}'
                     | '{' '}'
```

B.7 Function

```
Function             : 'FUNCTION' BinaryName '(' FuncDefList
                       ')' ':' FuncResClass InitialCodeBlock
                     | 'FUNCTION' Name '(' FuncDefList ')'
                       ':' FuncResClass InitialCodeBlock
                     | 'FUNCTION' Name ':' FuncResClass
                       InitialCodeBlock
                     | 'FUNCTION' BinaryName '(' FuncDefList
                       ')' ':' FuncResClass CommentList
                       InitialCodeBlock
                     | 'FUNCTION' Name '(' FuncDefList ')'
                       ':' FuncResClass  CommentList
                       InitialCodeBlock
                     | 'FUNCTION' Name ':' FuncResClass
                       CommentList InitialCodeBlock
FuncDefList          : FuncDefItem ';' FuncDefList
                     | FuncDefItem{CSFuncDefList}
                     | "nothing"
FuncDefItem          : NameList ':' Name
FuncResClass         : Name
```

B.8 Expression

```
Expr                 : Atom BinaryName Expr
                     | '#' String'#' "Primitive"
                     | Atom
FunctionCall         : Name '.' FuncCall
```

```
                    | Name '.' Name
                    | CollectionIndex '.' FuncCall
                    | CollectionIndex '.' Name
                    | Name
                    | FuncCall
FuncCall            : Name '(' FuncCallPList ')'
FuncCallPList       : ExprList
CollectionList      : '[' ExprList ']' CollectionList
                    | '[' ExprList ']'
CollectionIndex     : FunctionCall CollectionList
ExprList            : Expr ',' ExprList
                    | Expr
                    | "nothing"

Atom                : '(' Expr ')'
                    | FunctionCall
                    | Number
                    | Real
                    | String
                    | CollectionList
                    | CollectionIndex
                    | <char>
                    | <hex>
                    | <dchar>
                    | <hchar>
```

B.9 Code

```
Var                 : 'VAR'  NameList ':' Name
VarList             : Var VarList
                    | Var
InitialCodeBlock    : '{' VarList CB '}'
                    | CodeBlock;
CodeBlock           : '{' CB '}'
                    | '{' CommentList '}'
Stmt                : Name ':=' Expr
                    | CollectionIndex ':=' Expr
                    | FunctionCall
                    | '#' String'#' "Primitive"
                    | '##' String '##' "Pragma"
CB                  : CodeStruct CB
                    | Stmt CB
                    | CodeStruct
```

```
                        | Stmt
                        | CommentList CodeStruct CB
                        | CommentList Stmt CB
                        | CommentList CodeStruct
                        | CommentList Stmt
CodeStruct              : 'IF' Expr 'THEN' CodeBlock
                        | 'IF' Expr 'THEN' CodeBlock 'ELSE'
                          CodeBlock
                        | 'WHILE' Expr 'DO' CodeBlock
                        | 'FOR' Stmt 'TO' Expr 'DO' CodeBlock
```

B.10 Sundries

```
Name                    : <name>
String                  : <string>
Number                  : <number>
Real                    : Number '.'Number
NumberList              : Number  NumberList
                        | Number
BinaryName              : <binaryname>
NameList                : Name ',' NameList
                        | Name
Comment                 : <comment>
CommentList             : Comment CommentList
                        | Comment
```

Literature

[1] van der Aalst W.M.P, Interval Timed Coloured Petri Nets and their Analysis, In LNCS[1] 691, Springer Verlag 1993

[2] Ada 9X Project Office, Ada 9X Requirements. Ada 9X Project Report. Washington, DC: Office of the Under Secretary of Defense for Aquisition, 1990.

[3] Agerwala T., Flynn M., Comments on Capabilities, Limitations and 'Correctness' of Petri Nets, 1st Annual Symposium on Computer Architecture, ACM 1973

[4] Ajmone Marsen M., Balbo G., Conte G., Donatelli S., Franceschinis G., Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing, 1995

[5] Alur R., Henzinger T.A. Logics and Models of Real Time: A survey. In de Bakker J.W., Huizing C., de Roever W.P., Rozenberg G., editors, Real-Time: Theory in Practice, REX Workshop Proceedings, 1992

[6] Andrews D., Ince D., Practical Formal Methods with VDM, McGraw Hill,1991

1. Lecture Notes in Computer Science

[7] Antoniazzi S., Balboni A. Fornaciari W., Sciuto D., A Methodology for Control-Dominated Systems Codesign, Preceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994

[8] Asar P., Towards a Multi-Formalism Framework for Architectural Synthesis: the ASAR Project, Preceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994

[9] Baldassari M., Bruno G., An Environment for Object-Oriented Conceptual Programming Based on PROT Nets, In LNCS 340, Advances in Petri Nets 1988, Springer-Verlag, 1988

[10] Barros E., Sampaio A., Towards Provably Correct Hardware/ Software Partitioning Using Occam, Preceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994

[11] Battiston E., de Cindio F., Class Orientation and Inheritance in Modular Algebraic Nets, Proceedings of IEEE International Conference on Systems, Man and Cybernetics 2, Le Touquet, France, IEEE 1993

[12] Battiston E., de Cindio F., Mauri G., OBJSA Nets: A Class of High-level Nets having Objects as Domains, In LNCS 340, Advances in Petri Nets 1988, Springer Verlag 1988

[13] Baumgartner B., Petri Netze Grundlagen und Anwendungen, BI Wissenschaftsverlag ISBN 3-411-14291-X, 1990

[14] Bellini A., Bigalli O., Pirri F., Modular Statecharts for Designing Parallel Controllers, More FPGAs, Moore W., Luk W. Eds, Abingdon EE&CS Books, ISBN 0-9518453-1-4, 1994

[15] Benders L.P.M, Analysis of Real-Time Embedded Systems for Co-Design, Sixth IEEE International Workshop on Rapid System Prototyping, ISBN 0-8186-7100-9, 1995

[16] Benner Th., Ernst R., Könenkamp I., Schüler P., Schaub H.-C., A Prototyping System for Verification and Evaluation in Hardware-Software Cosynthesis, Sixth IEEE International Workshop on Rapid System Prototyping, ISBN 0-8186-7100-9, 1995

[17] Bernardinello L., De Cindio F., A Survey of Basic Net Models and Modular Net Classes, LNCS 609, Springer Verlag 1992

[18] Berry G., Couronné P., Gonthier G., The Synchronous Approach to Reactive and Real-Time Systems. IEEE Proceedings, 79, September 1991

[19] Berthomieu B., Diaz M., Modeling and Verification of Time Dependent Systems Using Time Petri Nets, IEEE Transactions on Software Engineering Volume 17 Number 3, 1991

[20] Best E., Fernandez C., Notations and Terminology on Petri Net Theory, Gesellschaft für Informatik, Special Interest Group 'Petri Nets and Related System Models' Newsletter 20, 1985

[21] Blaschek G., Object-Oriented Programming with Prototypes, Springer Verlag, 1994

[22] Booch G., Object Oriented Analysis and Design With Applications, The Benjamin/Cummings Publishing Company Inc. Redwood City, Calif. 1994

[23] Bortolazzi J., Tanurhan Y., Müller-Glaser K.D., Toward an Environment for System Specification and Design, Codesign Computer-Aided Software/Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1994

[24] Brandt S., Lehrmann Madsen O., Object-Oriented Distributed Programming in BETA, Computer Science Department, Aarhus University, 1994

[25] Bucci G., Vicario E., Compositional Validation of Time-Critical Systems Using Communicating Time Petri Nets, IEEE Transactions on Software Engineering Vol. 21, No. 12., 1995

[26] Buchenrieder K. Sedlmeier A., Veith C., CODES A Framework for Modeling Heterogeneous Systems, Codesign Computer-Aided Software/Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1994

[27] Buchenrieder K., Rozenblit J.W, Codesign: An Overview, Codesign Computer-Aided Software/Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1995

[28] Buchs D., Guelfi N., Open Distributed Programming Using the Object Oriented Specification Formalism CO-OPN, Proceedings of 5th International Conference on Puting into Practice Methods and Tools for Information System Design, Nantes, 1992

[29] Buck J., A Dynamic Dataflow Model Suitable for Effiecient Mixed Hardware and Software Implementations of DSP Applications, IEEE 1994

[30] Bütler B., Esser R., Mattmann R., A Distributed Simulator for High Order Petri Nets, In LNCS 674, Advances in Petri Nets 1990, Springer Verlag 1990

[31] Caspi P., Pilaud D., Halbwachs N., Plaice J., LUSTRE: A Declaritive Language for Programming Synchronous Systems, In proceedings of the 14th ACM Symposium on Principles of Programming Languages, January 1987

[32] CCITT Recommendation Z.100: Specification and Description Language SDL (Blue Book), General Secretariat - Places des Nations, CH1211 Geneva 20, 1988

[33] Chang W.-T. Kalavade A., Lee E.A. Effective Heterogenous Design and Co-Simulation, Nato Advanced Study Institute, Hardware/ Software Co-Design, Tremezzo Italy, June 1995

[34] Chiodo M., Giusto P., Hsieh H., Jurecska A., Lavagno L., Sangiovanni-Vincentelli A., A Formal Methodology for Hardware/ Software Codesign of Embedded Systems. IEEE Micro, August 1994

[35] Christensen S., Toksvig J., DesignBeta V2.0.1 BETA Code-Segments in CP-nets, University of Aarhus, Internal report, 1993

[36] Clarke E.M., Emerson E.A., Sistla A.P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic, ACM Transactions on Programming Languages and Systems, April 1986

[37] Computer, An Invitation to Formal Methods, IEEE Computer, April 1996

[38] Dähler J., Ein Werkzeug für den Entwurf verteilter Systeme auf der Basis erweiterter Petri-Netze, DISS. ETH Nr. 8770. Phd Thesis Swiss Federal Institute of Technology, 1989

[39] Dähler J., Gerber P., Gisiger H.-P., Kündig A., A Graphical Tool for the Design and Prototyping of Distributed Systems., ACM Software Engineering Notes. Vol 12 No. 3, 1987

[40] De Micheli G., Ku D., Mailhot F., Truong T. Stanford University, The Olympus Synthesis System, IEEE Design and Test of Computers, 1990

[41] Dennis J.B. Data Flow Supercomputers, Computer, vol 13, no. 11, November 1980

[42] Dittrich G., Modeling of Complex Systems Using Hierarchical Petri Nets, Codesign Computer-Aided Software/Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1994

[43] Dodge C.J., Undrill P.E., Allen A.R., Ross P.G.B, Application of Z in Digital Hardware Design, IEE Proceedings Computers and Digital Techniques Vol 143, No. 1, 1996

[44] Edwards M.D., Hardware/Software Codeign: Experiences with Languages and Architectures, Codesign Computer-Aided Software/ Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1994

[45] Eles P., Peng Z., Doboli A., VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment, Preceedings of the Third International Workshop on Hardware/ Software Codesign, Grenoble France, 1994

[46] Ernst R., Henkel J., Benner Th, TU Braunschweig, Hardware-Software Cosynthesis for Microcontrollers, IEEE Design and Test of Computers, 1993

[47] Fehling R., A Concept of Hierarchical Petri Nets with Building Blocks, In LNCS 674, Advances in Petri Nets 1993, Springer-Verlag, 1993

[48] Felder M., Ghezzi C., Pezzé M., Analyzing Refinements of State Based Specifications: The Case of TB nets. In Proceeding of the first International Symposium on Software Testing and Analysis (ISSTA), June 1993

[49] Felder M., Ghezzi C., Pezzé M., High Level Timed Petri Nets as a Kernel for Executable Specifications, 1993

[50] Fichtner W., Design of VLSI Systems, In Embedded Systems: New Approaches to Their Formal Description and Design, Kündig A., Bührer R.E, Dähler J., (Eds) LNCS 284, Springer-Verlag 1986

[51] Fierz H., Müller H., Netos S. CIP-Communicating Interacting Processes: A Formal Method for the Development of Reactive Systems. In SAFECOMP '93 Proceedings of the 12th International Conference on Computer Saftey, Reliability and Security, Poznan-Kiekrz, Poland October 1993. Edited Janusz Gorski, Springer Verlag, 1993

[52] Forrest J., Implementation-Independent Descriptions Using an Object-Oriented Approach, Codesign Computer-Aided Software/ Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1994

[53] Gajski D. D., Vahid F., University of California, Specification and Design of Embedded Hardware-Software Systems, IEEE Design and Test of Computers, 1995

[54] Gajski D. D., Vahid F., Narayan S., Gong J., Specification and Design of Embedded Systems, PTR Prentice Hall Englewood Cliffs, 1994

[55] Genrich H.J., Lautenbach K., System Modelling with High-Level Petri Nets, Theoretical Computer Science 13, North Holland, 1981

[56] Ghezzi C. Morasca S., Pezzè M., Validating Timing Requirements fot TB Net Specications, Dipartimento di Elettronica e Informazione. Politecnico di Milano, 1994

[57] Ghezzi C., Mandrioli D., On Eclectic Specification Environments, Advances in Object Oriented Software Engineering, Prentice Hall, D. Mandrioli and B. Meyer ISBN 0-13-006578-1, 1992

[58] Gisiger H.-P., Kausale Beschreibung von Systemen, TIK-Schriftenreihe Verlag der Fachvereine Zürich ISBN 3 7281 2050 2, 1993

[59] Goldberg A., Robson D., Smalltalk-80 The Language and its Implementation, Addison-Wesley ISBN 0-201-11371-6, 1983

[60] Green P., Rushton P., Beggs R., An Example of Applying the Codesign Method Moose, Preceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994

[61] Habel A., Hyperedge Replacement: Grammars and Languages, Lecture Notes in Computer Science 643, Springer-Verlag, ISBN 3-540-56005-X , 1992

[62] Harel D., StateCharts: A Visual Formalism for Complex Systems, Science of Computer Programming 8 (1987) 231-274 North-Holland, 1987

[63] Hoare C.A.R., Communicating Sequential Processes. Prentice Hall International, 1995

[64] Hsieh H., Lavagno L., Sangiovanni-Vincentelli, A.Embedded System Co-Design: Synthesis and Verification, Nato Advanced Study Institute, Hardware/Software Co-Design, Tremezzo Italy, June 1995

[65] IEEE, IEEE Standard VHDL Language Reference Manual, Institute of Electrical and Electronic Engineers,1988

[66] INMOS Limited, Occam 2 Reference Manual, Prentice Hall International Series in Computer Science, 1988

[67] Ismail T. B., Abid M. Jerraya A., COSMOS: A CoDesign Approach for Communicating Systems, Preceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994

[68] Jacobson I., Object-Oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley, 1992

[69] Jensen K., Coloured Petri Nets: A High Level Language for System Design and Analysis, Advances in Petri Nets 1990, G. Rozenberg (ed), LNCS 483, Springer-Verlag, 1990

[70] Jensen K., Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use - Volume 1: Basic Concepts, EATCS Monographs in Computer Science, Vol. 26, Springer-Verlag, 1990

[71] Johnson W.L., Feather M.S., Harris D.R., Representation and Presentation of Requirements Knowledge, IEEE Transactions on Software Engineering 18(10), October 1992

[72] Kalavade A., Lee E.A., UC Berkeley, A Hardware-Software Codesign Methodology for DSP Applications, IEEE Design and Test of Computers, 1993

[73] Koymans R., de Roever W.P., Examples of a Realtime Temporal Logic Specification, LNCS 207. Springer Verlag, 1985

[74] Kumar, Aylor, Johnson, Wulf, Object-Oriented Techniques in Hardware Design, IEEE Computer P64 - 70, 1994

[75] Kündig A.T., A Note on the Meaning of Embedded Systems, In Embedded Systems: New Approaches to Their Formal Description and Design, Kündig A., Bührer R.E, Dähler J., (Eds) LNCS 284, Springer-Verlag 1986

[76] Lakos C.A., Christensen S., A General Systematic Approach to Arc Extensions fo Coloured Petri Nets, In LNCS 815, Advances in Petri Nets 1994, Springer Verlag ISBN 3-540-58152-9, 1994

[77] Lakos C.A., Object Petri Nets - Definition and Relationship to Coloured Nets, Department of Computer Science, University of Tasmania, 1994

[78] Lakos C.A., Pragmatic Inheritance Issues for Object Petri Nets, Department of Computer Science, University of Tasmania, 1995

[79] Lakos C.A., Keen C.D., Applying Invariant Analysis to Modular Petri Nets, Department of Computer Science, University of Tasmania,

[80] Lakos C.A., Keen C.D., LOOPN++: A New Language for Object-Oriented Petri Nets, Department of Computer Science, University of Tasmania, 1994

[81] Lakos C.A., Keen C.D., Modelling a Door Controller Protocol in LOOPN, Proceedings of the 10th European Conference on the Technology of Object-oriented Languages and Systems, Versailles, Prentice Hall 1993

[82] Lakos C.A., Keen C.D., Modelling Layered Protocols in LOOPN, Proceedings of Fourth International Workshop on Petri Nets and Performance Models, Melbourne, Australia, 1991

[83] Lee E.A., Consistency in Dataflow Graphs, IEEE Transactions on Parallel and Distributed Systems, Vol 2, No. 2 April 1991.

[84] Lee E.A., Messerschmitt D.G., Synchronous Data Flow, Proceedings of the IEEE, Vol 75 No. 9, September 1987

[85] Ludewig J. Practical Methods and Tools for Specification, In Embedded Systems: New Approaches to Their Formal Description and Design, Kündig A., Bührer R.E, Dähler J., (Eds) LNCS 284, Springer-Verlag 1986

[86] Maggiolo-Schettini A., Peron A., Semantics of Full StateCharts Based on Graph Rewriting, In LNCS 776 Graph Transformations in Computer Science, Springer Verlag

[87] Merlin P., A Study of the Recoverability of Computer Systems, Thesis, Department of Computer Science, University of California, Irvine, 1974

[88] Merlin P., Faber D.J., Recoverability of Communication Protocols, IEEE Transactions on Communications, Vol. Com-24, No. 9, September 1976

[89] Murata T., Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, Vol 77, No. 4, 1989

[90] Nagl M., A Tutorial and Bibliographical Survey on Graph Grammars, Bericht Nr. 69/78 Abteilung Informatik, Universität Dortmund, 1978

[91] Niskier C., Maibaum T., Schwabe D., A Look Through PRISMA: Towards Pluralistic Knowledge-Based Environments. In Proceedings of the fifth International Workshop on Software Specification and Design, May 1989

[92] Nuseibeh B., Kramer J., Finkelstein A., Expressing the Relationships Between Multiple Views in Requirements Specification. In Proceedings of the 15th International Conference on Software Engineering, May 1993

[93] Oswald H., Esser R., Mattmann R., An Environment for Specifying and Executing Hierarchical Petri Nets, 12th International Conference on Software Engineering, Nice, France, 1990

[94] Peterson J.L., Petri Net Theory and the Modelling of Systems, Englewood Cliffs, N.J., 1981

[95] Petri C.A., Kommunikation mit Automaten, Dissertation der Hochschule Darmstadt, 1962

[96] Pezzé M., Cabernet: A Customizable Environment for the Specification and Analysis of Real-Time Systems, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1994

[97] Pulli P., Dähler J., Gisiger H.-P., Kündig A., Execution of Ward's Transformation Schema on the Graphic Specification and Prototyping Tool SPECS, CompEuro 88, Brussels, 11.-14.4.1988, 1988

[98] Ramchandani C., Analysis of Asynchronous Concurrent Systems by Timed Petri Nets, Massachusetts Institute of Technology, Project MAC, Technical report 120, February 1974

[99] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modeling and Design, Prentice Hall, 1991

[100] Selic, Gullekson, Ward, Real-Time Object-Oriented Modeling, Wiley ISBN 0-471-59917, 1994

[101] Spivey J.M., The Z Notation - A Reference Manual. Prentice Hall, 1989

[102] Starke P. H, Analyse von Petri-Netz-Modellen, B.G. Teubner Stuttgart, 1990

[103] Staunstrup B. The C++ Programming Language, Addison-Wesley, 1986

[104] Staunstrup J., Toward a Common Model of Software and Hardware Components, Codesign Computer-Aided Software/Hardware Engineering. Eds Rozenblit J., Buchenrieder K., IEEE Press ISBN 0-7803-1049-7, 1994

[105] Takach W., Wolf A., An Automation Model for Scheduling Constraints in Synchronous Machines. IEEE Transactions on Computers, 44, 1-12, January 1995

[106] Theißinger M., Stravers P., Veit H., Castle: An Interactive Environment for HW-SW Co-Design, Preceedings of the Third International Workshop on Hardware/Software Codesign, Grenoble France, 1994

[107] Ungar D., Smith R. B., Self: The Power of Simplicity, Proceedings of OOPSLA '87, 1987

[108] Vahid F., Narayan. S., Gajski D. D. University of California, SpecCharts: A VHDL Front-End for Embedded Systems, IEEE Transactions on Computer-Aided Design of Integerated Circuits and Systems, 1995

[109] Verkoulen P.A.C., Integrated Information Systems Design: An Approach Based on Object Oriented Concepts and Petri Nets, University of Technology: Eindhoven PhD Thesis, 1993

[110] Wang F., Mok A., Emerson E., Formal Specification of Asynchronous Distributed Real-Time Systems by APTL. In ICSE 14: Proceedings of the 14th International Conference of Software Engineering, May 1992

[111] Ward P.T., Mellor S.J., Structured Development for Real-Time Systems, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, 1985

[112] White I., Using the Booch Method: A Rational Approach, The Benjamin/Cummings Publishing Company Inc. Redwood City, Calif. ISBN 0-8053-0614-5?, 1994

[113] Yourdon E., Constantine L.L., Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice Hall Inc., Englewood Cliffs, 1979

Index

A

abstract data type 95
abstraction 11, 16, 52
ADA 36
ANSI C **35**, 63
application specific integrated
circuit 9
APTL 63
arc 34, **104**
ARIES 117

B

behaviour finite state machine 27
Beta 98
binding 77, 85, 91
Booch method **47**, 59, 171
boolean data flow 32

C

C++ 63, 96
Cabernet 64, 117
capacity function 68
CCITT 30, 38
class 95, 174
-hierarchy 135
-instance, see component
CLOWN 99
Code Sign 167, **171**
-kernel language 172
-project 173
codesign 10
-finite state machine 27
Communicating Interacting
Processes 29
Communicating Sequential
Processes 35
component 56, 100, **112**, 176
concurrency 10, 11
configuration 19, 52, **113**, 163
CO-OPN 99
COSYMA 35, **42**
C^x 35, 42

D

data flow
-boolean 32
-diagram 44
-graph 52
-synchronous, see synchronous
data flow
design exploration 20, 113
domain analysis 47

E

embedded component 145
embedded systems 6

embedding 144
encapsulation 11, 53
environment model 155
Esterel **29**, 63
extended syntax graphs 33

F
field programmable gate array 9
finite state machine **26**, 29, 52, 127, 128
-behaviour 27
-codesign 27
formal methods 63
functions 103

G
graph grammar 54, 115, **118**
-production 115, 118
-semantic **122**, 137, 148
-syntax **121**, 132, 148

H
hardware development 13
Hardware-C 35, 42
Harel 131
hierarchical flow graphs 33

I
inheritance 52, 53, 95, **103**
instantiation process 102
integrated circuit 13
interface generation 21

K
kernel component 115
kernel language 10, 54, **63**, 114, 148
-Code Sign 172
-syntax 181

L
LOOPN 99
LOOPN++ 99, 100
LUSTRE 63

M
marking 34
microcontroller 6
microprocessor 6
MOOSE 43, 59
multi-paradigm 18, 64
multiple formalisms 51, 54, **117**

O
Object Interaction Diagram 43
Object Modelling Technique 59
object orientation 11, **52**, 95, 143
OBJSA 99
Occam 35, 63
Olympus 42

P
partitioning 10, 15, 21
Petri nets **34**, 51, 64, **65**, 97
-hierarchical coloured 97, 99
-high level 34, **75**
-high level time 10, 54, 64, **82**, 91, 171
-object 99
-object oriented 98
-object oriented time 59, 121, 127, 148
-place-transition 66
-time 82, 119
-timed 82, 119
-with inhibitor arcs 72
place 34, **104**
place capacity 105
polymorphism 52, 107
PRISMA 117

process control system 7
Program State Machines 28
PROT Nets 98
Ptolemy **42**, 117

R

real time system 7
Real-time Object-Oriented Modelling **48**, 58
refinement 11, 18, 52, 108, 163
requirements 149
requirements analysis 47
reuse 52
ROOMCharts 48

S

scenario 49, 59, 151
semantic graph grammar **122**, 137, 148
sharing 52
SimCon object model 98
simulation 10
simulation backplane 53
Smalltalk 96, 171
software development 15
SpecCharts 28
Specification Description Language **30**, 38
StateCharts **28**, 48, 127, 144
  -class hierarchy 136
  -event mechanism 135
Structured Analysis 44
Structured Development for Real-Time Systems 45
subclassing 143
synchronisation 10
Synchronised Transitions 30
synchronous data flow **31**, 32, 57, 122, 123, 165
syntax graph grammar **121**, 132, 148
synthesis 10
system
  -design 47
  -model 16, 161
  -specification 149
  -structure 160

T

telephone 149
time 19
token 34
transition 34, **105**
transition instance 87

U

use case analysis 47, 49

V

VDM 63
VHDL 28, **36**, 63
VIEWPOINTS 117

W

weight function 68

Y

Y diagram **118**, 120, 124

Z

Z 63