

Prototyping Real-Time Systems

Author

Gary Clynch B.Sc.

Supervisor

Mr. Renaat Verbruggen

**Submitted to
The School of Computer Applications
Dublin City University
for the degree of
Master of Science**

January 1994

This is based on the candidate's own work

Acknowledgements

I would like to sincerely thank Renaat for all his help and guidance, Prof David Bustard for his helpful comments, my family and all my friends and fellow postgraduates at the School of Computer Applications for their encouragement and support I would especially like to thank my brother Conor for proof-reading this thesis

Table of Contents

	Page
1. Introduction	
1 1 Objective of Research	1
1 2 Introduction to Thesis	2
2 Software Prototyping	
2 1 Introduction	3
2 2 The Waterfall Life Cycle Model	3
2 3 The Prototyping Paradigm	4
2 3 1 Prototyping Framework	5
2 3 2 Classifying Prototyping	5
2 3 3 Prototype Construction using Executable Specification Languages	7
2 3 4 Prototyping Real-Time Systems	8
2 3 5 Advantages and Disadvantages of Prototyping	9
2 4 The STATEMATE Prototyping Tool	10
2 5 Summary	11
3. Real-Time Structured Analysis and Design	
3 1 Introduction	12
3 2 SA/SD and RTSA/SD	12
3 3 The Extended Systems Modelling Language (ESML)	13
3 3 1 The ESML Transformation Schema	13
3 3 1 1 Flows	13
3 3 1 2 Flow Transformations	15
3 3 1 3 Control Transformations	16
3 3 1 4 Stores	19
3 3 1 5 Terminators	20
3 3 1 6 Formation Rules	20
3 4 The Ward and Mellor RTSA/SD Method	21
3 4 1 The Essential Model	22
3 4 2 The Implementation Model	22

3 4 3	RTSA/SD and Object-Oriented Development	23
3 5	The ESML Execution Rules	23
3 5 1	Flows	24
3 5 2	Flow Transformations	24
3 5 3	Control Transformations	25
3 5 4	Stores	26
3 5 5	Multiple Token Placement	26
3 6	Summary	26
4.	Petri Nets	
4 1	Introduction	28
4 2	Classical Petri Nets	28
4 2 1	Classical Petri Net Structure	29
4 2 2	Transition Enabling and Firing	29
4 2 3	Classical Petri Net Analysis	31
4 2 4	Advantages and Disadvantages of Classical Petri Nets	31
4 3	High-Level Petri Nets	31
4 3 1	LOOPN Nets	32
4 3 1 1	Type Declarations	33
4 3 1 2	Places	34
4 3 1 3	Transitions	34
4 3 1 4	Modules and Module Instances	35
4 3 2	Advantages and Disadvantages of High-Level Petri Nets	36
4 4	Petri Net Simulation and Implementation	37
4 4 1	Centralised Implementation (C)	38
4 4 2	Distributed Implementation	38
4 4 2 1	Distribution of Control by Places	38
4 4 2 2	Distribution of Control by Edges	39
4 4 3	Petri Net Code Generators	39
4 4 3 1	The LOOPN Code Generator	40
4 4 3 1 1	Centralised Implementation in LOOPN	40
4 4 3 1 2	Distributed Implementation in LOOPN	41
4 4 3 2	Specification of Concurrent Systems (SPECS)	41
4 4 3 3	Concurrent Pascal with Petri Net (CPN)	42
4 4 3 4	AMI	43

4 4 3 4 1	TAPIOCA	43
4 4 3 4 2	PN_TAGADA	45
4 4 3 5	PROTOB	45
4 4 3 6	PROMPT	46
4 5	Summary	46
5	The ESML/LOOPN Prototyping System	
5 1	Introduction	47
5 2	Related Work	47
5 3	The ESML/LOOPN Prototyping System	49
5 3 1	Overview	49
5 3 2	The Prototyping Process	50
5 4	The Translation Process	51
5 4 1	Flows	53
5 4 2	Stores	57
5 4 2 1	Non-Depletable Stores	57
5 4 2 2	Depletable Stores	57
5 4 3	Flow Transformations	58
5 4 3 1	Flow Transformation with no Input Prompt	59
5 4 3 2	Flow Transformation with Trigger Input Prompt	61
5 4 3 3	Flow Transformation with an Activate (E/D) Input Prompt	62
5 4 4	Control Transformations	63
5 4 4 1	Control Transformation with no Input Prompt	64
5 4 4 2	Control Transformation with Activate (E/D) Input Prompt	65
5 4 4 3	Control Transformation with Activate (E/D) and Pause (S/R)	
Input Prompts		66
5 5	Summary	68
6	The APU Fuel Subsystem Case Study	
6 1	Introduction	69
6 2	The Auxiliary Power Unit (APU)	69
6 2 1	The APU Fuel Subsystem	70
6 3	Prototype Specification in ESML	72
6 4	Prototype Specification in STATEMATE	77
6 5	Comparing ESML with STATEMATE	84

6 6	Translating the ESML specification into a LOOPN net specification	85
6 6 1	Special Modules	87
6 6 1 1	Types Module	87
6 6 1 2	APU Module	88
6 6 2	Flow Transformations	88
6 6 2 1	Pump	88
6 6 2 2	Pump Bypass	89
6 6 2 3	Filter	90
6 6 2 4	Heat	90
6 6 2 5	Heat Bypass	92
6 6 2 6	Fuel Control Unit	92
6 6 3	Control Transformations	93
6 6 3 1	Fuel Control Valve	94
6 6 3 2	Boost Pump Control	95
6 6 3 3	Heat Control	96
6 6 3 4	Fuel Solenoid Valve	97
6 7	C Language Prototype Generation	98
6 8	Summary	98
7. Conclusions and Further Work		
7 1	Introduction	99
7 2	Research Summary	99
7 3	Analysis	100
7 4	Further Work	101
7 5	Conclusion	104
References		105
Appendix A		113
Appendix B		137
Appendix C		139

List of Figures

- Figure 2 1 The Waterfall Life Cycle Model
- Figure 2 2 The Prototyping Paradigm and its relationship to the Waterfall Life Cycle Model
- Figure 3 1 Graphical Components of the ESML Transformation Schema
- Figure 3 2 Flow Convergence and Divergence
- Figure 3 3 STD Behaviour of Control Transformation with Activate input prompt
- Figure 3 4 STD Behaviour of Control Transformation with Activate and Pause input prompts
- Figure 3 5 ESML Prompt Scenario
- Figure 3 6 ESML Formation Rules for Continuous Flows
- Figure 3 7 ESML Formation Rules for Discrete Flows
- Figure 3 8 ESML Formation Rules for Signal Flows
- Figure 3 9 Ward/Mellor Abstraction Levels
- Figure 4 1 Classical Petri net modelling of the chemical reaction $2\text{H}_2 + \text{O}_2 \rightarrow 2\text{H}_2\text{O}$
- Figure 4 2 LOOPN net for the Dining Philosophers Problem
- Figure 4 3 Token Filtering in LOOPN
- Figure 5 1 The ESML/LOOPN Prototyping System
- Figure 5 2 Templates for Flows
- Figure 5 3 Templates for Flow Convergence
- Figure 5 4 Templates for Flow Divergence
- Figure 5 5 Template for Non-Depletable Store
- Figure 5 6 Template for Depletable Store
- Figure 5 7 Template for Flow Transformation with no Input Prompt, Mixed Input and Output

- Figure 5.8 Template for Flow Transformation with no Input Prompt, Continuous Input, Mixed Output
- Figure 5.9 Template for Flow Transformation with no Input Prompt, Mixed Input, Continuous Output, Delayed Discrete Output
- Figure 5.10 Template for Flow Transformation with Trigger Input prompt
- Figure 5.11 Template for Flow Transformation with Activate Input Prompt
- Figure 5.12 Template for Flow Transformation with Activate Input Prompt, Discrete Input, Delayed Discrete Output
- Figure 5.13 Template for Control Transformation with no Input Prompt
- Figure 5.14 Template for Control Transformation with Activate Input Prompt
- Figure 5.15 Template for Control Transformation with matching Activate and Pause Input Prompts
- Figure 6.1 APU Fuel Subsystem Block Diagram
- Figure 6.2 APU Fuel Subsystem Behavioural Model
- Figure 6.3 STD for Fuel Control Valve
- Figure 6.4 STD for Boost Pump Control
- Figure 6.5 STD for Heat Control
- Figure 6.6 STD for Fuel Solenoid Valve
- Figure 6.7 Activity-chart for the APU Fuel Subsystem
- Figure 6.8 Statechart for FUEL_CONTROL
- Figure 6.9 Activity-chart for BP
- Figure 6.10 Statechart for BPC
- Figure 6.11 Activity-chart for FH
- Figure 6.12 Statechart for FHC
- Figure 6.13 Top-level LOOPN net equivalent of the APU Fuel Subsystem
- Figure 6.14 LOOPN net module for Pump
- Figure 6.15 LOOPN net module for Pump Bypass

- Figure 6 16 LOOPN net module for Filter
- Figure 6 17 LOOPN net module for Heat
- Figure 6 18 LOOPN net module for Heat Bypass
- Figure 6 19 LOOPN net module for FCU
- Figure 6 20 LOOPN net module for Fuel Control Valve
- Figure 6 21 LOOPN net module for Boost Pump Control
- Figure 6 22 LOOPN net module for Heat Control
- Figure 6 23 LOOPN net module for Fuel Solenoid Valve
- Figure 7 1 Future Version of the ESML/LOOPN Prototyping System

Abstract

The traditional software development paradigm, the waterfall life cycle model, is defective when used for developing real-time systems. This thesis puts forward an *executable prototyping approach* for the development of real-time systems.

A prototyping system is proposed which uses ESML (Extended Systems Modelling Language) as a prototype specification language. The prototyping system advocates the translation of *non-executable ESML* specifications into *executable LOOPN* (Language of Object Oriented Petri Net) specifications so that ESML can be used as a *graphical executable specification language* for the prototyping of real-time systems. If the translation process is automatic then the user need not be aware of LOOPN.

The ESML/LOOPN prototyping system defines an *execution semantics* for the ESML language in terms of LOOPN nets, a set of *translation templates* are supplied for the translation of ESML language specifications into LOOPN language specifications. The execution semantics are based on a set of *execution rules* (guidelines) which have been defined for ESML to allow prediction of the behaviour of ESML specifications over time. A C language program which can be run by the user as a prototype of the modelled system is generated automatically from the LOOPN specification.

The ESML/LOOPN prototyping system has been applied to build an exploratory prototype of a typical real-time system, i.e. the *Fuel Subsystem of the Auxiliary Power Unit* (APU), an avionic system used on the Boeing-737 airplane series.

Chapter 1

Introduction

1.1 Objective of Research

The objective of the research which has resulted in this thesis has been to develop a system which allows the prototyping of real-time systems. This has been achieved by the definition of the ESML/LOOPN prototyping system which facilitates the use of ESML as a graphical executable specification language.

The ESML/LOOPN prototyping system uses the graphical ESML language to specify prototypes at a level which is "natural" or "abstract" to the problem at hand. ESML enjoys the well known benefits of graphics-based languages, intuitiveness and ease of use as a vehicle for communication with users. Unfortunately, the intuitiveness of ESML has a price, i.e. the language is not executable and therefore cannot be used in isolation to prototype real-time systems. To overcome this problem the thesis first defines a set of execution rules for ESML. These rules are a set of guidelines which describe how an ESML specification will behave over time. The execution rules have been specified in terms of Petri net tokens.

The ESML/LOOPN prototyping system provides a set of translation templates which allow the translation of non-executable ESML specifications into executable LOOPN specifications. LOOPN is a high-level object-oriented Petri net language which can be executed and formally analysed. The translation templates are based on the guidelines set down by the ESML execution rules and so can be seen to define an execution semantics for ESML. Once the translation process is complete a C program is generated from the LOOPN specification which runs as the prototype. In this way ESML can be used as a graphical executable specification language to construct prototypes for use within a keep-it prototyping approach.

The ESML/LOOPN prototyping system ideally does not require the user to have any knowledge of LOOPN, but since the translation process has not been fully automated the user needs to understand LOOPN for the present time.

To sum up, the objective of the research has been to construct a prototyping system which allows the use of ESML as a graphical executable specification language, ESML therefore enjoys the twin benefits of intuitiveness and executability.

1 2 Introduction to Thesis

Chapter 2 investigates the prototyping of real-time systems. It outlines the deficiencies of the traditional software development paradigm, i.e. the waterfall life cycle model, before describing prototyping as an alternative. It outlines various categories of prototyping, describes a framework for constructing prototypes, focusing on the use of executable specification languages for constructing prototypes. Chapter 2 closes with a description of STATEMATE, a popular prototyping tool for real-time systems.

Chapter 3 introduces ESML and the Ward/Mellor Real-Time Structured Analysis and Design (RTSA/SD) method and defines a set of execution rules for ESML which allow the quantitative prediction of the behaviour of an ESML specification over time.

Chapter 4 introduces classical and high-level Petri nets, focusing on LOOPN nets and the LOOPN code generator. Various net implementation strategies are discussed.

Chapter 5 introduces the ESML/LOOPN prototyping system and presents the translation templates which are used to translate non-executable ESML specifications into executable LOOPN specifications. The translation templates are based on the ESML execution rules.

Chapter 6 introduces the case study, the APU Fuel Subsystem, and applies the ESML/LOOPN prototyping system to produce an executable prototype of it. STATEMATE has been used to build a prototype specification for comparison with the ESML specification.

Chapter 7 offers conclusions and comments about further work.

Chapter 2

Software Prototyping

2.1 Introduction

The objective of this chapter is to discuss the use of prototyping as a software development paradigm for real-time systems. The chapter does this by first discussing the waterfall life cycle model, the paradigm used since the early 1970s, and outlining its deficiencies. The chapter then describes prototyping, a simple framework which can be followed in the application of prototyping, and various classifications of prototyping. The chapter then describes executable specification languages, the technique for constructing prototypes that has been focused on in this thesis. The application of prototyping to real-time systems, a relatively new departure, is then discussed. The chapter then presents the advantages and disadvantages of prototyping. The chapter concludes with an introduction to the STATEMATE prototyping tool; it is used as a benchmark for evaluating the ESML/LOOPN prototyping system defined in Chapter 5.

2.2 The Waterfall Life Cycle Model

Since the early 1970s, software development has been based on the *waterfall life cycle model*, also known as the *traditional* or *conventional* development life cycle. The waterfall life cycle model was defined by Royce in 1970 [Royce70], and refined by Boehm in 1976 [Boehm76] to cope with the growing complexity of software projects being tackled at the time.

The waterfall life cycle model depicted in Figure 2.1 is a phase-oriented or linear approach which advocates that software development proceed through a number of distinct phases: requirements analysis; requirements specification; design; implementation; validation and verification; installation and maintenance. Its aim is to provide a basis for estimating the correct distribution of labour and capital over a well-planned period of time by dividing the development process into a number of phases, each with its own milestones and deliverables.

Unfortunately, the waterfall life cycle model makes some assumptions which can no longer be substantiated [Agesti86]. It provides very little user participation after requirement analysis; the first version of the system that can be executed is the implemented system. It assumes that a complete, concise and consistent specification of requirements can be completed before the design phase and that requirements will not change once the design effort commences. It does not facilitate experimentation with

different design decisions The waterfall life cycle model dictates that specification ("what") and design ("how") be kept separate, however it is very difficult to specify a problem without some notion of what the solution should be

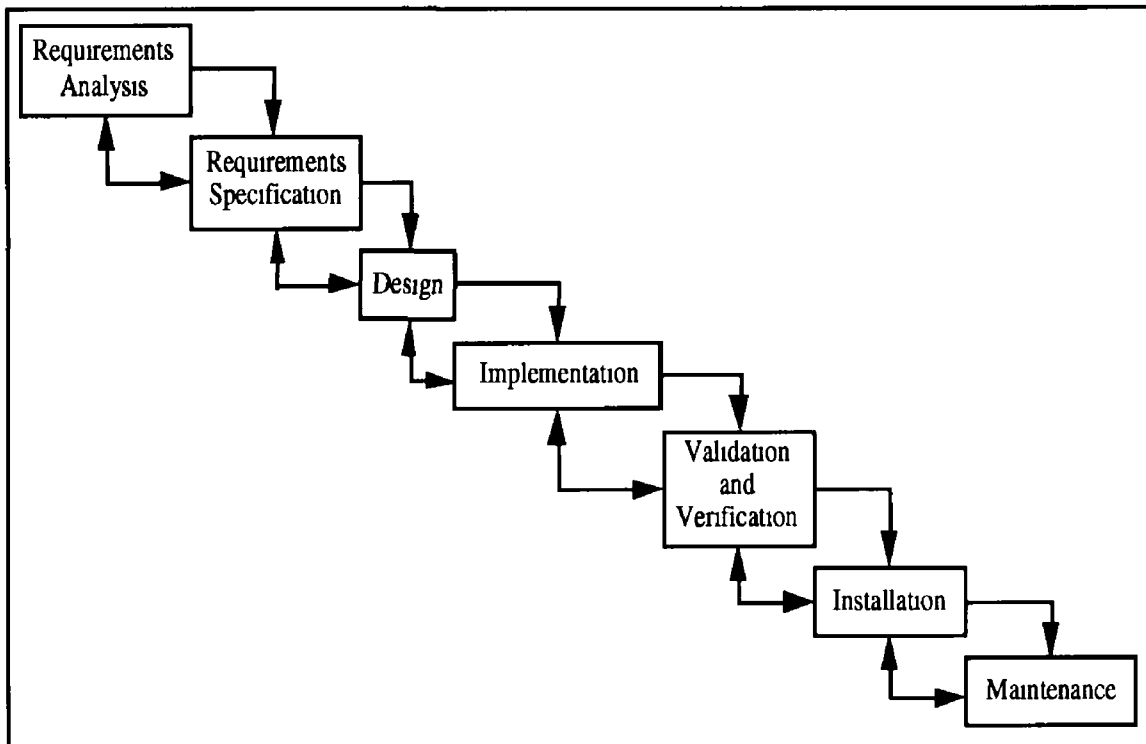


Figure 2 1 The Waterfall Life Cycle Model

2 3 The Prototyping Paradigm

Prototyping [Vonk90] is a new paradigm for software development which has been proposed to overcome the deficiencies of the waterfall life cycle model Some forms of prototyping can be used in conjunction with the waterfall life cycle model, some are new development approaches in their own right

Prototyping originated from those engineering disciplines which were involved in mass production [Hekmatpour88] There, it refers to a well established phase in the production process whereby a model is built which exhibits all the intended properties of the final product Such a model serves the purposes of experimentation and evaluation to guide further development and production In software engineering the notion of mass production is absent, instead production refers to the entire process of building one product For this reason, the concept of prototyping takes a rather different meaning Here, most commonly, it refers to the practice of building an early version of the system which does not necessarily reflect all the features of the final system, but rather those

which are currently of interest. In particular, the prototype must be able to be developed quickly at low cost, hence the term *rapid prototyping*.

The main points about this alternative development paradigm are that

- Prototyping is an approach, or strategy, that defines the outline of a software development process, while leaving the details unspecified. Prototyping has been proposed on the premise that software development, especially in the early stages, should be a learning process and should actively involve both the developer and the user.
- A *prototype* is a working model of part of a system, which emphasises specific aspects of that system. A prototype is incomplete, and can act as a learning device, so helping to reduce the risks of the development effort.
- Prototyping is an approach for system development that is characterised by a high degree of iteration, and by a very high degree of user participation in the development process.
- The chief premise of prototyping is that a prototype constitutes a better means of communication than a paper-based model, and that iteration is necessary to channel the inevitable learning process in the right direction.

2.3.1 Prototyping Framework

To be effective, prototyping needs to be carried out within a *systematic framework*. Such a framework provides a step by step guide on how the designer should proceed. Hekmatpour has outlined a general framework for applying prototyping. The framework consists of firstly establishing the prototype *objectives* and *scope*, i.e. what a prototype is supposed to be used for and what aspects of the proposed system it should include. Defining the scope consists of selecting the functions to be included in the prototype. Depending on the prototype objectives, prototyping may be carried out *horizontally*, *vertically* or *diagonally*. Horizontal prototyping involves including all the system functions in the prototype, where each function is considerably simplified and reduced. Vertical prototyping involves only including some of the functions, where each of these is fully realised. Diagonal prototyping is a hybrid of these two. Once the scope is defined, the prototype development plan is established. This entails deciding how the prototype is to be constructed, and what tools and techniques are to be used.

2.3.2 Classifying Prototyping

There are many classifications of prototyping, the most common are explained in the following paragraphs.

One classification of prototyping is based on the timing of the development process. *Exploratory prototyping* [Floyd84], also known as *behavioural prototyping*, is used to aid the task of analysing and specifying user requirements. Systems developed using the waterfall life cycle model approach usually satisfy outdated requirements, rarely current requirements. Accurate requirements specification is necessary to avoid incurring large costs in fixing software due to requirement errors later in the development process. Exploratory prototyping involves the iterative refinement of the prototype until it reflects the "real" requirements. This allows the developer to home in on the "moving target" of user requirements, the success of a system depending on the quality of the requirements specification. The user is an active participant, evaluating prototypes, proposing improvements and at the same time continuing to obtain a deeper insight into intended system behaviour. The ESML/LOOPN prototyping system currently addresses the construction of exploratory prototypes.

Experimental prototyping, also known as *structural prototyping*, has been described by Floyd. It is used as a complementary tool in the software design phase for studying the feasibility and appropriateness of various system designs. This was not possible in the waterfall life cycle model since the implemented system is the first version of the system which could be executed. A special type of experimental prototyping is *performance prototyping*, it involves the use of prototypes to evaluate the effect of design decisions on the ability of the system to handle the anticipated workload.

Another class of prototyping, which has been described by Hekmatpour, is *evolutionary prototyping*. This is a development approach in which the prototype evolves into the production system. Each version of the production system is then used as a prototype for its successor. This approach requires the system to be designed in such a way that it can cope with change during and after development. The success of the evolutionary approach is very much dependent on the ability of the designer to build flexibility and modifiability into the prototype from the offset, and for this reason prototypes are often constructed in an *object-oriented* manner [Booch91]. Evolutionary prototyping includes both exploratory and experimental prototyping within itself.

Another classification of prototyping, described by Hekmatpour, is based on what happens to a prototype after the objective for its construction has been achieved. If a prototype is used as the basis for the production system it is referred to as *keep-it prototyping*. If the prototype is discarded, the term *throw-away prototyping* is used. The need for rapid prototyping is greatest for throw-away prototyping. Since the prototype is to be used for a limited period, quality factors such as efficiency, structure, and maintainability are of little relevance.

Figure 2 2 shows the prototyping paradigm and its relationship to the waterfall life cycle model. It shows a framework for applying prototyping and illustrates that exploratory and experimental prototyping are compatible with the waterfall life cycle model.

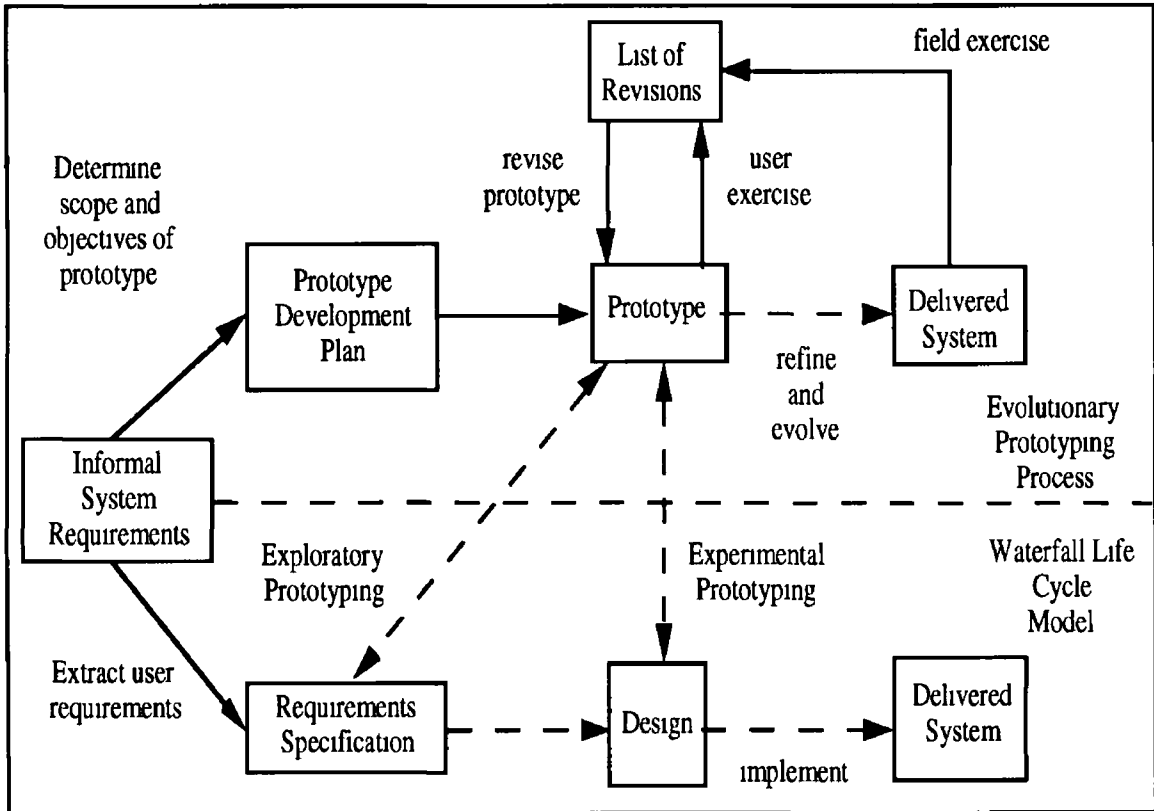


Figure 2 2 The Prototyping Paradigm and its relationship to the Waterfall Life Cycle Model

2 3 3 Prototype Construction using Executable Specification Languages

This thesis focuses on the use of *executable specification languages* for constructing prototypes of real-time systems. However, executable specification languages are just one of numerous techniques for constructing prototypes (e.g. *functional languages* [Henderson86]).

An executable specification (*operational specification*) is described by Hekmatpour as a specification which has an underlying operational semantics facilitating interactive interpretation of the specification within a software environment, or automatic generation of a computer program which executes the specification. Executable specifications, by definition, can be used as prototypes. They facilitate problem statement with some consideration of the solution, and so provide a link between the "what" and the

"how" of software development Executable specifications can be produced at low cost, and ensure that a precise level of documentation is always available to the developer It has been argued that executable specifications restrict expressiveness and adversely effect implementations [Hayes89], though this has later been refuted [Fuchs92]

Graphical executable specification languages are executable specification languages which use some form of graphical notation for model specification Examples of graphical executable specification languages which are suited to the prototyping of real-time systems, are Statecharts [Harel87] and Petri nets [Petri66] These languages have special constructs for specifying real-time behaviour

2.3.4 Prototyping Real-Time Systems

The above discussion mainly concerns the use of prototyping as an alternative development paradigm for transformational systems, such as information and data-processing systems However, this thesis concerns the prototyping of *real-time systems* The following paragraphs describe reactive and real-time systems and how prototyping applies to them

Reactive systems are systems whose primary role is to maintain some interaction with the environment [Ledru90] A reactive system, unlike a transformational system, is not adequately described by a simple relationship that specifies outputs as a function of inputs, but rather requires relating outputs to inputs though their allowed combination in time [Harel92] Typically, such descriptions involve complex sequences of events, actions, conditions and information flow, often with explicit timing constraints, that combine to form the overall system behaviour

Real-time systems [Agrawala92] [Laplante92] belong to the class of reactive systems A real-time system is a software system that has been developed to support the execution of a real-time application ensuring that the real-time requirements of the application are met Real-time requirements impose timing restrictions on the application, the correctness of a real-time system depends on the timing requirements being met Real-time systems are different from other types of software systems in that real-time systems normally interact with physical devices that have to be monitored and controlled In real-time systems, concurrent behaviour of the system components is the norm, a system consists of parts which co-exist, each conducting its affairs concurrently with the others, although influenced in doing so by the others' behaviour Real-time systems continue to operate in the absence of inputs, contrasting with transformational systems which only produce output when supplied with input

In recent years there has been growing interest in prototyping real-time systems [Hughes89] [Tsaı89], though, in the past, the systematic use of prototypes for developing real-time systems has not gained much attention in the literature, indeed real-time systems were considered by Hekmatpour to be not particularly fit for prototyping due to their inherent concurrency and time-dependent behaviour

Prototypes of real-time systems need the same quality of specification and design as in the production system. Attempts to shortcut careful specification and design, by quick and dirty methods, are sure to fail. For this reason, real-time systems require a keep-it prototyping approach, it would be wasteful to throw away the prototype after completion of the prototyping effort. The ESML/LOOPN prototyping system allows the construction of exploratory prototypes, they are intended for use within a keep-it prototyping approach.

For real-time systems, the use of *heterogeneous prototypes* [Mortensen90] is an emerging alternative. A heterogeneous prototype is an executable system model whose different parts reflect different system abstraction levels, and yet can be executed together as a unit. This type of prototype has been motivated by Boehm's spiral model of software development. Boehm's approach is an iterative risk-driven approach, it advocates that the high risk elements of a system should be developed before the low risk elements. Using a heterogeneous prototype, the high risk elements will be explored to a more detailed abstraction level before the low risk elements. The abstraction mix changes over time. A heterogeneous prototype can therefore be seen as an extension to horizontal prototyping allowing different parts of the system to develop at different speeds resulting in unequal abstraction levels.

2.3.5 Advantages and Disadvantages of Prototyping

The advantages of prototyping have been discussed at length by Vonk and Hekmatpour and also by Alavi [Alavi84]. They stem mostly from the benefits accrued from overcoming the deficiencies of the waterfall life cycle model.

- Prototyping facilitates effective communication between developers and users, a prototype is real and tangible, not abstract.
- Prototyping allows developers to cope with fuzzy changeable requirements, and facilitates experimenting with alternative design decisions.
- Prototyping encourages user participation in the development process, and creates a positive spirit between developers and users.
- Prototyping causes users to become more enthusiastic about system development, it provides early product visibility.

- Prototyping enables low-risk development, it can reduce development time and result in better quality software
- Prototyping ensures that the nucleus of the system is correct. Systems developed using prototyping approaches tend to be more user-friendly, more efficient, and more maintainable
- Prototyping allows a system to be gradually introduced into an organisation, it facilitates user training in parallel to system development

The disadvantages of prototyping, outlined in the following paragraphs, are far outweighed by its advantages

- Prototyping can result in users developing unrealistic expectations about the system to be delivered, indeed sometimes user interest wanes after a few initial iterations
- Prototyping is hard to control, resource planning and management can be difficult, since the number of iterations is unknown beforehand
- Prototyping requires automated tool and method support to ensure cost effectiveness
- Prototyping can cause misunderstandings, managers may overestimate the maturity of a prototype and may withdraw resources prematurely

2.4 The STATEMATE Prototyping Tool

STATEMATE [Harel88] is popular *CASE tool* which facilitates the prototyping of real-time systems by using graphical executable specification languages for prototype construction. *STATEMATE* has been used to build a prototype of the APU Fuel Subsystem case study (Chapter 6), it therefore serves as a benchmark for comparison with the prototyping system defined in this thesis. The ESML/LOOPN prototyping system enables the use of ESML as a graphical executable specification language. ESML is discussed in the next chapter.

The underlying premise of *STATEMATE* is the need to consider a system from three closely related viewpoints: *structural*, *functional*, and *behavioural* [Harel92]. *STATEMATE* provides three languages [1-Logix91] for the description of these viewpoints, *Module-charts*, *Activity-charts*, and *Statecharts*.

Module-charts are used to describe the system from a structural, or physical viewpoint. *Activity-charts* provide the dominant system decomposition, i.e. the system is described as a hierarchy of *activities* (functions), complete with details of the data items and control signals that flow between them. *Activity-charts*, like *Module-charts*, do not specify system dynamics. They do not state when activities are activated, whether or not

they terminate on their own, or whether they can be executed concurrently. Statecharts [Harel87], which are formally based on the *Finite State Machine* (FSM) [Ferrentino77], describe the system behaviour over time. The Statechart is an extension of the FSM to include *hierarchy, concurrency, and broadcast communication*.

2.5 Summary

This chapter has described the use of prototyping as an alternative paradigm for developing real-time systems. The deficiencies of the waterfall life cycle model are due mainly to its lack of executability at each stage in the development cycle. The prototyping paradigm overcomes these by providing model executability at each stage in the development effort and a flexible framework for prototype revision. The chapter has described a framework in which to apply prototyping, various prototyping classifications, and executable specification languages. It has noted that a keep-it prototyping approach is necessary for real-time systems. The emerging idea of heterogeneous prototyping has also been discussed. The following chapter discusses ESML, it is used to specify prototypes in the ESML/LOOPN prototyping system.

Chapter 3

Real-Time Structured Analysis and Design

3.1 Introduction

The previous chapter has discussed the use of prototyping as an alternative software development paradigm for real-time systems. The objective of this chapter is to describe the Extended Systems Modelling Language (ESML), and define a set of execution rules (guidelines) for ESML to allow the behaviour of ESML specifications to be predicted over time. The chapter starts with a discussion of Real-Time Structured Analysis and Design (RTSA/SD) methods, before outlining the ESML components i.e. flows, transformations, stores and terminators. ESML is used as the prototype specification language in the ESML/LOOPN prototyping system. The Ward/Mellor RTSA/SD development method is introduced, it can be used to guide the ESML specification effort. The chapter then defines a set of execution rules for ESML. These rules are a set of narrative guidelines on how an ESML specification will behave over time. The rules are specified in terms of Petri net tokens, a token being used to indicate actual or potential activity. The ESML/LOOPN prototyping system uses LOOPN nets to define an execution semantics for ESML according to the execution rules which are defined in this chapter.

3.2 SA/SD and RTSA/SD

The original *Structured Analysis/Structured Design* (SA/SD) method [DeMarco78] dates from the late 1970s and is intended mainly for non-reactive systems development. SA/SD uses intuitive graphical and textual notations to create specification and design models. The main graphical notation used in SA/SD is the *Data Flow Diagram* (DFD), it describes the flow of data and its processing within a system. Many variants of SA/SD exist, including those proposed by Gane and Sarson [Gane79], Ross [Ross77], and Yourdon [Yourdon89].

SA/SD has been extended to allow the modelling of real-time systems, the extensions being referred to as the *Real-Time Structured Analysis and Design* (RTSA/SD) methods. The two most widely used RTSA/SD methods are those proposed by Hatley and Pirbhai [Hatley87], and by Ward and Mellor [Ward85a] [Ward85b] [Ward86a]. The Ward/Mellor RTSA/SD method is a development method for constructing specification and design models using an extension of the DFD referred to as

the *Ward/Mellor transformation schema* The transformation schema extends the DFD with notations for describing control and timing

3.3 The Extended Systems Modelling Language (ESML)

The best characteristics of the graphical notations defined by the Ward and Mellor and Hatley and Pirbhai RTSA/SD methods have been combined to form the *Extended Systems Modelling Language (ESML)* [Bruyn88] ESML does not define a new development method but defines a new extension to the data-flow diagram (DFD), the *ESML transformation schema* ESML provides a more comprehensive and flexible set of constructs for modelling control logic than its predecessors The Ward/Mellor RTSA/SD method can be used to guide the ESML specification effort

ESML is a graphical language especially suited to the early stages of system development ESML enjoys the benefits of graphics-based languages, it facilitates comprehension of the problem to be solved while serving as a vehicle for communication between developers and users ESML addresses system modelling at a "natural" or "abstract" level to the problem at hand The intuitiveness of ESML has a price, the language lacks a set of rigorous definitions of its constructs and their possible combination, i.e. ESML is not executable The execution rules defined in this chapter are the basis for the definition of an execution semantics for ESML in terms of LOOPN nets (chapter 5)

3.3.1 The ESML Transformation Schema

The graphical components which make up the ESML transformation schema are depicted in Figure 3.1 They are based on the components used in the Ward/Mellor and Hatley/Pirbhai structured languages The following paragraphs describe each component in turn

3.3.1.1 Flows

Flows are directed arcs that carry some form of data from one component to another A flow is *discrete* if its data is defined at discrete points in time A flow is *continuous* if its data is defined continuously over a time interval *Data flows* are flows that carry variable data *Signal flows* do not carry variable data, they just report an event occurrence

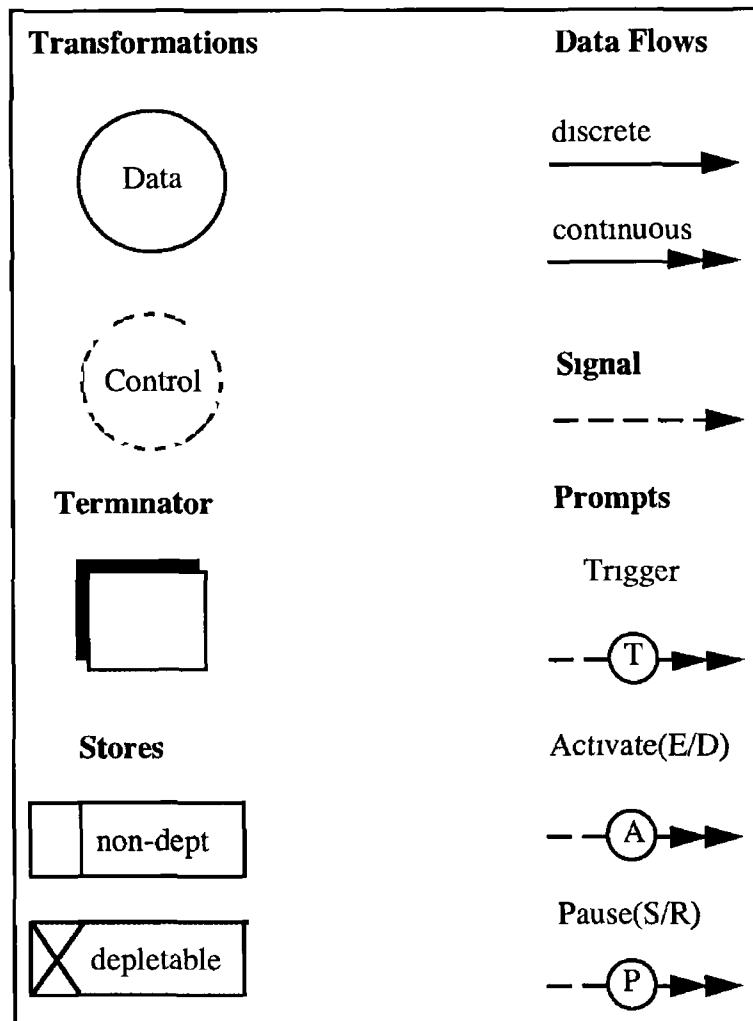


Figure 3 1 Graphical Components of the ESML Transformation Schema

ESML provides special flows, *prompts*, to facilitate the more accurate modelling of control logic. Prompts are a special kind of flow representing control imposed by one transformation on another. They are distinguished by a letter placed in a small circle at the head of the flow. ESML provides three prompts for use in its transformation schema: *Activate (E/D)*, *Pause (S/R)* and *Trigger (T)*. The Activate prompt is a composite of *Enable (E)* and *Disable (D)* prompts. The Pause prompt is a composite of *Suspend (S)* and *Resume (R)* prompts. The interpretation of prompts is different here from that described by Bruyn, the original paper being slightly ambiguous on the subject. Prompts are explained in the sections which describe flow and control transformations.

Flows of a particular type can converge or diverge to represent multiple sources, multiple destinations, or in the case of data flows, combination and separation of content. The various alternatives are shown in Figure 3.2.

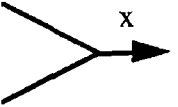
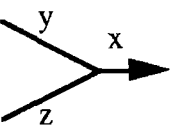
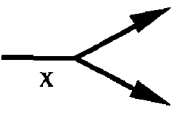
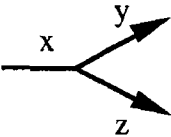
Notation	Interpretation
	All of x can be supplied from either of the two senders
	Two subsets of x are supplied by two senders
	All of x is sent to both of two receivers
	Two subsets of x are sent to two receivers

Figure 3 2 Flow Convergence and Divergence

3.3.1.2 Flow Transformations

Flow transformations, also known as *data transformations*, are depicted as circles and represent the basic functional entities of the system. Flow transformations are structured hierarchically into a levelled set of schema. At the lowest level, *primitive flow transformations* are specified using *mini-specifications*. A mini-specification is a definition of the algorithm used by a primitive flow transformation to transform its inputs into outputs. The balance on inputs and outputs between levels of the hierarchy must be maintained, hierarchy allows the splitting of large models into more manageable units.

Flow transformations can have discrete inputs (data and signals), continuous data inputs, and may have discrete and continuous outputs. A flow transformation can have no input prompt, an Activate input prompt, or a Trigger input prompt. A primitive flow transformation can execute if it is *enabled* and all its inputs are available. Its discrete inputs may arrive in arbitrary order. A flow transformation with no input prompt is always enabled. A flow transformation with an Activate input prompt is deemed to be enabled if the last prompt it received was an Enable prompt. Trigger prompts are reserved for flow transformations which have continuous inputs only. A flow transformation with a Trigger input prompt is enabled when it receives the Trigger prompt. A Trigger prompt causes a flow transformation to perform a time-discrete action, such as producing an discrete flow, or storing the instantaneous value of a continuously available flow.

The production of discrete data outputs may be associated with a delay. This delay, known as the *output delay*, is the amount of time between the establishment of the conditions for the production of discrete output and the completion of the production of this output. The output delay is specified in the mini-specification as a number of clock ticks.

3.3.1.3 Control Transformations

Control transformations, depicted as dashed circles, represent units of control logic within the system that dictate when, and for how long, other transformations are active.

In the Ward/Mellor transformation schema, control transformations can only have signals as input. ESML allows continuous data flows to be inputs to control transformations, the values of continuous inputs now influence the behaviour of the system. In the Ward/Mellor transformation schema, only one control transformation can exist per schema, in which case the control transformation represents the centralised control centre of the system. ESML relaxes this constraint, several control transformations, representing the distributed control structure of the system, can exist in the same schema. The control transformations can control each other through prompt flows, control transformations which are enabled simultaneously are assumed to execute concurrently.

State Transition Diagrams (STDs) are used to define the control logic of an individual control transformation, i.e. how it reacts to various signals and continuous inputs. A STD is the graphical representation of a *Finite State Machine* [Ferrentino77]. In a STD, *states* (rectangles) are connected by *transitions* (arcs), one state being deemed the *initial state*. Transitions are labelled using *inputs* and *outputs*.

Transition inputs, prefixed by "I", may consist of an event and flow condition (both optional). The event corresponds to an input signal, the flow condition is a Boolean condition on any continuous input. A transition input consisting of an event will occur if the signal occurs while the system is in the origin state. A transition input consisting of a flow condition only will occur if the value of the continuous input was attained before, when, or after the origin state is reached. A transition condition consisting of an event and one or more flow conditions is the conjunction of the event and flow conditions.

Transition outputs, prefixed by "O", represent zero or more actions to be performed concurrently with the transition. Each action may be the name of a signal output, a T, E, D, S or R prompt enclosed in "◇", or the assignment of a value to an output continuous flow.

A control transformation in ESML can have no input prompt, an Activate input prompt, or matching Activate and Pause input prompts. A control transformation with no input prompt is always enabled. A control transformation, that is connected to another transformation by an Activate prompt, can initiate and terminate the activity of that transformation by sending Enable and Disable prompts along the composite prompt. When a transformation is disabled, it forgets any intermediate results. A control transformation, that is connected to another control transformation by a Pause prompt, can suspend and resume the activity of that control transformation by sending Suspend and Resume prompts along the composite prompt. A suspended control transformation, when resumed, continues execution from the exact state it was in when suspended, i.e. intermediate results are remembered. The following paragraphs use STDs to define the behaviour of control transformation with various input prompts more exactly.

Figure 3.3 defines the generic behaviour of a control transformation which has an Activate (E/D) input prompt. Initially the transformation is disabled. When it receives an Enable prompt it becomes enabled and operates according to the STD which defines its own individual behaviour. A transformation is deemed to be *activated* by a control transformation if an Activate prompt flows from the control transformation to that transformation. On receipt of a Disable prompt the control transformation becomes disabled and propagates the Disable prompt on to any other transformations that it activates. This ensures that transformations activated by a control transformation are disabled once the control transformation itself becomes disabled.

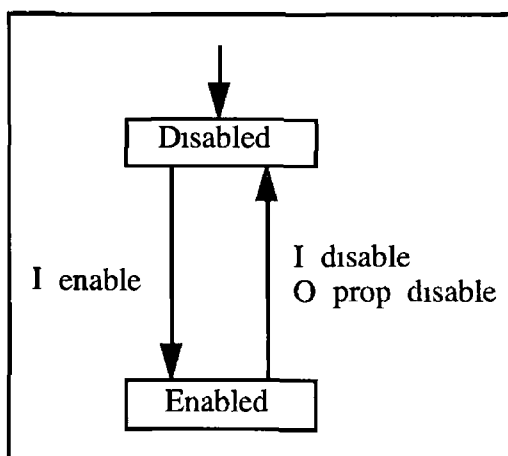


Figure 3.3 STD Behaviour of Control Transformation with Activate input prompt

Figure 3.4 defines the generic behaviour of a control transformation which has matching Activate (E/D) and Pause (S/R) input prompts. The behaviour is an extension of

that of Figure 3 3 to cope with Suspend and Resume prompts The control transformation will become suspended if it receives a Suspend prompt while enabled The Suspend prompt is also propagated on to any other control transformations that the control transformation activates and to which it is also connected by a Pause prompt This ensures that a control transformation that is activated and paused by another control transformation is suspended once the other control transformation is suspended The control transformation of Figure 3 4 will become enabled if it receives a Resume prompt while suspended The Resume prompt is propagated, as above, to any other control transformations that the control transformation activates and to which it is also connected by a Pause prompt It should be noted that in Figures 3 3 and 3 4, the unexpected arrival of a prompt (i e a prompt arriving at a state for which there is no outward transition) results in its consumption without effect

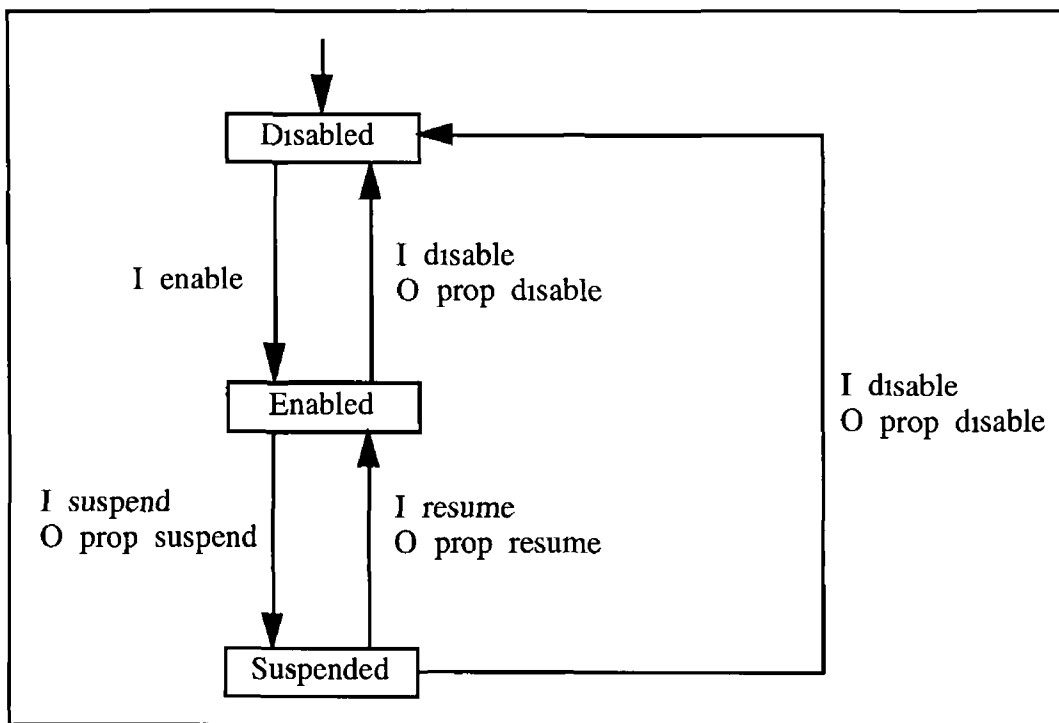


Figure 3 4 STD Behaviour of Control Transformation with Activate and Pause input prompts

To illustrate the above consider Figure 3 5 which consists of two flow transformations and four control transformations which are connected by a complex set of prompts Note that data and signal flows have been omitted for simplicity

Consider the behaviour of control transformation B If while enabled it receives a Disable prompt from A, B will forward the Disable prompt on to any transformations that it activates, in this case D, E, and F, and become disabled itself It does not forward the

Disable prompt on to G, since it is activated by H. If while enabled B receives a Suspend prompt from A, B will forward the Suspend prompt on to any transformations it activates and to which it is also linked by a Pause prompt. In this case it will send on the Suspend prompt to F only, and become suspended itself. If while enabled B receives a Resume prompt from A, then it resumes at the state where it was suspended, and forwards the Resume prompt on to F.

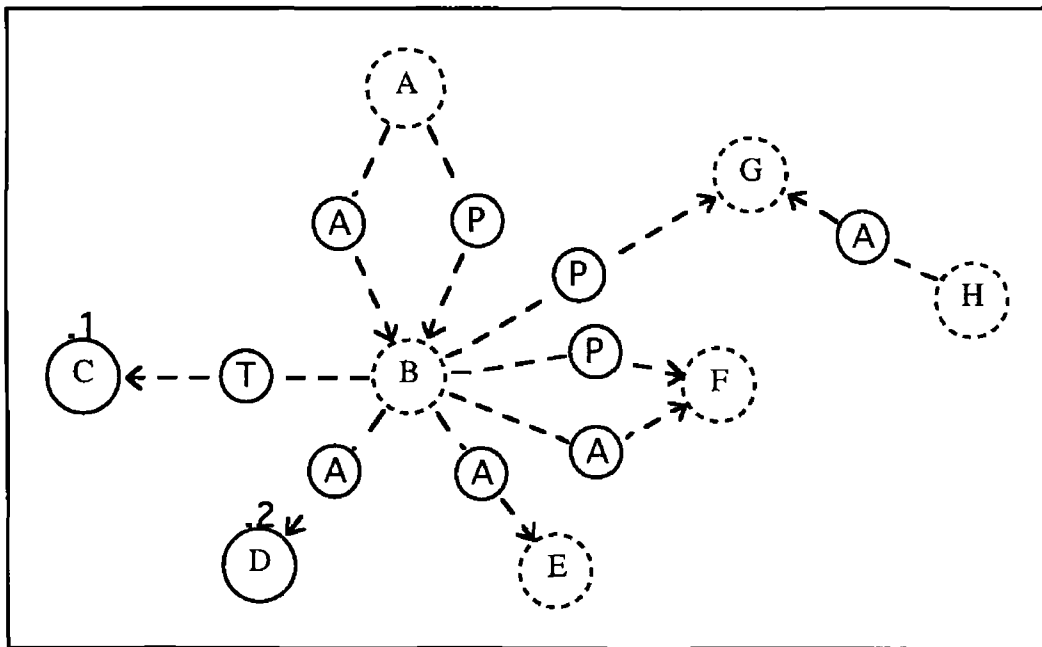


Figure 3.5 ESML Prompt Scenario

3.3.1.4 Stores

Stores are represented by pairs of horizontal lines. Stores hold information that persists within the schema which is accessible to transformations at discrete points in time.

When a *non-depletable store* is accessed, the contents of the store are merely copied and the information in the store continues to be available for use at a later time by other transformations. Writing to a non-depletable causes the store contents to be overwritten.

Depletable stores represent repositories for information that is consumed on being read. Depletable stores act as queues or stacks, a write adds an item to the store, a read removes an item. Depletable stores can have finite capacities.

3.3.1.5 Terminators

Terminators model the system environment, representing any real-world entities that are external to the modelled system, but with which the system must interact. Terminators can represent other systems, devices, or people, and are depicted as shaded boxes.

3.3.1.6 Formation Rules

Formation rules dictate which types of flow may be used to connect one graphical component to another. Unambiguous flow/component connections are prohibited. Figures 3.6, 3.7 and 3.8 show the formation rules for continuous, discrete and signal flows respectively.

from	to	Flow Trans	Control Trans	Non-dept Store	Dept Store	Term
Flow Trans		→→	→→			→→
Control Trans		→→	→→			→→
Non-dept Store						
Dept Store						
Term		→→	→→			

Figure 3.6 ESML Formation Rules for Continuous Flows

from	to	Flow Trans	Control Trans	Non-dept Store	Dept Store	Term
Flow Trans		→		→	→	→
Control Trans						
Non-dept Store		→				
Dept Store		→				
Term		→				

Figure 3.7 ESML Formation Rules for Discrete Flows

from	to	Flow Trans	Control Trans	Non-dept Store	Dept Store	Term
Flow Trans		--->	--->			--->
Control Trans		--->	--->			--->
Non-dept Store						
Dept Store						
Term		--->	--->			

Figure 3 8 ESML Formation Rules for Signal Flows

3.4 The Ward/Mellor RTSA/SD Method

The Ward/Mellor RTSA/SD method, which can be used to guide the ESML specification effort, is described in the following paragraphs

The Ward/Mellor RTSA/SD method advocates a three phase approach to the development of real-time systems which is characterised by progression from higher to lower levels of abstraction. Development should proceed by firstly building the *Essential Model*, the logical system model, and then building the *Implementation Model*, the physical system model. The final phase consists of building the system described by the implementation model. The Ward/Mellor abstraction levels are shown in Figure 3 9

Model	Sub-Model	Implementation Dependence	Notation	Development Phase
Logical	Environmental	Independent	Context diagram Event lists	Analysis Design
	Behavioural		DFD, STD, ERD, Textual mini-specs	
Physical	Processor Environment	Dependent	Structure chart	
	Software Environment			
	Code Organisation			
Implementation			Code	Implementation

Figure 3 9 Ward/Mellor Abstraction Levels

3.4.1 The Essential Model

The *Essential (requirements) Model* describes the logical essence of the system, while assuming perfect technology. Creating the essential model consists of building two sub-models: the *Environmental Model*, and the *Behavioural Model*.

The Environmental Model itself consists of two sub-models: the *Context Schema*, and the *Event List*. Creating the Context Schema consists of determining the boundary between the system and the environment, identifying the terminators with which the system must communicate, and defining the interfaces between the system and these terminators. The event list records the external events (stimuli) to which the system must respond.

The Environmental Model is used to build the Behavioural Model, which describes the system's externally observable behaviour using a levelled set of transformation schema, and uses STDs for defining the control logic of the system. The behavioural model is constructed using stimulus-response partitioning, the system response to each event contained on the event list is considered in turn.

The data components of the system are described using the *Data Schema* and the *Data Dictionary*. The Data Schema models the system as a network of data categories that are linked to one another by relationships. The basic notation used is that of the *Entity-Relationship (ER)* diagram [Chen76]. The Data Dictionary is an organised list of definitions of all the data elements that are pertinent to the system.

Since the Essential Model assumes perfect technology, it does not consider output delays for flow transformations, except for those reflecting externally imposed requirements, and assumes that depletable stores have infinite capacities.

3.4.2 The Implementation Model

The *Implementation Model* elaborates the essential model to reflect how the system is to be implemented. The Implementation Model views the system as a real machine with limited resources.

The Implementation Model consists of three sub-models: the *Processor Environment Model (PEM)*, the *Software Environment Model (SEM)*, and the *Code Organisation Model (COM)*.

The PEM details the allocation of the system activities and data of the essential model to processors that will be used to implement the system, a processor may be human or mechanical.

The SEM is a description of the software architecture inside one processor, building the SEM involves imposing sequences on potentially concurrent data.

transformations, and dropping the assumption that transformations operate instantaneously

The COM describes the modularisation scheme used to implement the software, and identifies a hierarchy of modules. *Structure Charts* are used to describe the internal structure of transformations which represent single tasks on single processors

3.4.3 RTSA/SD and Object-Oriented Development

The emergence of *object-oriented languages*, and *object-oriented analysis (OOA) and design methods (OOD)* has influenced RTSA/SD. Ward argues that OOA and OOD are compatible with RTSA/SD, indeed the Ward/Mellor RTSA/SD has imported some of its concepts [Ward89]. The influence has been seen at the analysis phase, where object-identification is used as the criterion for analysis model partitioning. The objects are identified by considering the ER diagram constructed at the analysis phase. Stimulus-response analysis is reinterpreted as identification of operations to which object classes must respond. The analysis objects migrate to the design phase. Object identification results in cleaner interfaces between system components, it has been followed in the construction of the ESML specification of the case study (section 6.3)

3.5 The ESML Execution Rules

This section defines a set of *execution rules* for ESML. These execution rules are a set of *guidelines* on how an ESML specification should behave over time. The ESML execution rules are defined in a similar manner to the execution rules which Ward has defined for his transformation schema [Ward86b]. There is, however, one important difference between the ESML execution rules and the Ward execution rules. The Ward execution rules facilitate *token-based* [Brackett87], or *qualitative*, model execution. This type of execution describes the acceptance of inputs and the production of outputs over time, but not input and output values. The ESML execution rules facilitate *prototype (functional)* model execution, this is a more extensive type of execution which involves modification of data values. The ESML execution rules are defined narratively in terms of Petri net tokens. A *token*, which indicates actual or potential activity, can be placed on a flow, transformation or store. It should be noted that the ESML execution rules are a set of guidelines on how an ESML is to be executed, they form a theory. The ESML/LOOPN prototyping system of Chapter 5 uses LOOPN nets to define an execution semantics for ESML based on the guidelines set down by the ESML execution rules. The following paragraphs present the execution rules for each of the ESML constructs in turn.

3.5.1 Flows

Prototype model execution requires that tokens associated with continuous and discrete data flows be assigned values to indicate the data content of the flows. The ESML execution rules are therefore specified in terms of *high-level Petri nets* (section 4.3), where tokens which can be composed of attributes of specific types, so allowing a token to hold data values.

Continuous data flows which are inputs from terminators always carry a token, the attributes contain the data values. Continuous data flows produced by a flow transformation carry tokens whenever the transformation carries a token and also during the output delay, the tokens indicate that the transformation is actively controlling the values of its continuous outputs.

The placement of a token on a discrete data flow at a point in time indicates that data is present on the flow. The placement of a token on a signal or prompt flow at a point in time indicates an instance of the signal or prompt has occurred. Tokens which represent signal flows are unstructured. Composite prompts, such as the Activate and Pause prompts, require that tokens placed on prompt flows be assigned values to indicate the actual prompt sent.

3.5.2 Flow Transformations

A token placed on a flow transformation indicates that it is capable of transforming its discrete and continuous inputs into discrete and continuous outputs. The placement of a token on the input prompt of a flow transformation causes an interaction between the prompt and the transformation which always results in the removal of the token placed on the prompt. The placement of a token on a composite Activate input prompt to indicate an Enable prompt, causes a token to be placed on the flow transformation if it has not already got one. The placement of a token on a composite Activate input to indicate a Disable prompt, causes the removal of any token that the flow transformation carries, and any tokens that continuous outputs carry. Flow transformations with an Activate input prompt therefore carry tokens while they are enabled, i.e. during the period following receipt of an Enable prompt and preceding receipt of a Disable prompt, except during their output delays.

A flow transformation with no input prompt, or a Trigger input prompt, always carries a token, except during the output delay. A token placed on a Trigger input prompt forces production of discrete output from available continuous input.

A primitive flow transformation which carries a token will transform its inputs into outputs according to its mini-specification if all its discrete and continuous inputs carry tokens. Tokens are placed on continuous outputs immediately, tokens are placed on

discrete outputs after the output delay has expired, during the output delay the transformation does not carry a token. Any discrete inputs that have tokens placed on them while the flow transformation does not carry a token (i.e. while it is disabled or during the output delay) will have the tokens removed without effect. A flow transformation with continuous inputs only, which carries a token, will compute its outputs only if its continuous inputs carry tokens, and its continuous inputs have changed since the transformation was last executed.

3.5.3 Control Transformations

Control transformations may carry a token, this is associated with one of the states of STD. It serves to define the state of the control transformation over the time intervals between state transitions. The presence of a token on a control transformation indicates that it is actively sampling the values of continuous inputs and awaiting input signals in order to change state and generate outputs according to its STD logic.

The placement of tokens on the input prompts of a control transformation dictate whether the transformation carries a token. The token on the input prompt is always removed. A control transformation can be enabled, disabled or suspended, as a result of the receipt of input prompts. A control transformation with an Activate input, or matching Activate and Pause inputs, carries a token while it is enabled. Control transformations with no input prompts always carry a token.

The placement of a token on an input signal flow of a control transformation causes an interaction between the flow and the transformation which always results in the removal of the token from the flow. If the control transformation carries a token, a state change may occur, as specified by the STD, possibly resulting in a change of state, and the placement of tokens of output signal flows. State changes may also occur depending on the values of continuous inputs.

The placement of tokens on the input prompts of a control transformation often results in the placement of tokens on output prompts so that the effect of the input prompt may be propagated to other transformations. This has been defined by the STDs of Figures 3.3 and 3.4.

3.5.4 Stores

A non-depletable store carries a token whenever it contains data, i.e. after it is written. Depletable stores can contain multiple tokens up to the store capacity. The number of tokens in a depletable store equals the number of units of information stored within.

3.5.5 Multiple Token Placement

The Ward execution rules use special features to deal with circumstances which result in simultaneous placing of several tokens. Examples of these circumstances include the simultaneous placement of tokens on several outputs by a flow transformation, the simultaneous placement of tokens on a diverging output flow, or simultaneous placement of tokens on several signal outputs of a control transformation as a result of a state transition. The Ward execution rules require carrying out the placements and resulting interactions sequentially, but in arbitrary order. Each branch of the interaction is carried out to its conclusion before returning to the next. If more branches are encountered during an interaction, another arbitrary sequencing decision is made and the procedure is applied recursively. This is therefore a *depth-first search* with *random selection* and *backtracking* scheduling strategy.

The ESML execution rules dispense with the depth-first random selection and backtracking scheduling strategy of Ward. This scheduling policy limits possible model concurrency by requiring that branches be carried out sequentially in random order. The execution rules of Ward have been criticised for not directly supporting the execution of behavioural models of *distributed real-time systems*. The Ward scheduling strategy assumes that an external event entering the system has to be completely served before the next event can enter. A distributed real-time system is intrinsically concurrent, any implementation must respect this property. The ESML execution rules adopt a scheduling policy that allows more concurrency. Where multiple token placement occurs, it is undetermined what branch is followed, indeed several branches may be followed at the same time. Execution may begin on one branch before another branch has completed, the interaction followed is chosen randomly. This results in a more distributed execution pattern which now reflects the nature of the distributed system.

3.6 Summary

This chapter has introduced ESML for use as the prototype specification language in the ESML/LOOPN prototyping system. It has been noted that ESML is *intuitive* and easy to use. A set of formation rules have been presented for ESML. The behaviour of control transformation as they receive various sequences of prompts has been defined using STDs. The Ward/Mellor RTSA/SD development method has been introduced, it can be used to guide the use of ESML in the specification process. It has been noted that RTSA/SD and OOA/OOD are compatible. The chapter has defined a set of execution rules for ESML, they provide guidelines on how ESML specifications are to be executed, the rules have been specified in terms on Petri net tokens. A simple scheduling policy has

been adopted to allow more concurrency in the execution of an ESML specification. The ESML/LOOPN prototyping system uses LOOPN nets to define an execution semantics for ESML according to the execution rules. The following chapter describes Petri nets, focusing on LOOPN nets and the LOOPN code generator.

Chapter 4

Petri Nets

4.1 Introduction

The previous chapter has introduced ESML and defined a set of execution rules for it. The objective of this chapter is to introduce Petri nets, in particular LOOPN nets and the LOOPN code generator. The chapter starts with a description of classical (low-level) Petri nets. Classical net structure is defined, the transition firing rule is illustrated by a simple example. Formal analysis of Petri nets is also discussed. The chapter then describes high-level Petri nets, focusing on LOOPN nets which are illustrated using the dining philosophers problem. LOOPN nets are used to define an execution semantics for ESML in the ESML/LOOPN prototyping system in Chapter 5. The chapter then concentrates on Petri net simulation and implementation strategies. The LOOPN code generator is described in detail, it is used in the ESML/LOOPN prototyping system to generate automatically a C language program from a LOOPN net specification. A number of other Petri net code generators are also surveyed.

4.2 Classical Petri Nets

Petri nets [Petri66] are a graphical and mathematical tool suited to the modelling of many types of reactive and real-time systems. As a graphical tool, Petri nets can be used as a visual communication aid, but unlike most visual languages, Petri nets have a mathematical basis, *multi-set theory*, facilitating formal analysis, simulation and automatic implementation. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behaviour of the modelled system.

4.2.1 Classical Petri Net Structure

A *classical Petri net (low-level Petri net)* [Murata89] is a 5-tuple, $N = \{P, T, F, W, M_0\}$, where

$$P = \{p_1, p_2, \dots, p_m\}$$

$$T = \{t_1, t_2, \dots, t_n\}$$

$$F \subseteq (P \times T) \cup (T \times P)$$

$$W: F \rightarrow \{1, 2, 3, \dots\}$$

$$M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$$

$$P \cap T = \emptyset$$

$$P \cup T = \emptyset$$

P is a finite set of *places*, T is a finite set of *transitions*, F is a *flow relation* (set of directed arcs), W is a *weight function* on arcs and M_0 is the *initial marking*. A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted by N . A Petri net with a defined initial marking is denoted (N, M_0) , a *marking* is an assignment of *tokens* to places. The net marking varies during Petri net execution according to the *transition firing rule*. Multiple arcs may connect the same place to the same transition and vice-versa. In a graphical sense, places are drawn as circles, transitions as rectangles, and tokens as dots in a place.

4.2.2 Transition Enabling and Firing

A Petri net can be considered to be a game board where tokens are markers which can only be positioned on places [Jensen90]. Each transition represents a potential move in the "*Petri net game*". The Petri net transition firing rule determines when moves can be made and so determines the dynamic behaviour of the net.

- (a) A transition t is said to be *enabled* if each input place p of t is marked with at least $w(p, t)$ tokens, where $w(p, t)$ is the weight of the arc from p to t .
- (b) An enabled transition may or may not fire.
- (c) The firing of an enabled transition removes $w(p, t)$ tokens from each input place p of t , and adds $w(t, p)$ tokens to each output place p of t , where $w(t, p)$ is the weight of the arc from t to p .

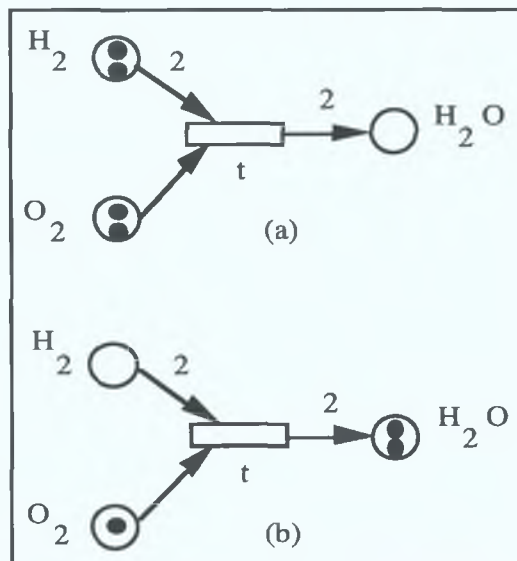


Figure 4.1 Classical Petri net modelling of the chemical reaction: $2\text{H}_2 + \text{O}_2 \rightarrow 2\text{H}_2\text{O}$

The transition firing rule is illustrated by Figure 4.1 which models the chemical reaction for the production of water. Two tokens in each input place in Figure 4.1(a) indicate that two units of H_2 and O_2 are available, and that transition t is enabled. After firing t , the marking changes to the one shown in Figure 4.1(b), t is no longer enabled.

For the above transition firing rule, it is assumed that each place can accommodate an unlimited number of tokens. Such a Petri net is referred to as an *infinite capacity net*. For modelling many systems, it is natural to consider an upper limit to the number of tokens that a place can hold. Such a Petri net is referred to as a *finite capacity net*. *Condition Event nets* (CE-nets) [Petri66] are finite capacity nets whose places can hold at most 1 token. The places of *Place Transition nets* (PT-nets) [Peterson81] can hold multiple tokens.

Petri nets are inherently *concurrent* or *parallel*. The enabling of a transition depends on local conditions only, transitions at disjoint locations in the net can fire concurrently. A *step* is a mathematical description of the set of concurrently enabled transitions. The elements of a step are the transitions which can fire simultaneously at this marking. A transition may even fire concurrently to itself, depending on the availability of enough tokens. In general each step is a *multi-set* over the set of all transitions. A multi-set is analogous to a set, except that it may contain multiple appearances of the same element. It should be noted that two transitions are concurrently enabled if they are independent in the sense that they operate on disjoint sets of tokens; transitions in conflict cannot fire concurrently. When a step with several transitions occurs, the effect is the sum of the effects of the individual transitions.

4.2.3 Classical Petri Net Analysis

A major strength of Petri nets is their support for *formal analysis* [Jensen81] [Murata89]. Two types of properties can be studied with a Petri net model [Murata89]: those that depend on the initial marking (*behavioural properties*), and those that are independent of the initial marking (*structural properties*). Murata has described a large number of behavioural properties and has given a mathematical definition of each. A sample behavioural property is *reachability*. It involves determining whether certain markings are reachable from the current marking by the firing of transitions. Reachability is the most fundamental property for studying systems modelled using Petri nets. Reachability can be used to check whether a system can deadlock.

4.2.4 Advantages and Disadvantages of Classical Petri Nets

The main advantages of classical Petri nets stem from their mathematical foundation. Classical Petri nets have many well developed formal analysis methods which allow a designer to reason about a system before implementation.

Classical Petri nets have been criticised for a lack of intuitiveness, they lack any concept for data representation. For practical applications it has turned out that classical Petri nets are too low-level to cope with many real-world applications in a manageable way. They are only practical for modelling small systems, since the net grows exponentially for even moderately sized systems.

4.3 High-Level Petri Nets

To improve the practicality of Petri nets, while retaining analytical power, *high-level Petri nets* were developed. High-level Petri nets allow tokens to carry complex information, and so provide improved modelling convenience. *Predicate Transition Nets* (PrT-nets) [Genrich81] were the first kind of high-level Petri net constructed without any particular application area in mind. The step from classical Petri nets to high-level Petri nets can be compared to the step from assembly languages to third generation programming languages which contain an elaborated type concept.

A large number of high-level Petri net classes have been defined in the literature. They differ in the inscription language used, which can be based on *algebraic* [Reisig91] [Battiston88], *object-oriented* [Bruno86] [DiGiovanni91], or *functional* [Jensen92] concepts. Most high-level Petri nets include some notation for representing time, so facilitating the modelling of the idiosyncrasies and timing dependencies that are commonly found in real-time systems.

4.3.1 LOOPN Nets

LOOPN (Language of Object-Oriented Petri Net) [Lakos90] is a language for specifying systems in terms of High-level Timed Petri nets. It includes object-oriented features such as *subtyping*, *inheritance* and *polymorphism* which allow for the convenient modularisation of complex specifications. LOOPN nets are currently specified textually in the LOOPN source language. A graphical interface, XLOOPN, is under development which will allow the graphical specification of LOOPN nets.

As a simple example, consider the famous *dining philosophers problem*. A group of philosophers are seated at a round table, with one fork between each pair of philosophers, and one bowl of spaghetti in the centre of the table. Initially, all the philosophers are thinking and all the forks are free. At random intervals each philosopher becomes hungry and decides to eat. This is possible if the philosopher can get hold of the fork on each side. This will not be possible if an adjacent philosopher is eating.

The LOOPN net for the dining philosophers problem is shown in Figure 4.2. The high-level net contains five places (circles) and four transitions (rectangles). A place can contain tokens whose type, the *token colour*, must belong to the *colour set* of the place (place type). Colour sets not only define possible token colours, but also define operations and functions which can be applied to the colours. Each place in Figure 4.2 carries tokens which are marked with a number in the range $1..n$. Place names are located beside the place. If the place has an initial marking it is defined within brackets following the place name. Transition names are specified within the rectangle. Arcs may be annotated with expressions (*arc inscriptions*) specifying the particular tokens which need to be transferred on transition firing. Arc inscriptions are specified within brackets beside the arc. The net inscriptions may include named variables or constants prefixed by "#". Transitions can be annotated with additional *input token selection sections* which must satisfy if the transition is to fire, this allows the firing of a transition to depend on the values of input tokens. Transitions may also include special *output actions* to be performed on output token values at the time of firing. To avoid cluttering, the transition input and output token sections (if any) are specified beneath the diagram.

As an example, consider the transition "*takeRight*". For this transition to fire a token for philosopher i must be present in the place "*hasLeft*" and the fork numbered $i+1$ must be available in the place "*freeForks*" (with addition *modulo n* assumed). On firing this transition a token for philosopher i is added to the place "*eating*". The colouring of tokens in this case means that the one net structure serves for an arbitrary number of philosophers, whereas the use of colourless tokens (classical Petri nets) would require duplication of the net structure.

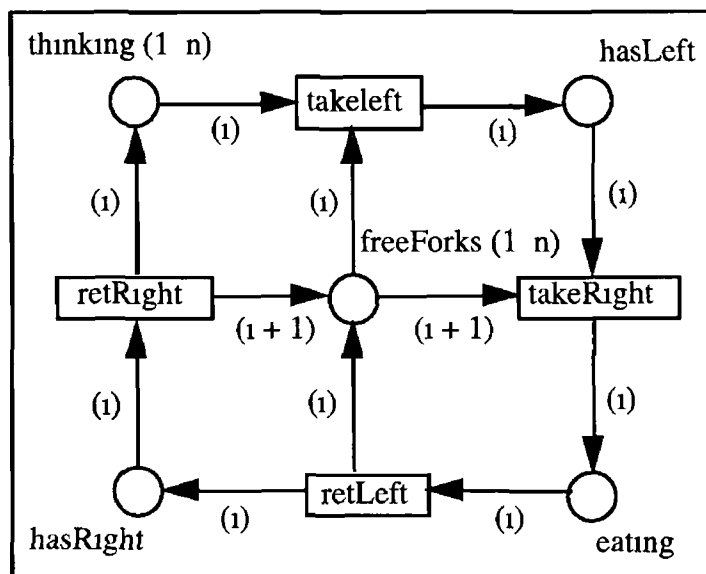


Figure 4.2 LOOPN net for the Dining Philosophers Problem

4.3.1.1 Type Declarations

Type declarations in the LOOPN source language include a subset of Pascal type declarations, in particular, *enumeration*, predefined types (*integer*, *real*, *boolean*, *char*), *strings*, *single dimensioned arrays*, and *records*. *Token types* are defined as extended record structures which encapsulate a set of data fields (each of some basic type) and a set of associated functions in a single object. The data fields determine the range of possible colour combinations that tokens of this type may assume. A place holds a list of tokens of a specified token type. Each time a token is added to a place it is timestamped with the *current simulation time*.

In true object-oriented style, a token type is declared as a *subtype* of one or more other token types, and thus *inherits* the parent's data fields and functions (i.e. its *features*). It may augment the features of the parent and may override the parent's defined functions. With token type inheritance comes *polymorphism*, a token of some subtype can be used where tokens of a parent type are expected.

The most elementary token type, from which all others inherit, is the *"null"* type. While it has no data fields, it includes definitions of the functions *"first"*, *"last"*, and *"delay(t)"*, which are available to every other token type. The functions *"first"*, *"last"* are parameterless functions returning *true* if the specified token is the first, respectively last, token to arrive at the place. The function *"delay(t)"* returns *true* if the token has been resident at the place for time *t*. The token type declaration for the dining philosophers net could be

TYPE

```
phil_num = 1 n,  
phil_type =  
  TOKEN null WITH  
    ph phil_num,  
    next = phil_num self ph mod n + 1,  
    has (p phil_num) = EXISTS x x ph = p,  
END,
```

This declares a token type which is a subtype of "*null*", with a field holding a philosopher number. The function "*next*" returns the number of the neighbouring philosopher. The identifier "*self*" can be used to refer to the current token. The function "*has*" can be used to determine if a place contains a specified philosopher. The existential quantifier *EXISTS* ranges over all tokens currently in the place.

4.3.1.2 Places

Places are declared to be of a specific *token type*, thus restricting the tokens which may reside in the place. Place declarations may also specify an optional *place restriction* which imposes a universal filter on the place. Only those tokens satisfying the restriction are visible in the net. This universal filter is useful for localising or encapsulating the desired place behaviour, but it can be considered simply as syntactic sugar for the conjunction of this condition to every transition taking input tokens from the place.

LOOPN supports *timed places*, tokens become visible in a place when the delay has expired, the delay is specified using the "*delay(t)*" function in the appropriate place restriction.

4.3.1.3 Transitions

Transitions specify *input places*, *output places*, and *auxiliary actions*, each of which are optional. Each input token is explicitly named together with the place from which it is derived. There is also an optional condition which the selected token must satisfy in order for the transition to fire. Such conditions are equivalent to the global restrictions applied to places, except that the restriction of token visibility is now local to the transition rather than global.

The transition output places are also explicitly named, together with the tokens which are to be added to these places. All token identifiers have scope local to the transition. The token values may be copies of existing (input) tokens, copies of existing

tokens with certain named fields changed, or newly generated tokens with values specified for named fields. The optional auxiliary action of a transition may consist of one or more procedure calls. It does not affect the firing of the net, but allows interaction with the environment, typically it is used to report the progress of the simulation.

4.3 1.4 Modules and Module Instances

A LOOPN net can be coded as one or more *modules*, each of which consists (in the simplest case) of constant, type, place, and transition declarations. Immediately preceding the transitions is the initialisation for the module. Syntactically, this is similar to a transition with no input places. Semantically, it is executed once at start-up to establish the initial module marking. Modules provide a notation for the hierarchical decomposition of the LOOPN net model.

The LOOPN module specification for a dining philosophers problem of five philosophers could be as follows:

```

MODULE philosophers (CONST which integer),
  CONST n = 5,
  TYPE
    phil_num = 1 .. n,
    phil_type = ,
  PLACE
    thinking, eating, freeForks,
    hasLeft, hasRight phil_type,
  INITIALISATION,
  OUTPUT
    thinking <-i1 = [ph 1],
    thinking <-i2 = [ph 2],

    freeForks <-i1,
    freeForks <-i2

  TRANSITION takeLeft,
  INPUT
    i <-thinking,
    j <-freeForks | j ph = i ph,
  OUTPUT

```

```

        hasLeft <-i,
ACTION
printf ("Philosopher %d has raised his left fork", i ph),

```

END MODULE

Complex Petri nets can be built by including module instances in the definition of other modules. The declaration of an instance effectively duplicates the associated subnet, including its places, transitions and nested instances.

Modules can interact with their environment in a number of ways. At the simplest level, they may include *constant parameters* so that an instance can determine its identity (out of a number of instances). Secondly, a module may interact with its environment by declaring *parameter places*. Here the interaction with the environment is by transferring tokens into and out of the module. Because of this interface to places, modules may be thought of as *super transitions*. Parameter places may be declared with usage *INPUT*, *OUTPUT* or *INOUT* (i.e. input and output). Only those parameters with *INPUT* or *INOUT* usage may have a restriction attached, in which case the tokens in that place are only visible within the module if that condition is satisfied in addition to other global visibility restrictions.

LOOPN therefore uses three ways to control token visibility. At the global level, a place can have a restriction which effects the visibility of all tokens in that place. At the local level each transition can decide which tokens are appropriate for itself, by imposing conditions on input tokens. At the intermediate level, modules can specify which tokens are relevant, by specifying restrictions at the module boundary. This three level filtering scheme is depicted in Figure 4.3.

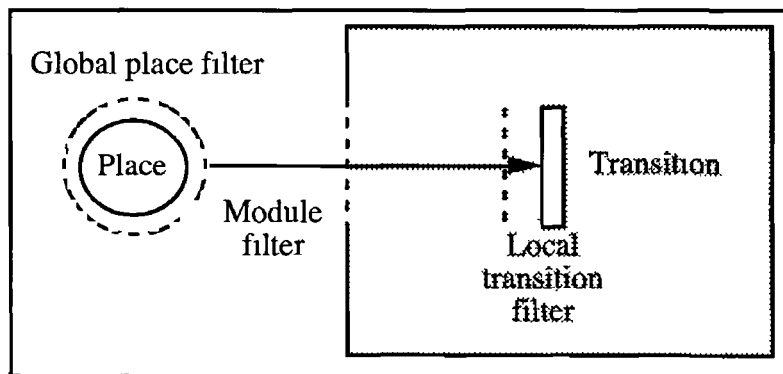


Figure 4.3 Token Filtering in LOOPN

LOOPN supports the definition of *subclasses* of modules with inheritance. A module can be defined to be a subtype of another module and inherit all of the features of its parent module type. A module subtype can augment the features of its parent and can override declared identifiers of the parent. Polymorphism of modules is supported by allowing an instance of a module subtype to be used where an instance of a parent module type is specified. In this way, complex Petri nets may be built by including instances of other modules, and by augmenting or modifying existing modules.

4.3.2 Advantages and Disadvantages of High-Level Petri nets

High-level Petri net offer better modelling convenience than classical Petri nets, the hierarchical techniques supported by most high-level Petri nets allow the system to be modelled at different levels of abstraction, while facilitating top-down and bottom-up model construction. High-level Petri nets offer notations for representing both the data and control structures of the system being modelled. Though high-level Petri nets offer better modelling convenience than classical Petri nets, they can result in nets in which the entire structure of the system is embedded in the net inscriptions. High-level Petri nets have been criticised for lacking intuitiveness, they do not allow modelling of the system at a natural level.

The analysis methods of classical Petri nets have been extended for high-level nets, though not without cost. The methods are not as well developed, though a lot of research is progressing in this area.

4.4 Petri Net Simulation and Implementation

A *Petri net simulator* is an algorithm which "*plays the token game*" [Valette91], i.e. carries out the occurrence of the transitions in accordance with the availability of tokens. Petri net simulators work according to a selected *simulation strategy* (also known as a *scheduling* or *implementation strategy*). A Petri net simulation is usually carried out by a tool which animates the system model as the transitions are fired.

A *Petri net implementation* refers to a programming language program which simulates the net. This program is produced automatically or semi-automatically from the net specification. A tool which generates a Petri net implementation from a net specification is referred to as a *Petri net code generator*. A great diversity of Petri net implementations are discussed in the literature [Silva86]. They differ in the class of Petri net concerned, the target language, and the implementation strategy. The implementation may concern classical or high-level Petri nets, and may be *centralised*, *distributed*, or *hybrid*, each of which can be *compiled* or *interpreted* [Silva89].

4.4.1 Centralised Implementation (C)

Centralised implementation is a simple, though widely used, implementation strategy for Petri nets. One central process, *the manager*, manages all places and tokens, each transition is represented as a separate process. The manager manages an explicit representation of the net marking, and provides indirect synchronisation between transitions. A transition process which wishes to fire communicates its wish to the manager, who in turn checks whether the transition can fire, and communicates the result back to the transition process. If the request to fire was successful the manager updates the marking. Since there is only one process for all places and tokens, i.e. the manager, transitions cannot be fired concurrently.

4.4.2 Distributed Implementation

Distributed Petri net implementations [Taubner88] have been proposed to allow the concurrent firing of transitions. Distributed implementation strategies differ with respect to the grain of parallelism supported.

4.4.2.1 Distribution of Control by Places

Motivated by the structure of Petri nets, an idea which suggests itself is to provide one process per place for the management of tokens. A transition has to communicate with several place processes before permission to fire can be established, reservations have to be made. It is possible to either reserve the whole place, or to reserve individual tokens only.

Conflicts can occur while establishing occurrence permission, a problem not present with centralised implementation. This happens, if two or more transition processes have made some of their reservations and every transition process needs a further reservation but cannot make it because of the reservations of the others. For the resolution of such a conflict there are several possibilities:

- (A) Abandonment of the wish to occur of all involved transitions, in general this leads to an unproductive implementation.
- (W) A winner amongst transitions is elected depending on a priority order on transition processes, the priority alters dynamically.
- (O) Transitions request their places in a fixed order, the reservations are queued by the place process.

Taubner classifies distributed Petri net implementation strategies according to a three letter code (*the Taubner code*), the first letter indicates the control strategy, the

second letter indicates the reservation unit, the third indicates the method of conflict resolution. The Taubner code can be used to classify the general behaviour of many distributed net implementation strategies.

The *PPO strategy* uses one process per place, reservations are made on places, while fixed order reservation ensures freedom from deadlock. A transition process informs the processes which manage its input places about its wish for reservation and waits for the answer. Reservations are made in place number order. After the first refusal or after all places have been reserved, the transition process cancels the reservations, or initiates the movement of tokens respectively. Both can be done for all places in parallel.

A place process puts a reservation request in its queue, removes transition processes with too large requests, and checks if the next reservation can be accepted. The cancellation of a reservation and communications for the movement of tokens are handled correspondingly.

PPA can be derived from *PPO* by removing the queue, informing requesting transitions immediately if the place is reserved, and allowing a transition process to run its reservations in parallel. *PPA* has a smaller overhead for managing places, tokens and transitions than *PPO*.

The above strategies can be generalised for tokens as the reservation unit. *PTO* involves smaller communication overhead and allows the greater parallelism in the execution of transitions.

4.4.2.2 Distribution of Control by Edges

The bottleneck in previous implementation strategies has been that a place process can communicate with only one process at a time, even if there are enough tokens, resulting in *communication overhead*. To avoid this bottleneck, the management of a place is now split up into one place centre process and two part processes for management of place input and output transitions respectively. Reservation units, in such strategies, are tokens, since a reservation on a place no longer makes sense. An example is *ETO*, where the E indicates that the distribution of control is determined by the number of edges, T indicates tokens as the reservation units, and O the fixed-order conflict resolution strategy.

4.4.3 Petri Net Code Generators

This section surveys a number of Petri net code generators, i.e. tools which generate computer programs to play the "token game". LOOPN is discussed first, followed by a number of other generators (any of which could be used to replace LOOPN as the back-end of the ESML/LOOPN prototyping system). Generators which produce

distributed implementations are classified, where possible, according to their Taubner code

4.4.3.1 The LOOPN Code Generator

The LOOPN tool supports the centralised and distributed implementation of LOOPN nets. The following paragraphs give overviews of both capabilities

4.4.3.1.1 Centralised Implementation in LOOPN

The LOOPN tool generates a C language program from a LOOPN net specified in its source language. This program runs as a centralised implementation of the specified net

The *LOOPN scheduler* is responsible for implementing the LOOPN net using a centralised strategy, it is equivalent to the manager described in the C strategy. The scheduler must cope with time since tokens can be delayed in places, for this purpose traditional discrete event simulation techniques are employed. The scheduler maintains a list of pending events sorted by event time (time at which the event will happen). Events for the same time are randomly ordered to ensure fairness. Pending events correspond to tokens which are currently delayed, but which are to appear at some time in the future. When the scheduler places a token in a delayed place, the token is timestamped with the current simulation time. The scheduler places a pending event notice on the event queue for this event, the event time being equal to its timestamp plus the delay period. When all enabled transitions for the current time have been fired, the simulation time is advanced to the event time of the next pending event (earliest) on the event list. This guarantees the time ordering of events. If there are no pending events at this stage then simulation is complete

A LOOPN source program is translated in two passes into the target language [Lakos93a]. The parser isolates source-specific details and produces an intermediate *clausal form*. The clausal form is a clausal representation of the LOOPN source program, essentially a binary tree data structure of clauses. The parser uses a special symbol table in the production of the clausal form. It also does syntax-analysis, reporting any errors to the user

The code generator is template driven in conjunction with the clausal form. Templates or scripts for modules, places, transitions and types are used to direct the code generation. The templates consist of text, macro-calls and variable references. The template text is produced as source code output, the macro-calls result in clause-specific code generation functions being called. Variable references are replaced by appropriate values. Macros are specified in the template by a dollar ('\$'), optionally followed by a flag

('+', '*', '^', '>') and followed by the macro name. For each macro name there is a corresponding code generation function of the same name, which is invoked, depending on the macro-flag. The flags having meaning as follows

- \$+ iterate over all components at current level
- \$* iterate over all components at current and inherited levels, excluding the first parent
- \$^ iterate over all components at the current level
- \$> iterate over related components at the current level

The templates are parameterized by including variable references. A variable reference is flagged in a coding template by enclosing the variable identifier within the brackets '\${' '}''. In generating code, a variable reference is replaced by the corresponding value of the variable, which has been defined in the symbol table. The special code generation functions add variable definitions to the symbol table.

4.4.3.1.2 Distributed Implementation in LOOPN

Lakos [Lakos92] has extended the PPO strategy, which was defined for classical Petri nets, for use with LOOPN nets. Reservation requests indicate the specifically coloured tokens required, furthermore, the ability of a transition to fire depends on any transition condition present. In LOOPN nets, tokens may have defined functions as well as data values, these functions may be called in the process of evaluating the transition condition. As a result of this the reservation must request one token at a time, and only when all the tokens are selected will the input condition be evaluated. The strategy uses fixed order reservation order to prevent deadlock, iterating over all possible token selections for each input place until the input condition is satisfied or no possible combination satisfies it.

4.4.3.2 Specification of Concurrent Systems (SPECS)

SPECS [Dahler87] is a tool for the construction, animation and implementation of Petri nets. The tool contains centralised and distributed simulator components [Butler90], a *transputer network* [Fleming88] being the target hardware for the distributed component.

SPECS advocates a two stage approach to developing software prototypes of distributed real-time systems. First, the system is modelled together with a simulation of the system environment, and animated to discover errors. Then the model is transformed and simulated on a transputer network, the real environment is connected to the running

prototype through dedicated I/O interfaces. SPECS automatically generates a configuration file and OCCAM code for each processor. The configuration file specifies the mapping of processes to processors. The distributed prototype execution can be monitored within SPECS.

SPECS uses *FunPrE nets* [Butler90], an extension of *Pr/E* nets with firing functions which consist of conditions and actions, for system modelling. FunPrE net places can hold a maximum of one token only.

The implementation strategy is based on the Taubner PPO strategy. Control distribution centres on places, places are the reservation unit, while fixed order reservation prevents deadlock. Transitions reserve their input places, and then calculate the firing condition based on input tokens. If a combination of input tokens has been selected which satisfies the firing condition, the transition calculates the values of output tokens and reserves output places. If output places are reserved, then token movement is initiated.

The simulator performance depends on how well individual processes are distributed on processors. SPECS uses a net specific configuration tool which employs *simulated annealing* techniques to map processes to processors.

4.4.3.3 Concurrent Pascal with Petri Net (CPN)

Hartung [Hartung88] proposes a language for programming multiprocessor systems which combines Concurrent Pascal with high-level Petri nets (CPN). The language is implemented on the *M5PS* multiprocessor system which consists of eight closely coupled multiprocessor (*CCMp*) systems built from standard microprocessors. Global memory is used for inter-process communication.

The proposed language treats parallelism implicitly, as defined in the net part of a CPN program. The language contains constructs for the textual definition of places and transitions, together with inscriptions, enabling conditions and transition actions. A special class of common object is used to allow concurrent access to a shared data structure.

The parallel execution of nets described using CPN is based on a PTO strategy. Tokens and places are stored in common global memory, *player processes*, which are allocated to processors, execute a parallel and de-central token player strategy. The players can execute any transition they wish, and so transitions do not have to be statically allocated to processors. This flexibility allows the work to be shared evenly amongst the processors.

The player process chooses a transition to fire by a simple round robin method, and tries to reserve all adjacent places in a pre-defined order. If the reservation fails, then

the player process picks another transition. If reservation was successful the player tests the transition enabling condition by using combinations of input tokens. If the transition is enabled by any combination, the player takes all input tokens and reserves space for output tokens in output places. After the transition action has completed firing, output tokens are entered in output places and all place reservations are released. The reservation strategy used involves trying to reserve all places in a pre-defined order. If during the reservation process, a place is found that is already reserved, then all reservations are released. By using a pre-defined order on reservations livelock is prevented. By releasing reservations when a full reservation cannot be made, other processes may reserve places just released.

The strategy can be considered PPO since player processes behave as transitions, places are the unit of reservation, and reservation is done in a pre-defined order to prevent deadlock/livelock, though transitions do release reservations if a full reservation on all adjacent places cannot be made.

The problem with the distributed implementation of Petri nets on a multiprocessor architecture is that the global memory is a bottleneck for large Petri nets. The message passing transputer architecture, which has no global memory, is more suited to the distributed nature of Petri nets.

4.4.3.4 AMI

AMI is a software environment for the specification, validation and implementation of distributed real-time systems modelled using Petri nets. *AMI* consists of several Petri net analysis tools and two code generators, *TAPIOCA* [Breant90] and *PN_TAGADA* [Kordon90] for OCCAM and Ada code generation respectively.

4.4.3.4.1 TAPIOCA

TAPIOCA [Breant90] generates a distributed OCCAM code implementation of Petri nets specified in the *AMI* software environment. The *TAPIOCA* application uses an *architecture description* of the target hardware, a *mapping specification*, and the *Petri net model* itself, in the production of OCCAM code. The application has recently been extended to allow the distributed implementation of *Coloured Petri nets* (CP-nets) [Jensen90].

The code generated by *TAPIOCA* can be imported into the *Transputer Development System* (TDS), and compiled and executed on the transputer network. The code has been executed on a *B012 Inmos board* [INMOS88] containing five transputers, connected with a B004 board, and plugged into a PC. Default transition stubs, which are

called on transition firing, can be manually filled in by the user. TAPIOCA also generates a configuration file which is used to set-up the transputer network.

TAPIOCA proposes a three step prototyping method [Breant91], each of which gathers information needed by the code generator. The process is based on using the results of *formal model analysis* to decompose the model into an optimised number of state machines, each of which can be implemented as a sequential process. TAPIOCA is proposed in opposition to implementing each transition as a process, which though distributed, increases drastically the number of processes and conflicts. TAPIOCA cannot therefore be described using a three letter Taubner code.

The first step, *the Analysis/Translation step*, involves using *linear invariants* [Jensen81] found during the model analysis phase to decompose the Petri net into a set of state machines, linked by synchronous and asynchronous interaction mechanisms. The resultant decomposition is specified as the *interaction net*. The decomposition into a set of state machines is based on the premise that each place belongs to, at most, one state machine, and any transition belongs to at least one state machine. The use of model invariants results in a more optimised implementation. The *asynchronous interaction mechanism* (AIM) models a place which does not belong to any state machine. *Synchronous interaction mechanisms* (SIM) model multiple rendezvous, and correspond to a transition shared by many state machines. The interaction net describes the net decomposition, nodes represent state machines or buffers, arcs represent interactions. The interaction net serves as the basis for distributed implementation.

The second step, *the Object Location step*, builds a *mapping net* from the interaction net by applying a set of rules. The mapping takes into account the software environment, the architecture constraints, and the model structure. Each state machine and buffer becomes an OCCAM process, moreover a supplementary process is generated for each SIM.

The third and final step, *the Code Generation step*, integrates target language constraints, generating three types of OCCAM process, state machine processes, data manager processes, and rendezvous manager processes. State machine processes sequence transition firings. Transition firing involves precondition evaluation (input places), done as a result of communication with data managers and other state machine processes. Once the transition is fired, tokens are sent to postconditions (output places), this being done by the sending of messages to data managers.

4.4.3.4.2 PN_TAGADA

PN_TAGADA (Petri Net Translation, Analysis and Generation of Ada code) [Kordon90] uses the same three step approach, as described for TAPIOCA, to generate automatically a distributed Ada code implementation of a CP-net. The method, as above, is based on using linear invariants to decompose the model into an optimal number of sequential processes (state machines)

4.4.3.5 PROTOB

PROTOB [Baldassari91] supports the automatic generation of Pascal, C, Ada, and OPS5 prototypes from *PROT net models* of real-time systems [Bruno86]. *PROT* nets are a type of object-oriented high-level Petri net.

The *PROTOB* translator translates the *PROT* net model into the target code. The translator generates and compiles a module for each *PROT* net. This module is linked with the simulator kernel to produce the executable code. The translator is able to produce code according to centralised or distributed implementation strategies, in the latter case several processes are generated which run on the workstations of a *local area network*, the distributed implementation code is installed automatically by the translator according to the system configuration requirements.

For the generation of a distributed (Ada) implementation, process types are considered as ordered collections of places of the same type, tokens represent process instances, tokens range over the places described by the process type. The places of a process type identify process states. Process types are represented by a data structure which identifies process instances, and the place types. The presence of a token in a process place indicates that the process instance is in that state.

The translation of *PROT* nets into Ada program structures involves two steps, the identification of process types, and the implementation of synchronisation according to the logic expressed in the net.

Process types are identified by performing a *depth-first search* of the *PROT* net. The process type places can be scattered throughout the *PROT* net, places of the same type may not be contiguous at transitions. Processes are cyclic by nature, and are equivalent to the circuits of the net, a circuit being a directed path where no vertex appears more than once. The strategy builds a process tree from a *PROT* net which contains the reduced feasible main circuits of a given process type.

Transitions cause process instances to change state. When a transition has both an input place and an output place belonging to the same process type, the corresponding states of the process are to be synchronised with the states of other processes which

participate in the same transition. When a transition has only an input place of a given type, the firing of the transition results in the process being suspended. A transition with only an output place of a given process type resumes the execution of a suspended process when it fires. Transitions are implemented as Ada tasks. Process instances are also implemented as tasks, they cycle continually through their states, state change only occurs after synchronisation by rendezvous with the appropriate transition task.

The implementation strategy visualises tokens as active process instances, and so cannot be described using a Taubner three letter code.

4.4.3.6 PROMPT

PROMPT (PROtocol Manufacturing Prototyping and Testing) [Parker90] is a tool for the specification and centralised implementation of high-level Petri nets [Lai91].

PROMPT produces C code automatically for a textual net specification written in XNL (*eXtended Net Language*). XNL supports *modularity*, *transition folding* and *data encoding*. PROMPT includes tools for model debugging, tracing and trace file analysis.

4.5 Summary

This chapter has described Petri nets, in particular LOOPN nets and the LOOPN code generator. LOOPN nets are used to define an execution semantics for ESML within the ESML/LOOPN prototyping system. The LOOPN code generator is used in the same prototyping system to generate a C language program from a LOOPN net specification. The chapter has outlined the advantages and disadvantages of classical and high-level Petri nets. It has been noted that though Petri nets are graphical they lack intuition, even high-level Petri nets are difficult to use. However, they are executable, amenable to formal analysis, and can have computer programs generated automatically to play their "token games". The chapter has described how distributed net implementation strategies can be classified according to their Taubner code. Several code generators, including SPECS, CPN, AMI, PROTOB, and PROMPT, are discussed. The tools which facilitate distributed net implementation are classified according to their Taubner code where relevant. Any of those tools could replace LOOPN as the code generator component in the ESML/LOOPN prototyping system, e.g. an execution semantics could be defined for ESML in terms of CP-nets and the TAPIOCA tool used to produce an OCCAM language prototype to run on a transputer network (i.e. the ESML/TAPIOCA prototyping system). The following chapter defines the ESML/LOOPN prototyping system.

Chapter 5

The ESML/LOOPN Prototyping System

5.1 Introduction

The previous chapter has described LOOPN nets and the LOOPN code generator. This chapter defines the ESML/LOOPN prototyping system. The chapter starts with a survey of related work in the area. It then presents an overview of the prototyping system before outlining in detail how to produce exploratory prototypes of real-time systems for use within a keep-it prototyping approach. The prototyping system uses a set of translation templates to translate non-executable ESML specifications into executable LOOPN specifications. The translation templates therefore define an execution semantics for ESML based on the guidelines set down by the ESML execution rules in Chapter 3. The bulk of the chapter concerns the translation templates. They provide each component of the ESML language with an LOOPN net equivalent. A step by step guide on how to apply the translation templates is described.

5.2 Related Work

The ESML/LOOPN prototyping system advocates the translation of *non-executable* ESML specifications into *executable* LOOPN specifications so that ESML can be used as a *graphical executable specification language* for the prototyping of real-time systems.

The combination of less formal languages, such as the RTSA/SD languages, with more formal languages, such as Petri nets, has gained some attention in the literature [Blumofe88] [Martin93] [Pulli86a]. At least two separate approaches can be taken. The first approach consists of associating formal semantics with SA/SD constructs [Tse88]. The second approach has been to translate the SA/SD constructs into more formal language equivalents [Pulli86b]. The following paragraphs mention some of the work performed in each of these areas.

France [France92] proposes two types of extended DFDs: *control-extended DFDs* (C-DFD); and *semantically extended DFDs* (ExtDFD). A C-DFD is a DFD supplemented with a notation for describing control dependencies amongst the DFD elements. An ExtDFD is a C-DFD with a formal algebraic semantics. A semi-automatic approach to generating ExtDFDs is suggested. Martin has proposed a process for the translation of SA

specifications into LOTOS specifications, to ease the introduction of LOTOS into the development process

The most common language used to add formality to SA/SD notations has been Petri nets. They form a natural choice, since the Ward and ESML execution rules have been specified in terms of Petri net tokens.

Lee and Tan [Lee92] have used Petri nets to analyse and validate DFD specifications. Tse and Pong [Tse88] have provided a formal theoretical framework for DFDs through extended Petri nets, the new diagrams being referred to as *Formal Data Flow Diagrams* (FDFDs). FDFDs combine the theoretical power of Petri nets with the user-friendliness of DFDs. Petri nets have also been used to define an execution semantics for SDL (the Specification Description Language) [Munemori88]. Sacha [Sacha91] [Sacha92] describes the use of Petri nets for implementing PAISLey specifications, a number of behaviour preserving transformations that can be applied to Petri nets are described. Auer [Auer88] has developed a tool for the automatic prototyping of real-time systems modelled using the Ward/Mellor transformation schema. The tool generates PL/M-86 programs, and produces a Petri net equivalent which can be used for reachability analysis. *Teamwork/ES* (Executable Specification) is an interpreter which has been developed by Blumofe to support the token-based execution of the Ward/Mellor transformation schema. Brackett and Reilly [Brackett87] have developed an interpreter which interfaces with *Teamwork/ES* to allow prototype execution of the Ward/Mellor transformation schema. The tool automatically translates a system model, stored in the *Teamwork/ES* database, into an OPS5 program. This program can be interpreted using the OPS5 inference engine. Coomber and Childs [Coomber90] have developed an interpreter which supports the prototype execution of the Ward transformation schema. It requires mini-specifications to be written in Smalltalk. Pulli has developed an automated process for the translation of systems specified using the Ward/Mellor transformation schema into Smalltalk-inscribed high-level Petri nets. The resultant Petri net can be interpreted using the SPECS tool, which implements a special two-level Ward scheduling algorithm. This process also requires mini-specifications to be written in Smalltalk. Elmstrom [Elmstrom92] is currently devising a similar translation process which generates high-level Timed Petri nets as its target. It requires mini-specifications to be written in VDM-SL, an executable subset of VDM. Pinci and Shapiro [Pinci91] propose a software development methodology based on a combination of *SADT* (Structured Analysis and Design Technique) [Marca88] and *Hierarchical Coloured Petri nets* (HCP-nets). The system is partially specified in SADT, and translated into HCP-nets.

for simulation and generation of ML code prototypes. The development methodology has been applied to NORAD Command Post application [Shapiro90]

5.3 The ESML/LOOPN Prototyping System

This section defines the *ESML/LOOPN prototyping system*. An overview of the system is first presented followed by detailed explanation of how to build exploratory prototypes using it.

5.3.1 Overview

The ESML/LOOPN prototyping system has been defined to facilitate the construction of exploratory prototypes of real-time systems for use within a keep-it prototyping approach. The approach taken has been to use LOOPN nets to define an execution semantics for ESML specifications according to the guidelines set down by the ESML execution rules. LOOPN nets are used to provide a *rigorous interpretation* of ESML language constructs and their combinations. LOOPN nets have been chosen for defining an execution semantics for ESML since there are a high-level object-oriented Petri net formalism, mirroring the ESML execution rules which have been specified in terms of Petri net tokens. The prototyping system uses ESML as the prototype specification language. The advantages of graphical languages such as ESML are their intuitive nature and ease of use (Chapter 3). The execution semantics has been defined for ESML to allow its use as a graphical executable specification language.

At the heart of the ESML/LOOPN prototyping lies a *template-driven translation process* which is used to translate non-executable ESML specifications into executable LOOPN specifications. The LOOPN tool is used to generate automatically a C language program from its input net specification, this program runs as the prototype in the target environment. When the translation process (the application of the translation templates) is automatic the presence of LOOPN is invisible to the user. The user is only concerned with specifying the prototype in ESML, and exercising it in the target environment.

LOOPN could be used to define an execution semantics for other non-executable languages. The whole concept is similar to the compilation of different programming languages into the same underlying machine language. Programming language compilation involves describing programming language constructs in machine language, here LOOPN nets provide a rigorous interpretation of the ESML constructs according to the ESML execution rules.

5.3.2 The Prototyping Process

The ESML/LOOPN prototyping system consists of three components, *languages*, *methods* and *tools*. The languages used within this prototyping system are ESML and LOOPN nets. The method used is the Ward/Mellor RTSA/SD method, it guides the ESML specification effort. The tools used are TurboCASE¹ and LOOPN. TurboCASE supports the construction of ESML specifications. LOOPN, as described in Chapter 4, is a code generator for LOOPN nets.

The ESML/LOOPN prototyping system, as depicted in Figure 5.1, provides a step by step guide on how to build exploratory prototypes of real-time systems. The first step involves manually specifying the prototype in ESML. The Ward/Mellor RTSA/SD method guides the specification effort. The prototyping system needs to address the execution of mini-specifications of primitive flow transformations. Mini-specifications can be implemented in the output token generation section of the LOOPN transitions, which allow output token values to be specified using LOOPN expressions. If the mini-specification is simple, it can be implemented directly using the LOOPN source language. If the mini-specification is complex, it needs to be coded externally in the target language (C in this case) as part of the prototype specification effort. The external C function can be then called during evaluation of the LOOPN expressions used in the transition output token generation section. Once the prototype specification is complete, the non-executable ESML specification needs to be translated into an executable LOOPN specification. The translation is achieved by successively applying a set of translation templates (section 5.4) to the ESML specification to produce an executable LOOPN net specification. The translation process produces a graphical LOOPN net specification which currently needs to be specified manually in the LOOPN source language.

Once the LOOPN source program is complete, the LOOPN tool is used to generate automatically the C program which will run as the prototype in the target environment. The LOOPN tool achieves this by first parsing the LOOPN source program into an intermediate clausal form, and then generating the target code (section 4.4.3.1). The parsing and code generation phases are automatic. The ESML execution rules specify a random execution order in the case of multiple token placement (section 3.5.5). Petri net scheduling strategies randomly select the transition to fire from a set of enabled transitions, a native Petri net scheduling strategy can therefore be used to direct the execution of the generated prototype. This is the case in the ESML/LOOPN prototyping system, the prototype generated from the LOOPN tool executes the Petri net according to

¹ TurboCASE is a trademark of Structsoft Inc

a native Petri net scheduling strategy (currently a centralised scheduling strategy (section 4.4.1)) This contrasts with systems such as those of Pulli which require an implementation of the Ward execution strategy to limit the possible concurrency which the generated Petri net would exhibit if allowed to execute according to a native Petri net scheduling strategy

The ESML prototyping system therefore facilitates the use of ESML as a graphical executable specification language It does not specifically address how the prototype is to be used within a prototyping approach to software development, though it has been noted that ESML can only be currently used to build exploratory prototypes of real-time systems, and that a keep-it prototyping approach is necessary for such systems

5.4 The Translation Process

This section describes how to use the translation templates to translate a non-executable ESML specification into an executable LOOPN net specification

The translation templates (described in sections 5.4.1 to 5.4.4) provide each of the ESML constructs with a LOOPN net equivalent, i.e. they model each ESML construct using a LOOPN net fragment The templates are based on the guidelines set down by the ESML execution rules The translation process uses LOOPN net modules to partition the generated LOOPN net specification, every control and flow transformation is specified as a separate LOOPN net module The modules are connected together to form the overall LOOPN net specification through their input and output flows Templates are defined for flows, stores, flow transformations and control transformations The templates are generic in nature, i.e. they describe a typical ESML construct and its LOOPN net equivalent When applying the translation templates the user fills in the actual detail (e.g. actual flow names and types in the case of control and flow transformations) To translate a non-executable ESML specification into an executable LOOPN net specification the user needs to examine the ESML specification and proceed as follows

Step 1. Identify all flows, determine the flow type (discrete data, signal, prompt, continuous data) and apply the appropriate translation template This step must also consider any flow convergence and divergence and the connection of flows to terminators The templates for this step are described in section 5.4.1

Step 2: Identify all stores, determine the store type (depletable or non-depletable), and apply the appropriate template The templates for this step are described in section 5.4.2

Step 3: Identify all flow transformations, determine the flow transformation type (i.e. no input prompt, Trigger input prompt, Activate input prompt),

and apply the appropriate template to build the LOOPN net module. The templates for this step are described in section 5.4.3

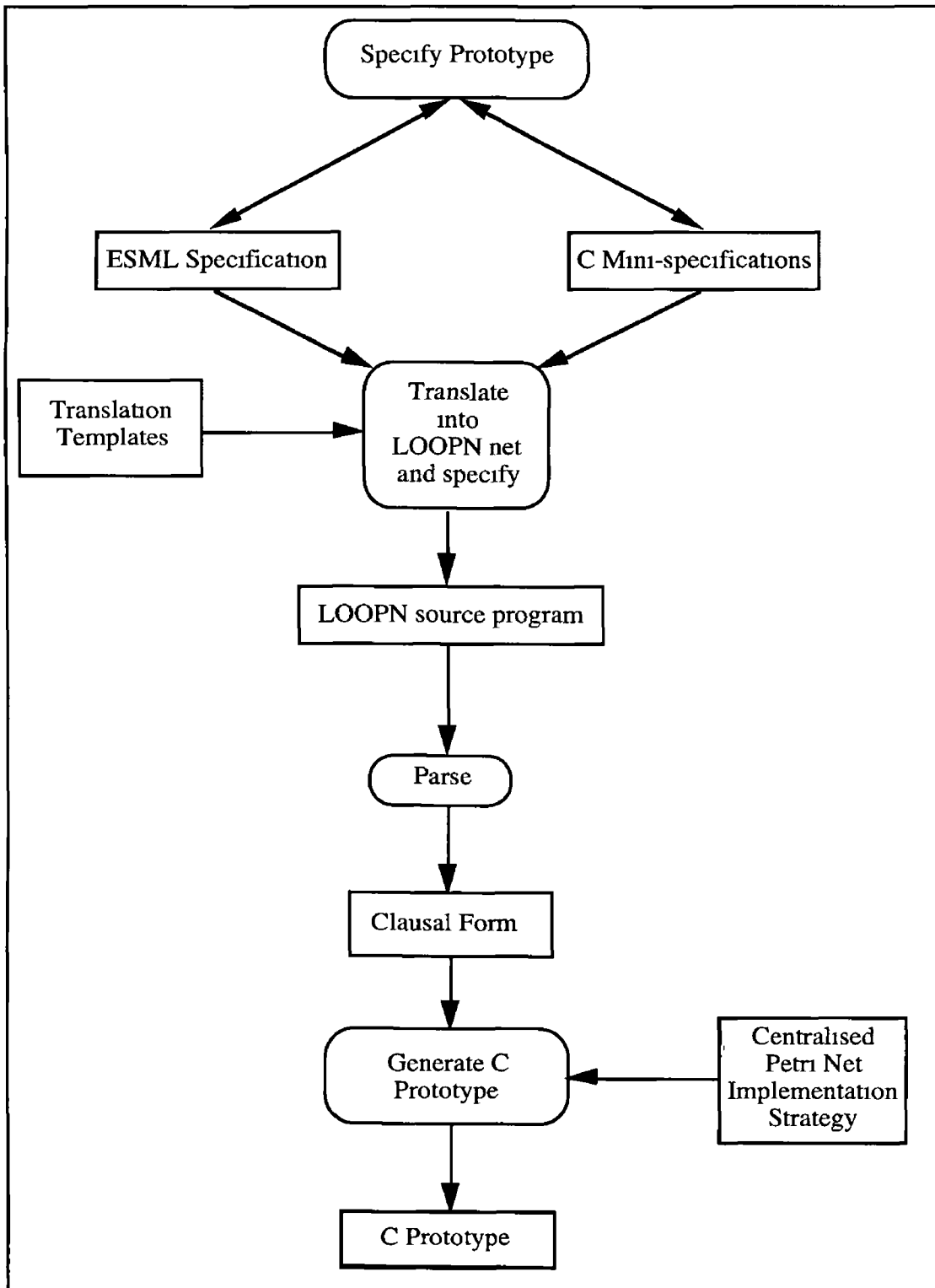


Figure 5.1 The ESML/LOOPN Prototyping System

Step 4 Identify all control transformations, determine the control transformation type (i.e. no input prompt, Activate input prompt, Activate and Pause input prompts), and apply the appropriate template to build the LOOPN net module. The templates for this step are described in section 5.4.4.

The translation templates are graphical in nature, however, for the present the generated LOOPN net specification needs to be specified textually in the LOOPN source language. The LOOPN net modules for the flow and control transformations of steps 3 and 4 can use parameter places (section 4.3.1.4) for connecting to their input and output flows. The LOOPN source program also requires two special modules: one for declaring the flow data types, the other is a driver module which specifies the LOOPN net portions of steps 1 and 2 and connects them with instances of the modules defined by steps 3 and 4. The following paragraphs describe each of the translation templates in turn.

5.4.1 Flows

Figure 5.2 shows the translation templates for discrete data flows, signals, prompts, and continuous data flows.

Discrete flows are modelled as a simple place connected to readers and writers by uni-directional arcs. Writing causes the place to be marked, reading causes any token in the place to be removed. For discrete data flows, the place type indicates the data structure of the flow. A token in the place indicates that data is present on the flow. For signal flows, the place type is unstructured ("*null*"). A token in the place indicates an occurrence of the signal.

Prompts are modelled like discrete data flows. The token contains a special attribute labelled "E", "D", "S", "R", or "T" to indicate the actual prompt. Places which implement composite prompts may carry more than one token at a given time. To ensure that tokens are removed in the order they arrive, each place that implements a prompt is restricted using the special LOOPN "*first()*" function. This ensures that prompts transmitted along a composite prompt are processed in the order they arrive.

Continuous data flows are modelled as places connected to readers and writers by bi-directional arcs. A bi-directional arc corresponds to two uni-directional arcs in opposite directions. These places are always marked. The place type indicates the data structure of the continuous flow. The place type contains an additional "*undefined*" Boolean attribute which determines whether the place contains meaningful data. Writing a value involves changing the token value, and setting the "*undefined*" attribute to false, reading involves taking a copy of the token. A read can only be performed if the "*undefined*" attribute is false. The bi-directional arcs ensure that the place remains marked. The ESML execution rules refer to the placement of "*tokens*" on continuous

flows Here, "tokens" refers to net tokens whose "undefined" attribute is false The "undefined" attribute is of implementation concern only

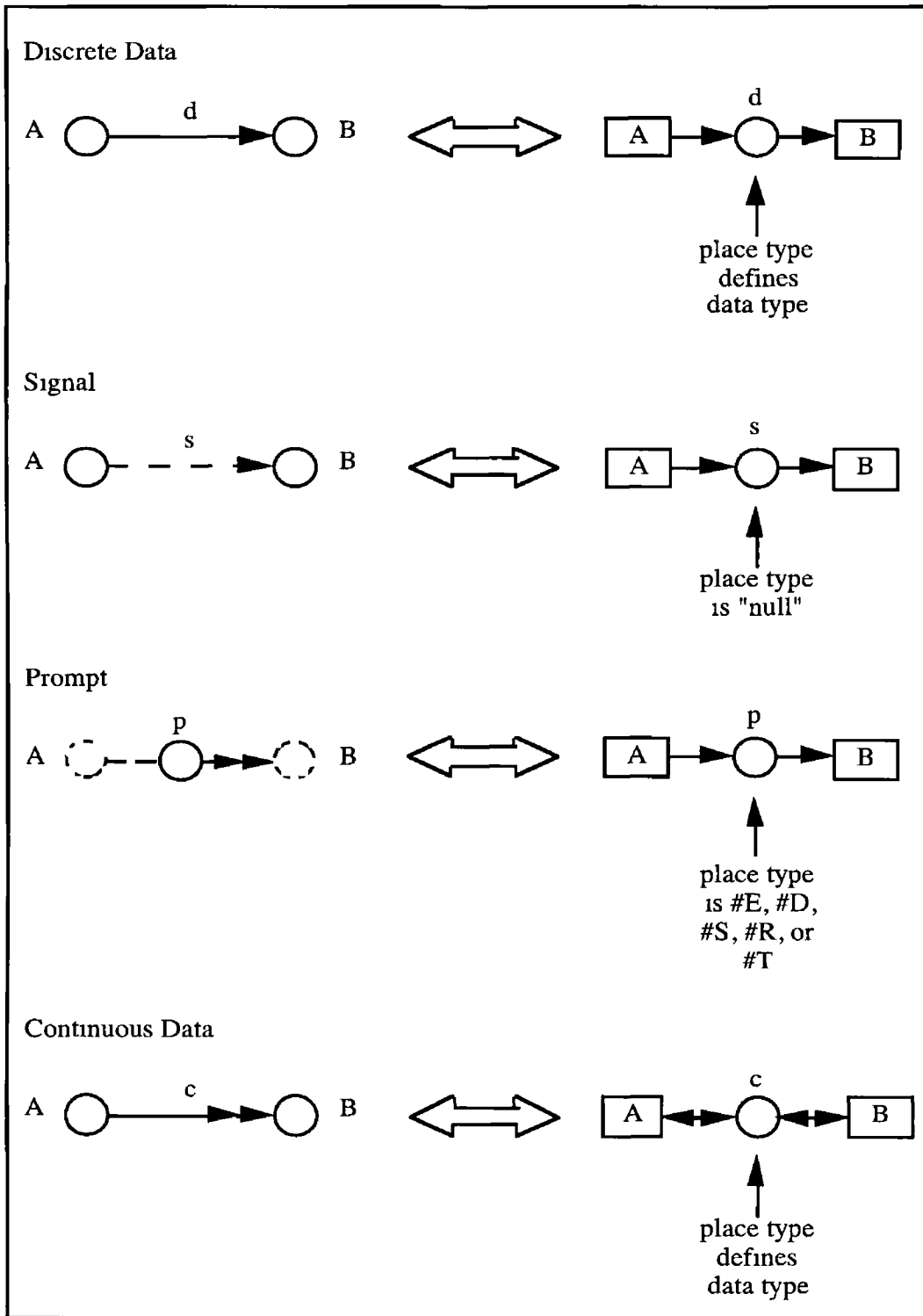


Figure 5.2 Templates for Flows

Flows may converge and diverge to represent multiple sources, multiple destinations, as well as separation and combination of content (as in Figure 3 2) Figures 5 3 and 5 4 show the templates for convergence and divergence of continuous and discrete flows

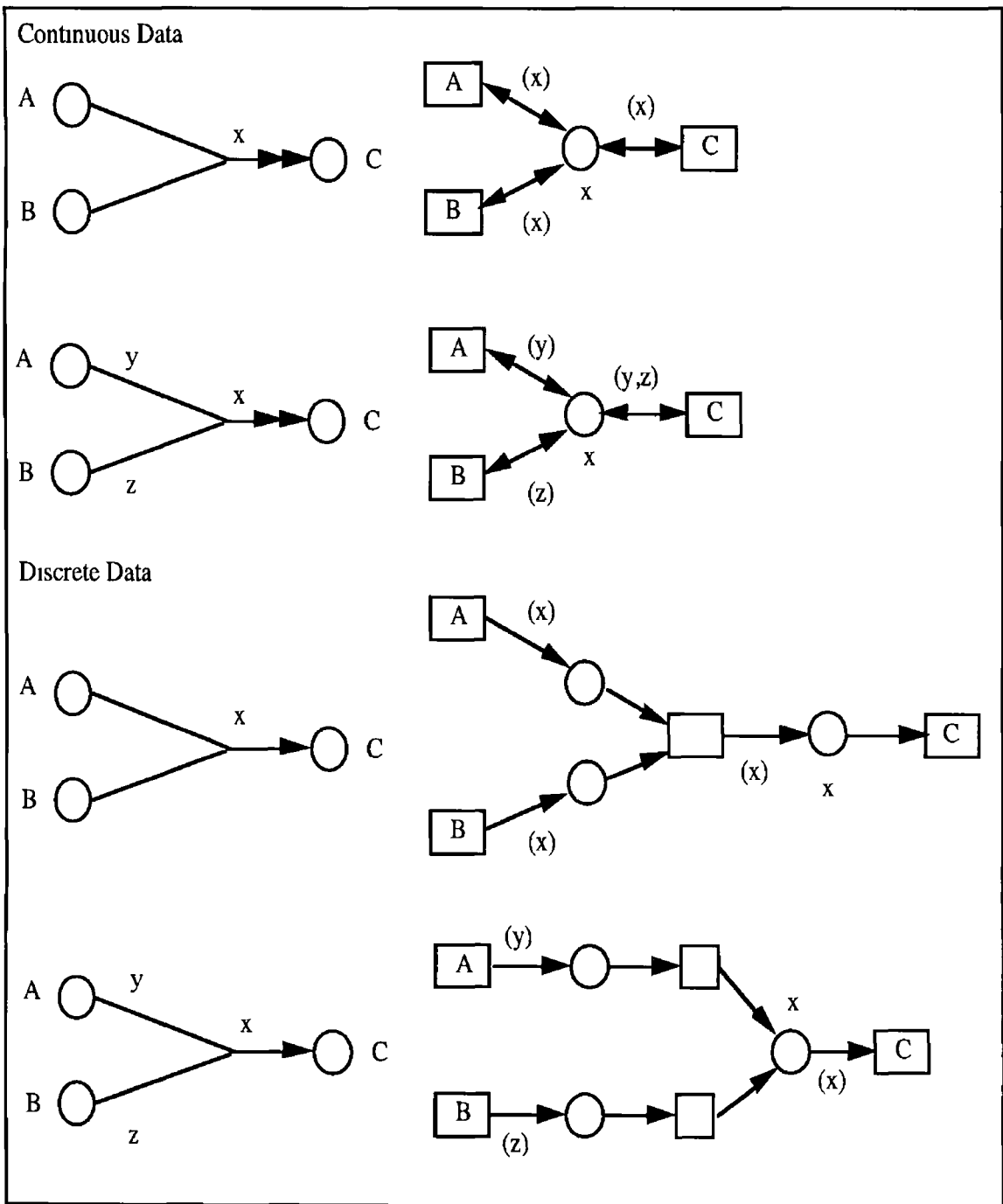


Figure 5 3 Templates for Flow Convergence

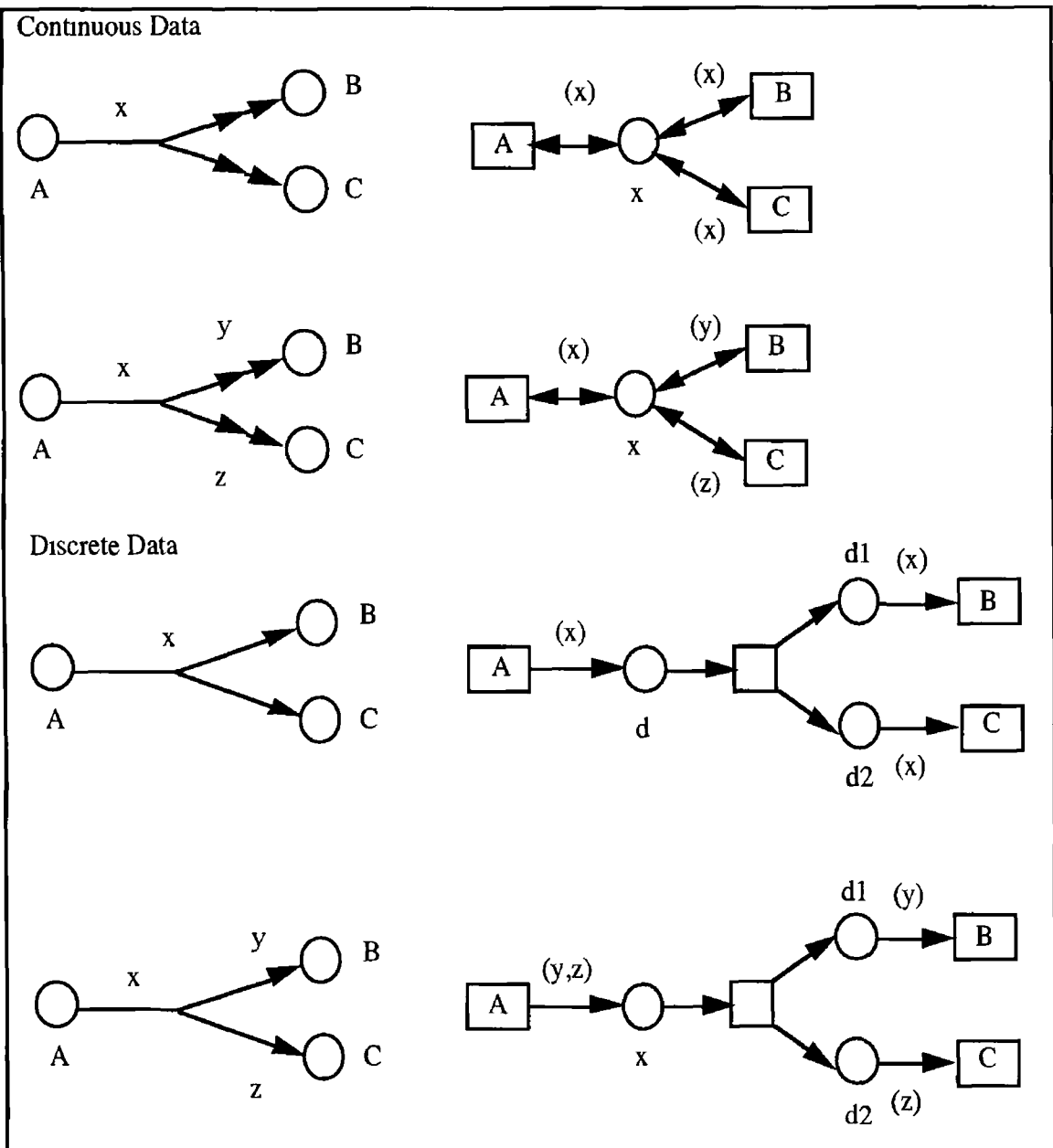


Figure 5.4 Templates for Flow Divergence

Special consideration must be given to flows which are to/from terminators. Flows from terminators are represented as places which are marked in the initial marking. If the flow is a data flow, the initial marking must define values for the attributes of these initial tokens. If the flow is a discrete data flow or signal, the places used to model these flows must be restricted with "*delay(t)*" functions, so that tokens become visible within the places at different times as execution proceeds. This is used to simulate the generation

of discrete inputs and signals by terminators. Flows to terminators are also modelled as places, these places are marked by the net as execution proceeds.

5.4.2 Stores

The following paragraphs describe the translation templates for non-depletable and depletable stores.

5.4.2.1 Non-Depletable Stores

Non-depletable stores are modelled like continuous data flows, as a place, which is always marked, connected by bi-directional arcs to readers and writers. The place type determines the structure of the data stored. This type defines an additional "empty" attribute which is set to true in the initial marking. Reading can only be performed if the "empty" attribute is not true. The execution rules refer to the placement of "tokens" on non-depletable stores, here a non-depletable carries a "token" if its place carries a token whose "empty" attribute is false.

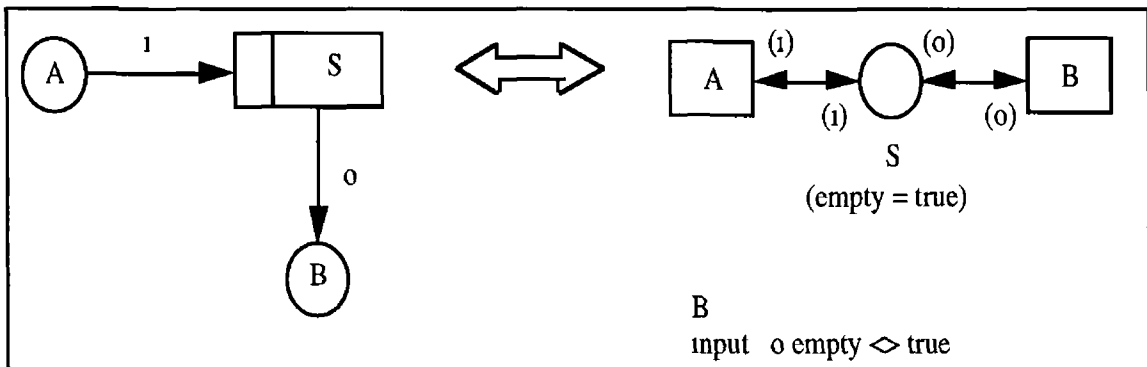


Figure 5.5 Template for Non-Depletable Store

Figure 5.5 shows the template for a non-depletable store connected to a reader and writer. A reader takes a copy of the store value, a writer modifies the store value. A non-depletable store must balance in terms of net inputs and outputs, and can only be attached to transformations by discrete data flows.

5.4.2.2 Depletable Stores

The contents of a depletable store are consumed on being read. Depletable stores are therefore modelled like discrete data flows as a place which may contain multiple tokens. The place type determines the structure of the individual data items stored. Tokens are added and removed as writes and reads are made. Readers and writers are attached by uni-directional arcs.

Figure 5.6 shows the template for a depletable store connected to a reader and writer. A special "Tally" place is used to define the store capacity. It stores a token which has a special "cnt" attribute which tallies the number of tokens in the store place "S". The "cnt" attribute is set to zero in the initial marking. Writers to the store can only do so if the number of tokens will not exceed the store capacity. Readers and writers are responsible for maintaining the "cnt" attribute of the token stored in "Tally".

The way in which readers access the depletable store indicates whether it is ordered as a queue or a stack, this being done using special functions for selection of the first (FIFO) or last token (LIFO) in the place based on when the tokens were added to the place. An infinite capacity depletable store would be implemented as in the finite case, but the "Tally" place would be omitted. A non-depletable store must balance in terms of net inputs and outputs, and can only be attached to transformations by discrete data flows.

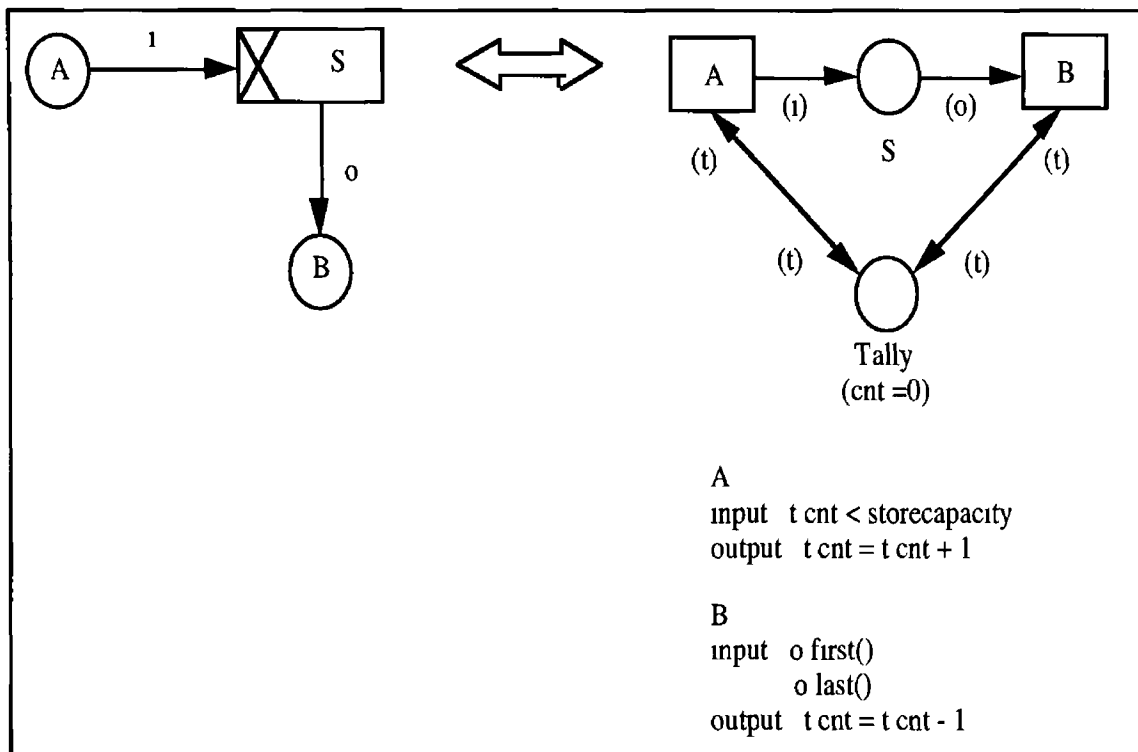


Figure 5.6 Template for Depletable Store

5.4.3 Flow Transformations

Flow transformations convert discrete and continuous inputs into discrete and continuous outputs. If the transformation is primitive, this operation is described using a mini-specification. The mini-specification is implemented using the functions which can

be applied within LOOPN transitions to generate output tokens, which may include a call to an external C function

Since the ESML specification is of a prototype, the translation process must deal with the implementation of non-primitive flow transformations. Transitions which implement non-primitive flow transformations generate output tokens which are copies of input tokens, or which contain dummy values.

The format of the translation templates for a flow transformation depends on whether the input and outputs are continuous, discrete, or both. Discrete outputs can be delayed. A flow transformation can have no input prompt, an Activate input prompt, or a Trigger input prompt.

5.4.3.1 Flow Transformation with no Input Prompt

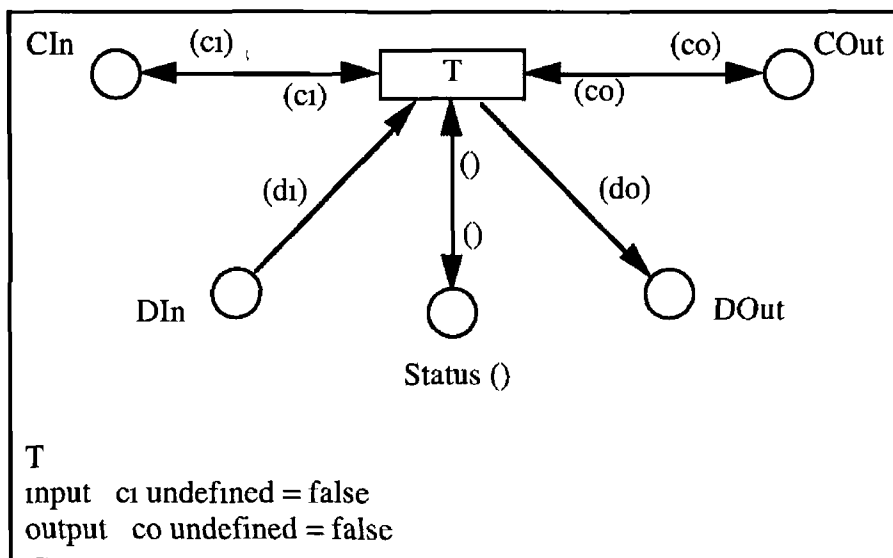


Figure 5.7 Template for Flow Transformation with no Input Prompt, Mixed Input and Output

Figure 5.7 shows the template for a flow transformation which has no input prompts, but has mixed discrete/continuous input and output. The transition "T" must check that the continuous input "CIn" is defined. This is done in the input token selection section of "T". The "Status" place visualises the interaction between the transition and input flows. The execution rules refer to flow transformations carrying tokens. A flow transformation carries a token if the "Status" place carries a token. If the transformation is primitive then the output token generation section of the "T" specifies expressions for output tokens, this must also set the "undefined" attribute for any continuous outputs to

false regardless of whether the transformation is primitive or not Continuous inputs are accessed via bi-directional arcs Discrete inputs are accessed by uni-directional arcs

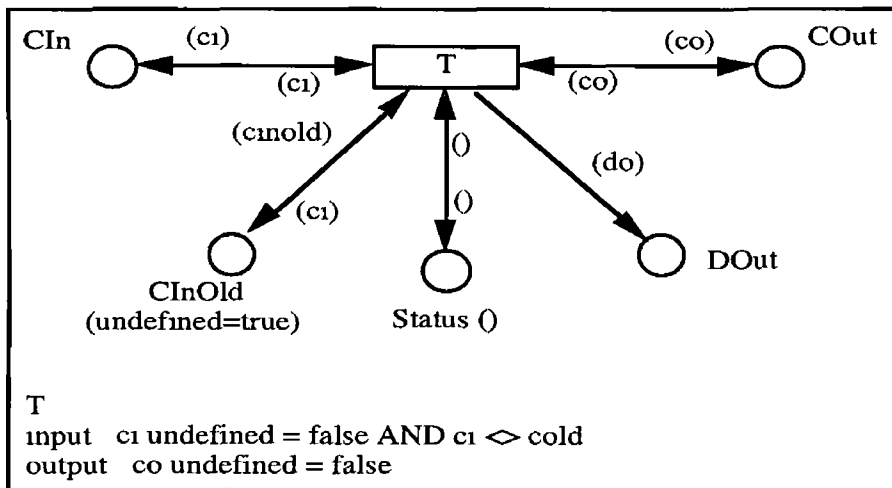


Figure 5.8 Template for Flow Transformation with no Input Prompt, Continuous Input, Mixed Output

Figure 5.8 shows the template for a flow transformation which has no input prompts, but has continuous input and mixed discrete/continuous output. This template is similar to the one in Figure 5.7, but the transition "T" only fires when the continuous input has changed. This is achieved by storing the last value of the continuous input in a new place, "CInOld", and comparing it with the current input to determine whether "T" should fire. The "CInOld" place is of the same place type as the actual input, "CIn", and is marked initially with "undefined" set to true.

Figure 5.9 shows the template for a flow transformation which has no input prompts, but has mixed input and output, where the discrete output is delayed. The delay period is determined from the mini-specification. It only makes sense to delay discrete outputs. The "T" transition fires when discrete input and defined continuous input are available. Continuous output, "COut", is produced immediately. The discrete output is placed in the place "Delay", and place "Delayed" is marked to indicate that the discrete output is currently being delayed. The "Delay" place has attached a place restriction which uses the LOOPN "delay(t)" function to delay any tokens which arrive in it by t time units, where t is the delay period. The "Status" place is unmarked during the delay period. The "Eat" transition removes any discrete input that arrives during the delay period. When the delay period has elapsed, the discrete output becomes visible, the "Out" transition fires making the discrete output available in "DOut", "Status" is marked and "Delayed" is unmarked.

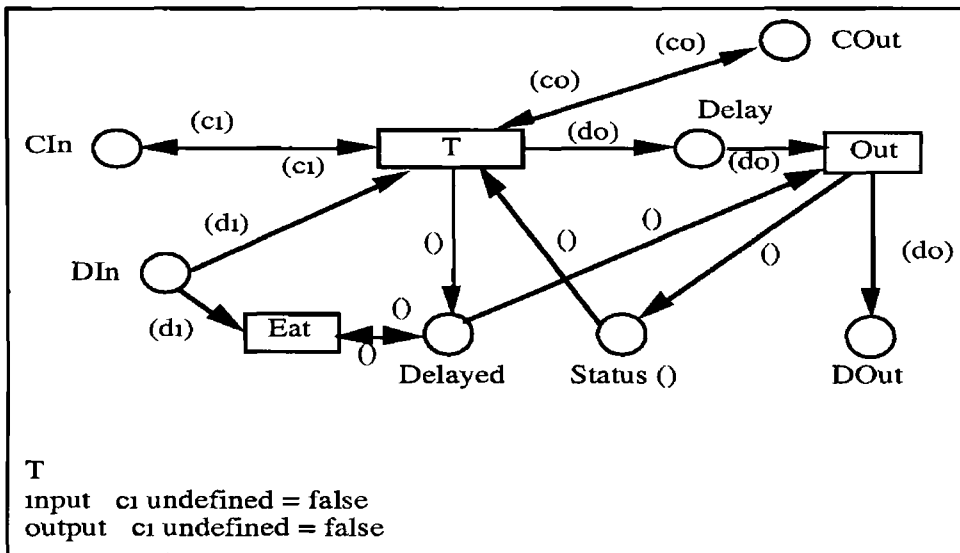


Figure 5.9 Template for Flow Transformation with no Input Prompt, Mixed Input, Continuous Output, Delayed Discrete Output

5.4.3.2 Flow Transformation with Trigger Input Prompt

Figure 5.10 shows the template for a flow transformation with Trigger input prompt. Such a flow transformation may only have continuous input flows and discrete outputs. The place which stores the Trigger prompt allows the transition to fire when it has a token. As before, the input token selection section of "T" must check the "undefined" attributes of any continuous inputs.

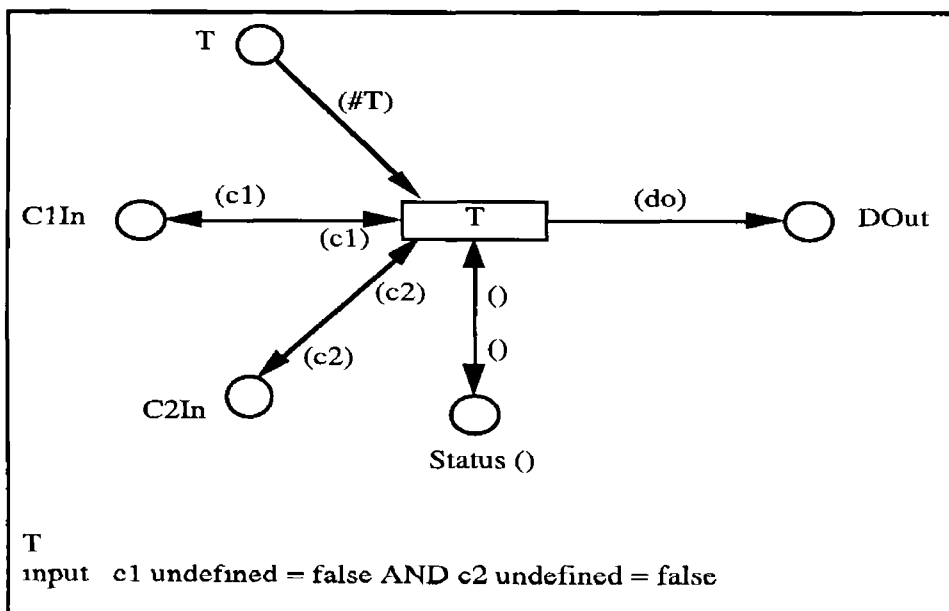


Figure 5.10 Template for Flow Transformation with Trigger Input prompt

5.4.3.3 Flow Transformation with an Activate (E/D) Input Prompt

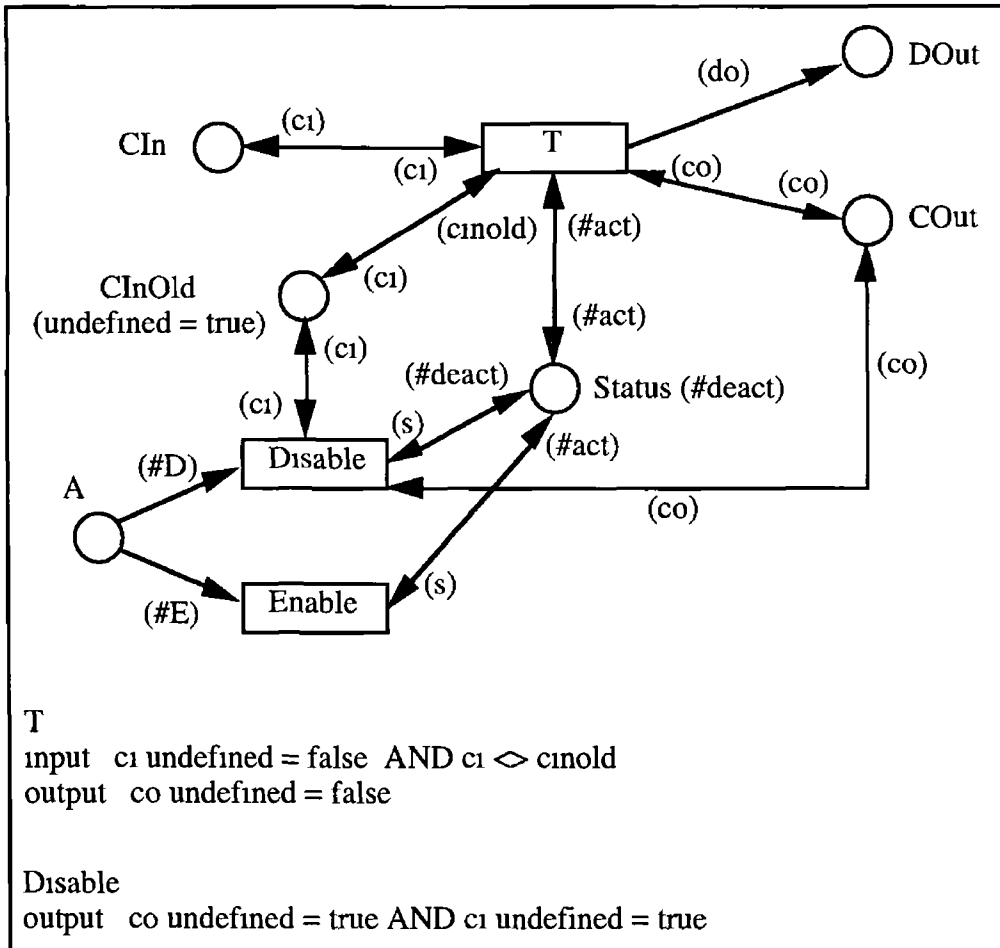


Figure 5 11 Template for Flow Transformation with Activate Input Prompt

Figure 5 11 shows the template for a flow transformation with Activate input prompt The "Status" place indicates whether the transformation is activated ("*act*") or deactivated ("*deact*"), as a result of receipt of an Enable or Disable prompt The transformation is initially deactivated The transition "T" can only fire if the transformation is active

If there are only continuous inputs (as in the template), then to prevent the continual firing of the transition, the "CInOld" place is used as before Any discrete inputs which arrive while the transformation is deactivated must be removed without effect

When the transformation is deactivated, the "undefined" attribute of all continuous outputs is set to true, this indicates that the transformation is not controlling the value of

its continuous outputs This is achieved by the "Disable" transition, which also undefines any value stored in "CInOld" As before, if the transformation is primitive, the transition output token generation section implements the mini-specification

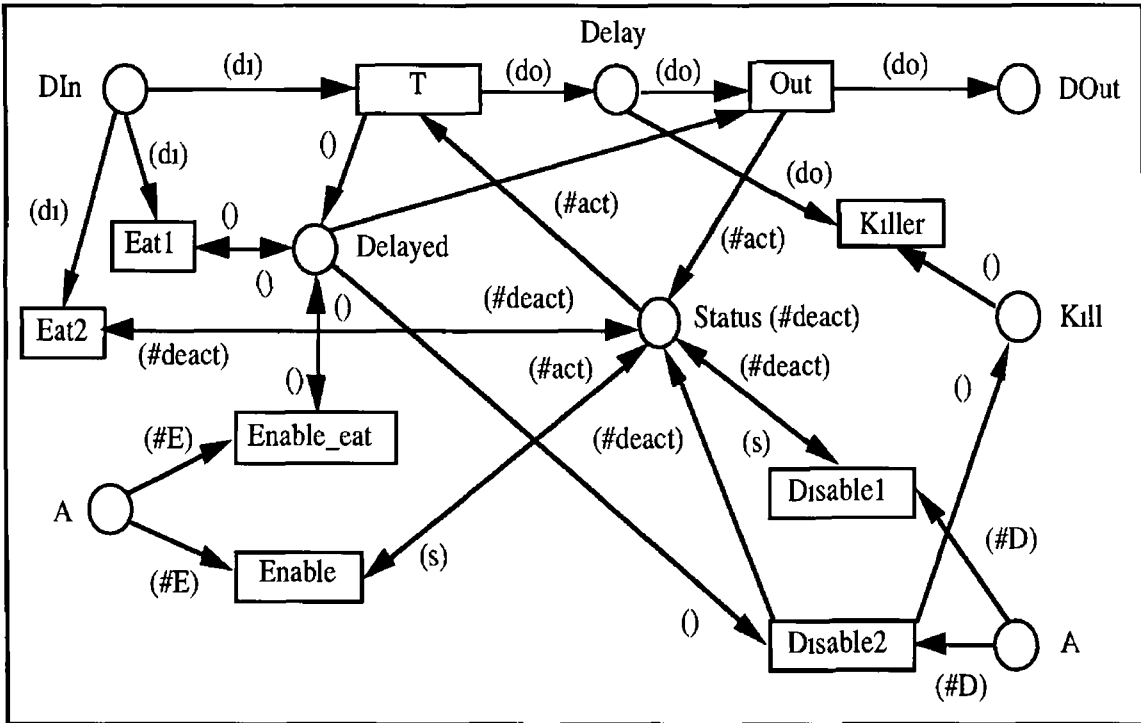


Figure 5.12 Template for Flow Transformation with Activate Input Prompt, Discrete Input, Delayed Discrete Output

Figure 5.12 shows the template for a flow transformation with an Activate input prompt which delays its discrete output The template is a combination of Figures 5.9 and 5.11 The "Eat1" transition eats any discrete input that arrives during the delay period "Eat2" eats any discrete input that arrives while the transformation is deactivated The "Enable_eat" transition eats any Enable prompts that arrive during the delay period The "Disable2" transition is responsible for deactivating the transformation during the delay period, it marks the "Kill" place which enables the "Killer" transition which discards the discrete output before it is made available to the environment

5.4.4 Control Transformations

Control transformations implement the control logic of the ESML model and are described using STDs, the graphical representation of FSMs A control transformation can have no input prompt, an Activate input prompt, or matching Activate and Pause

input prompts It is invalid for a control transformation to have a lone Pause input prompt, there must be a matching Activate input

5.4.4.1 Control Transformation with no Input Prompt

Figure 5 13 shows a translation template for a control transformation with no input prompt The LOOPN net template implements the control structure of a generic control transformation with no input prompt

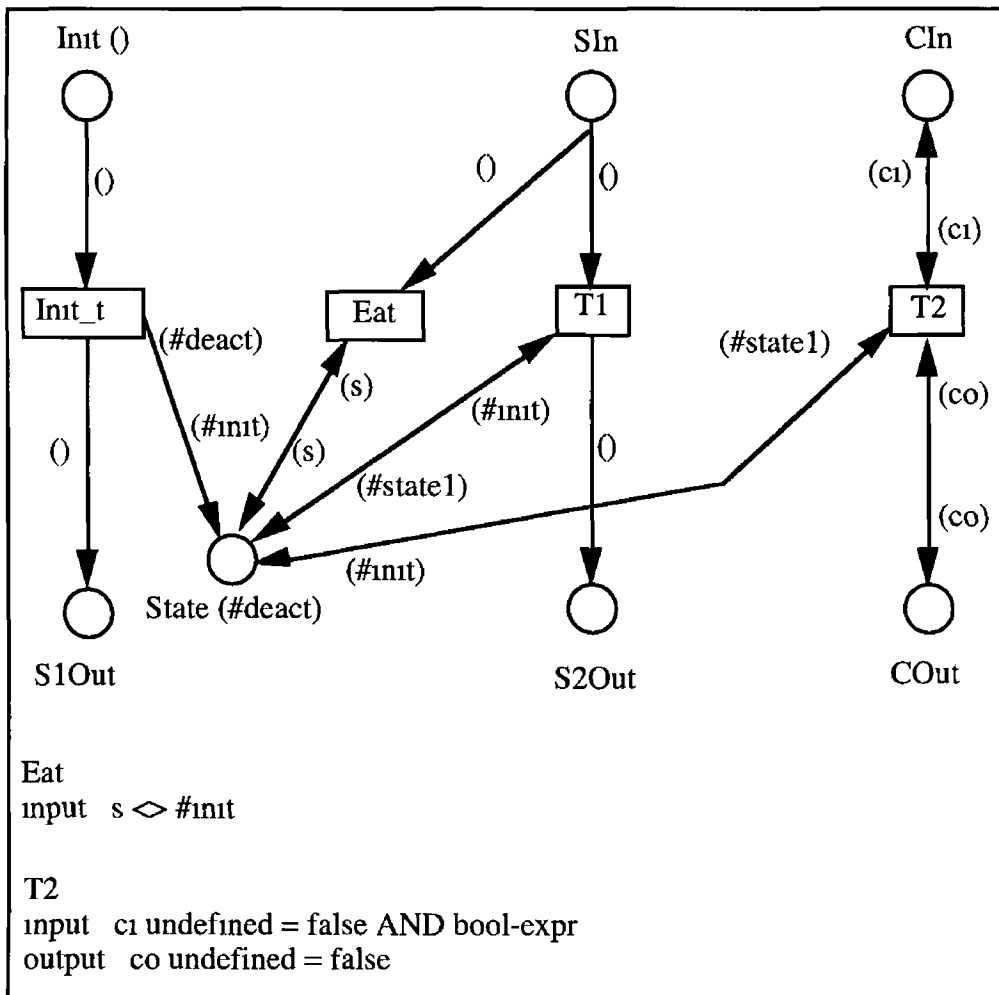


Figure 5 13 Template for Control Transformation with no Input Prompt

The current state of the FSM is stored as an attribute within the "State" place The "State" place is initially marked as "#deact" (deactivated) The "State" place is the key to the operation of the template Its content changes as transitions fire The execution rules refer to control transformations carrying tokens A control transformation carries a token

if the *"Status"* place carries a token. The token in the *"Status"* place is associated with the current state of the FSM.

The *"Init"* place is initially marked and is used to fire the *"Init_t"* transition, which corresponds to the initial transition of the state machine. The initial transition sets up the initial state and optionally produces output. In the template the *"Init_t"* transition sets up the initial machine state (*#init*), and generates an output signal *"S1Out"*. State transitions fire on receipt on input signals, or when a continuous input fulfils a Boolean expression. The firing of the transition causes a state change and the optional generation of output. The output may include the sending of prompts to other transformations. The arrival of unexpected signals does not effect FSM operation, and are removed by special *"Eat"* transitions.

The template shows a STD with two state changing transitions, *"T1"* and *"T2"*. *"T1"* fires if the FSM is in the initial state (*#init*) and signal *"SIn"* arrives, and causes the FSM to change to a new state (*#state1*), and produce an output signal *"S2Out"*. The *"Eat"* transition removes signal *"SIn"* if it arrives during any state other than the initial state. *"T2"* fires if the state is *"#state1"*, and if the continuous input *"CIn"* satisfies a Boolean expression, and causes the state machine to change to the initial state again, and set the value of the continuous output *"COut"*. Input continuous flows can only be read if their *"undefined"* attributes are false. Boolean conditions on input condition flows are tested in the input token selection section of the LOOPN transition. On writing continuous outputs, the *"undefined"* attribute is set to false.

5.4.4.2 Control Transformation with an Activate (E/D) Input Prompt

Figure 5.14 shows the template for a control transformation with an Activate input prompt. The template is an extension of Figure 5.13 to cope with the prompt.

The FSM is initially deactivated (*"#deact"*). On receipt of an Enable prompt along the Activate flow, the *"Enable"* transition fires and places the FSM in the initial state (*"#init"*). The template does not show any initial actions. Any such actions would be hooked up as output of the *"Enable"* transition. The *"Eat1"* transition removes spurious Enable prompts that may arrive while the state machine is enabled. The *"Disable"* transition fires on receipt of a Disable prompt along the Activate flow. It deactivates the state machine, which cannot respond to inputs until it becomes enabled again. The execution rules require that a control transformation, on receipt of a Disable prompt, should propagate on the Disable prompt to any further transformations that it activates. This would be achieved by producing the Disable prompts as outputs of the *"Disable"* transition.

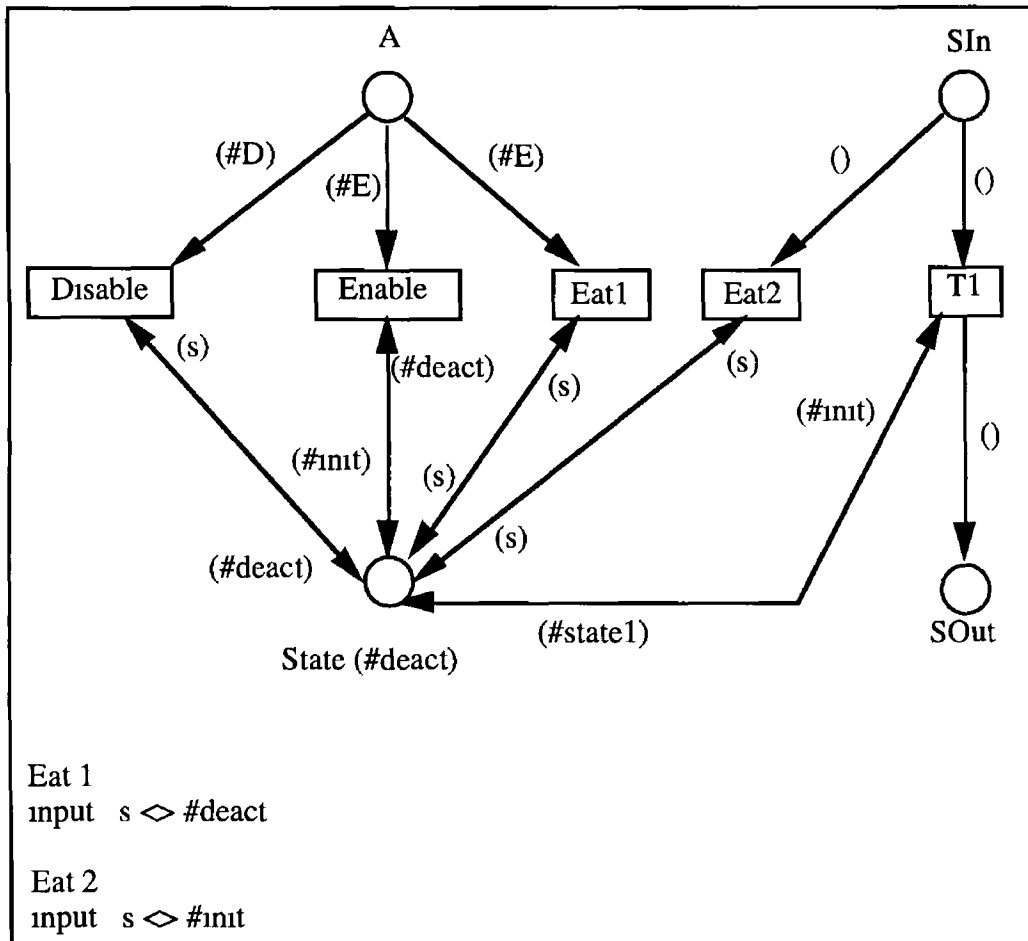


Figure 5 14 Template Control Transformation with Activate Input Prompt

The template shows one simple state changing transition "T1", which causes a state change on receipt of an "SIn" signal, and produces the output signal "SOut". The signal "SIn" is eaten if it arrives unexpectedly. If a state changing transition produces continuous output, the "undefined" attribute of this output must be set to true when the FSM is deactivated. This is to indicate that the FSM is not controlling the values of its continuous outputs, and is achieved by the "Disable" transition.

5.4.4.3 Control Transformations with Activate (E/D) and Pause (S/R) Input Prompts

Figure 5 15 shows the template for a control transformation with matching Activate and Pause input prompts. The template is an extension of the Figure 5 14 to cope with the Suspend and Resume prompts.

The FSM is initially deactivated ("#deact"). The "Enable" and "Enable_eat" activate the FSM and remove spurious Enable prompts. If the FSM is not suspended, the "Disable1" transition fires on receipt of a Disable prompt.

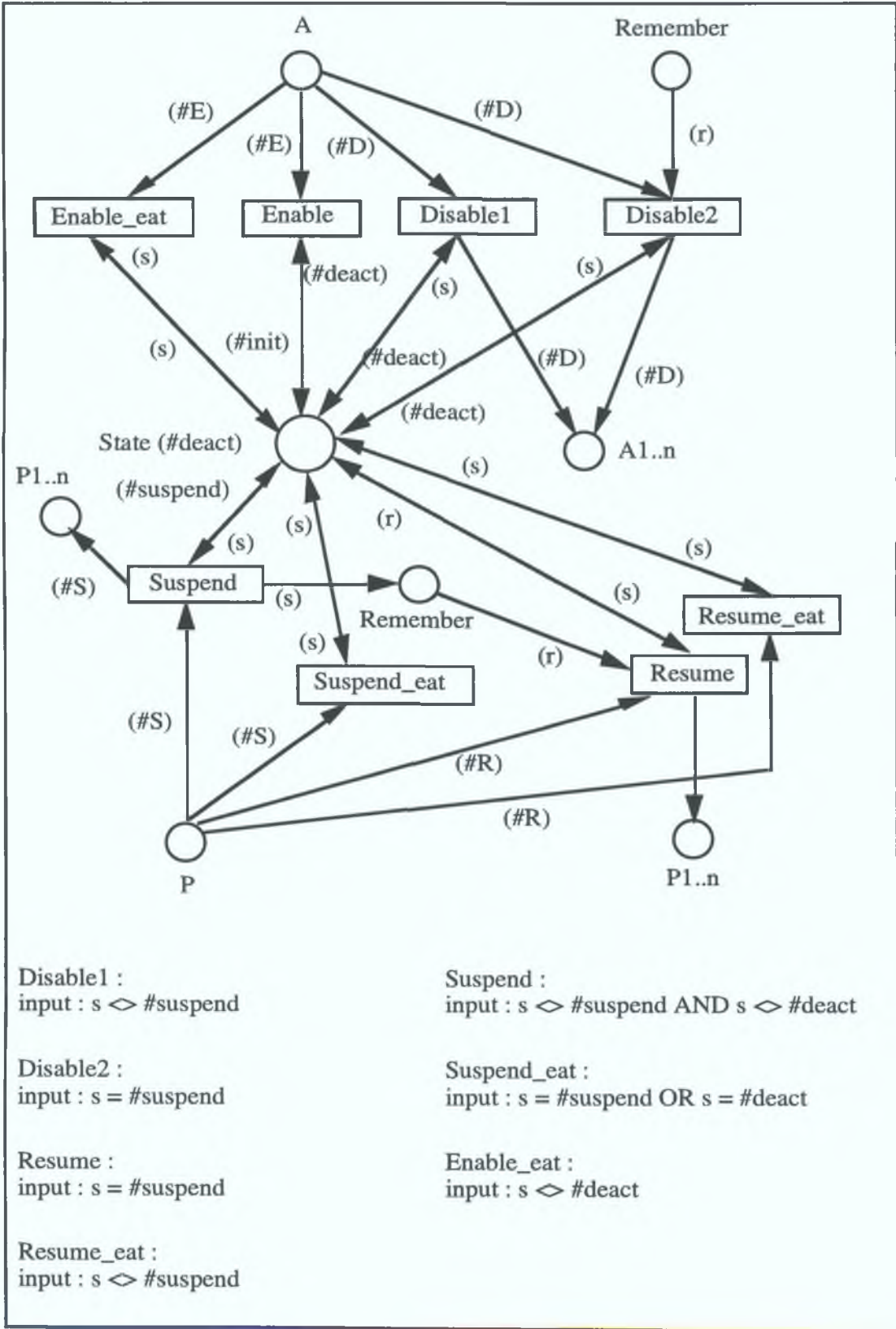


Figure 5.15 Template for Control Transformation with matching Activate and Pause Input Prompts

The "*Suspend_eat*" transition removes Suspend prompts from input if the FSM is already suspended or deactivated. On receipt of a Suspend prompt, if the FSM is not suspended or deactivated, the "*Suspend*" transition fires, saves the current state in the "*Remember*" place, updates the state to "*#suspend*", and propagates the Suspend prompt onto any transformations that are paused by this transformation, but also activated by this transformation, as defined in the execution rules. The place labelled "*PI n*" is used to represent these transformations.

The "*Disable2*" transition is used to disable the transformation when it is suspended. When it fires, it empties the "*Remember*" place, and propagates the Disable prompt onto any transformations activated by this transformation. These transformations are represented by the place labelled "*AI n*". The "*Disable1*" transition also propagates the Disable prompt onto these transformations. Note that the template shows several places of the same name, these are in fact the same place, they have been duplicated only as a drawing convenience. The template also omits any state changing transitions for simplicity.

The "*Resume*" transition is responsible for resuming a suspended transition. It fires if the FSM receives a Resume prompt while suspended. The "*Resume*" transition restores the transformation state to that stored in the "*Remember*" place, and propagates the Resume prompt onto any transformations paused by this transformation that are also activated by this transformation ("*PI n*"). The "*Resume_eat*" transition removes Resume prompts that arrive while the FSM is not suspended.

5.5 Summary

This chapter has defined the ESML prototyping system and has presented the translation templates used to translate non-executable ESML specifications into executable LOOPN specifications, so defining an execution semantics for ESML according to the ESML execution rules. The prototyping system facilitates the use of ESML as a graphical executable specification language for constructing prototypes for use within a keep-it prototyping approach. The translation templates model the ESML language constructs using LOOPN net fragments. The translation templates concern the various flow types, stores, flow transformations and control transformations. Stores are further refined by type (non-depletable/depletable). Flow transformations are further refined by the type of input prompt (i.e. no input prompt, Trigger input prompt, Activate input prompt). Control transformations are refined similarly by input prompt type (i.e. no input prompt, Activate input prompt, matching Activate and Pause input prompt). The following chapter applies the prototyping system to the case study, the APU fuel subsystem.

Chapter 6

The APU Fuel Subsystem Case Study

6.1 Introduction

The previous chapter has defined the ESML/LOOPN prototyping system. This chapter describes the chosen case study, i.e. the APU (Auxiliary Power Unit) Fuel Subsystem, and uses the ESML/LOOPN prototyping system to generate an exploratory prototype of it. The chapter starts with a general description of the APU, followed by a detailed description of the Fuel Subsystem. The chapter then describes the specification of the prototype of the case study in ESML, the Ward/Mellor RTSA/SD method having guided the specification effort. The chapter then describes an equivalent specification of the prototype in STATEMATE (using the Activity-chart and Statecharts languages). This results in a comparison of ESML with the languages of STATEMATE. The chapter then describes the application of the translation templates and shows the executable LOOPN specification which is produced. The chapter closes with a description of the prototype execution output.

6.2 The Auxiliary Power Unit (APU)

The case study is a subsystem of a typical real-time system, *the Auxiliary Power Unit (APU)*, an avionic system used on the Boeing-737 airplane series. The APU is an avionic system used for the production of electrical power for the airplane electrical system, and for the production of compressed air for engine starting and air conditioning. The bulk of the APU itself is located under the main tail section. It interfaces with components located in the flight compartment, electronic equipment compartment, main wheel well, and aft cargo compartment. The APU consists of several subsystems, the APU engine, a start subsystem, an air intake subsystem, an accessory cooling and bleed subsystem, an oil subsystem, and a fuel subsystem.

The *APU engine* is a gas-turbine engine consisting of a two-stage centrifugal compressor directly coupled to a single-stage radial inflow turbine. The turbine shaft is geared to the accessory drive section and provides power for driving the engine accessories and the generator. The generator supplies power to the airplane electrical system.

The *Start Subsystem (SS)* consists of a starter motor, which is responsible for rotating the APU engine to starting speed, and an igniter and spark plug, which are

responsible for igniting the fuel/air mix in the combustion chamber. A special 90 second timeout operates to cut power from the starter and ignition circuits in case of malfunction.

The *Air Intake Subsystem* (AIS) consists of an inlet door for air inflow from the atmosphere. The incoming airflow is split into two flows, one for the engine compressor, the other for accessory cooling. Used cooling air is exhausted overboard through a hole in the titanium shroud which encloses the APU. This shroud provides a fireproof sound reducing enclosure.

The *Accessory Cooling and Bleed Subsystem* (ACBS) fans the accessories with cool air and regulates the bleeding of air from the engine. It consists of pneumatic, mechanical and electronic components, which function automatically to regulate the rate and maximum amount of bleed air that can be drawn from the APU, so as not to exceed the APU operational limits. Bleed air is supplied from the APU engine to the airplane pneumatic system through a bleed air valve. A fan driven by the accessory drive circulates cooling air to the electric generator, lubricating oil cooler and engine accessories. A surge bleed valve prevents compressor surge during APU operation in-flight.

The *Oil Subsystem* (OS) lubricates the APU engine bearings and gearbox with a supply of pressurised oil from a pump. The oil is then scavenged and passed through a cooler and filter and returned to the oil tank.

6.2.1 The APU Fuel Subsystem

The *APU Fuel Subsystem* delivers fuel from the airplane fuel tank no. 1 to the combustion chamber of the APU engine. The subsystem consists of a fuel control valve, fuel solenoid valve, pump, filter, heater, fuel control unit, and bypass valves. Figure 6.1 presents a block diagram description of the subsystem.

The boost pump and control valve are located in the left main wheel well. The remaining components are located in the APU engine and in the upper shroud. The APU Fuel Subsystem interfaces with the P5 overhead cockpit panel, the airplane fuel tank no. 1, the AIS, the engine speed detection switch, the ACBS, the APU engine, the atmosphere, and the OS.

The *APU fuel control valve*, also known as the *APU fuel shutoff valve*, allows fuel supply from fuel tank no. 1 to the APU engine. When the APU switch, located on cockpit panel P5, is placed in the On position, the valve opens allowing fuel to flow. The valve signals the AIS to open the air inlet door so as to allow inlet airflow for engine turbine operation. When the APU switch is placed in the Off position, the valve closes and signals the inlet door to close.

The *fuel solenoid valve* (FSV) allows fuel flow to the engine combustion after initial rotation of the APU. The valve is controlled by the sequencing oil pressure switch and the APU switch. The valve opens when the sequencing oil pressure switch closes as a result of adequate oil pressure, activating the fuel control unit. The valve closes when the APU switch is placed in the Off position.

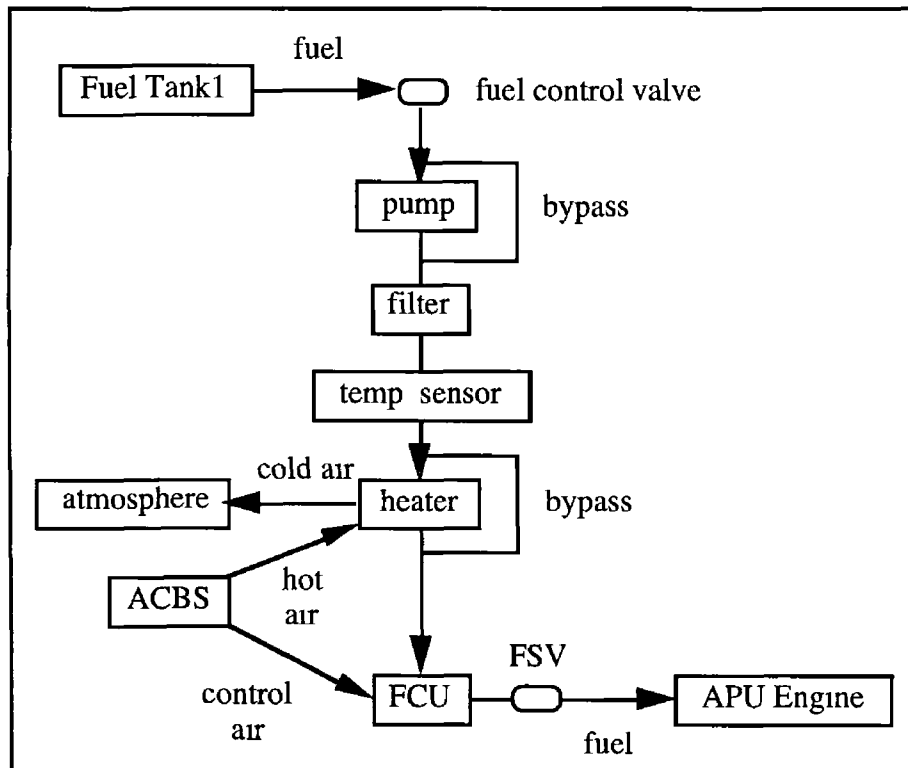


Figure 6-1 APU Fuel Subsystem Block Diagram

The *boost pump* is used during engine starting for providing pressurised fuel. It is controlled by the fuel control valve and fuel tank no. 1 switches. If the APU switch is in the On position and the fuel tank no. 1 switch is in the Off position, the boost pump is activated. When the speed switch detects 95% engine rpm during the start cycle, the boost pump is deactivated, and the pump bypass operates. Fuel flow from the pump and bypass is passed through a cleaning filter.

The *fuel heater* is used to prevent the fuel becoming frozen. It is controlled by the fuel control valve and a temperature sensor. When the fuel temperature drops to 37° F, the heater is activated. It uses hot bleed air from the accessory cooling and bleed system to heat the fuel, the resultant air is exhausted into the atmosphere. When fuel temperature increases to 64° F, the heater is deactivated. A bypass operates when the heater is deactivated.

The *fuel control unit* (FCU) is used to regulate fuel flow to the engine combustion chamber during starting and normal operation. The regulated fuel flow controls the acceleration of the engine during the starting operation. The FCU is controlled by the fuel solenoid valve. The FCU regulates the fuel flow to maintain a near constant speed and a safe operating temperature of the unit.

Airplane maintenance personnel required a software simulation of the APU Fuel Subsystem which could be used for training and fault diagnosis. The problem was that the exact requirements for the proposed system were not known and that developers were unsure on how to simulate physical system components. The ESML/LOOPN prototyping system has been used to construct an exploratory prototype of the case study. The prototype is to be evolved into the production system as part of a keep-it prototyping approach.

6.3 Prototype Specification in ESML

The ESML/LOOPN prototyping system first requires that the prototype be specified manually in ESML. The Ward/Mellor RTSA/SD guides the specification effort. A Behavioural Model specification of the case study has been built using TurboCASE. The ESML specification, depicted in Figure 6.2, has been constructed by identifying the objects in the problem domain, and partitioning the model on that basis. A context schema and event list have been used as the basis for constructing the Behavioural Model. The specification is that of a *diagonal prototype*, all functions have been included, but they have been described in varying amounts of detail.

The Behavioural Model consists of four control transformations, six flow transformations, and eight terminators. The mini-specifications of primitive transformations are simple, not requiring external specification as C functions.

All data flows within the model are continuous, reflecting the physical nature of the system. Fuel is modelled as a composite continuous data flow consisting of pressure, temperature and filtered components. The pressure and temperature components are floating point values, the filtered component is a Boolean data item. Air is a similar composite flow, but omits the filtered component.

The fuel control valve controls the boost pump and heat controllers, which in turn activate the pump, heater and bypasses. The STD for the fuel control valve is shown in Figure 6.3. The valve opens on receipt of an APU_On signal from the cockpit Panel_P5. On opening, the valve signals the AIS to open the air inlet door, and enables the boost pump and heat control transformations. The valve closes on receipt of an APU_Off

signal, and signals the air inlet door to close and disables the boost pump and heat control transformations

The STD for the boost pump control, which provides pressurised fuel during engine starting, is shown in Figure 6 4 It disables the fuel pump and enables the pump bypass on its initial transition, which happens once the transformation is enabled On receipt of a Tank1_Off signal from Panel_P5, the pump is enabled and bypass disabled Once the engine reaches 95% rpm, which is sensed by the speed switch, the pump is disabled and bypass enabled again

The STD for the heat control is shown in Figure 6 5 The heat control disables the heater and enables the heater bypass initially The fuel temperature is an input to the heater control transformation which determines when the heater should operate If the temperature falls below 37° F, the heater is enabled and the bypass is disabled If the temperature rises above 64° F, the heater is disabled and the bypass is enabled

The FSV works independently of the other control transformations, its STD is shown in Figure 6 6 When the sequencing oil pressure switch closes, the fuel control unit is enabled When the APU is switched off, the unit is disabled

The pump transformation is a primitive flow transformation which increases the pressure component of the input fuel composite flow The pump bypass simply copies its input to output The filter transformation is a primitive flow transformation which sets the filtered component of the continuous fuel to true The heat transformation is a non-primitive flow transformation which uses hot bleed air to heat incoming fuel, and expels the resultant air to the atmosphere The heat bypass copies its input to output The FCU is a non-primitive data transformation which regulates fuel flow to the engine based on a control air input

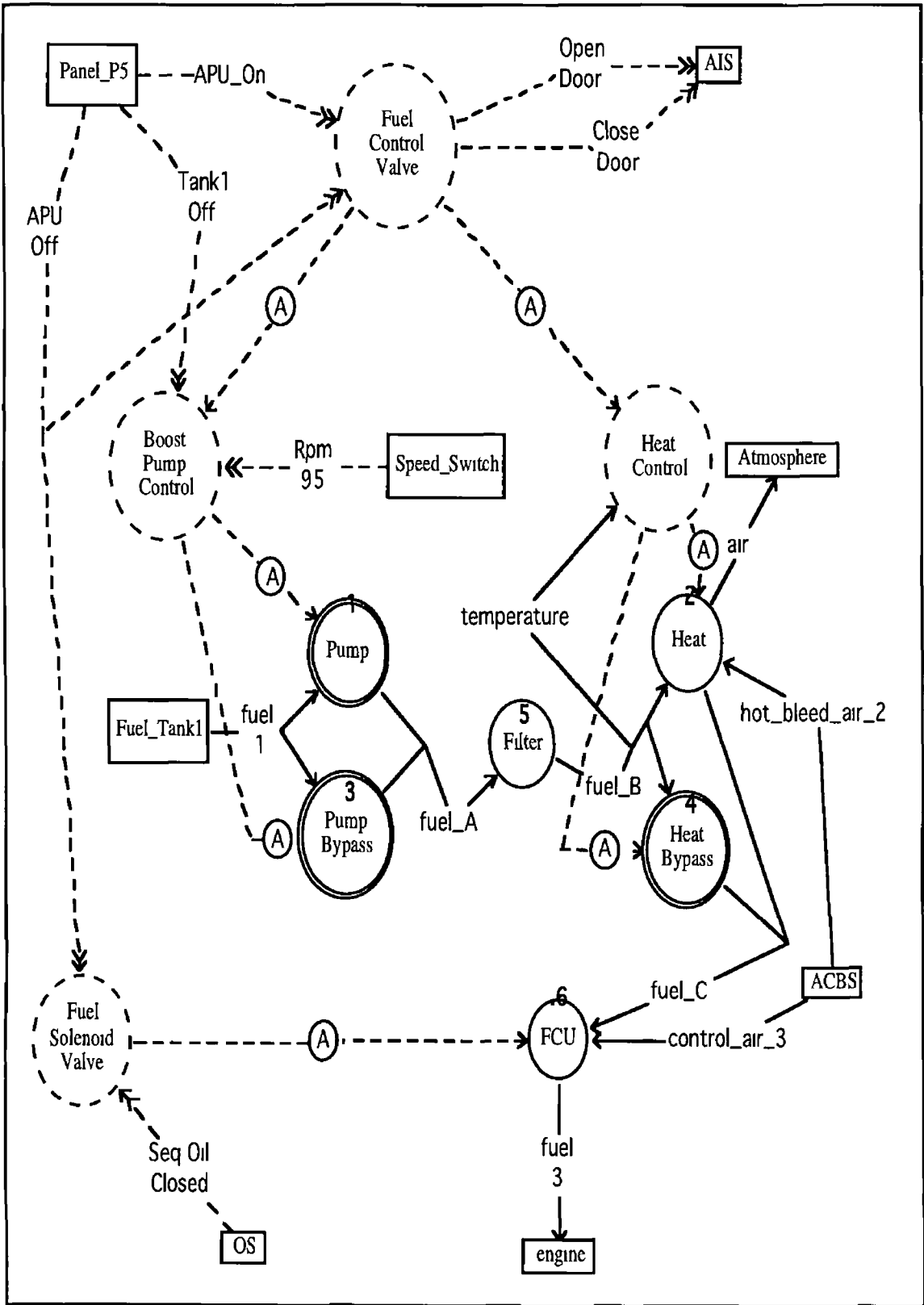


Figure 6 2 APU Fuel Subsystem Behavioural Model

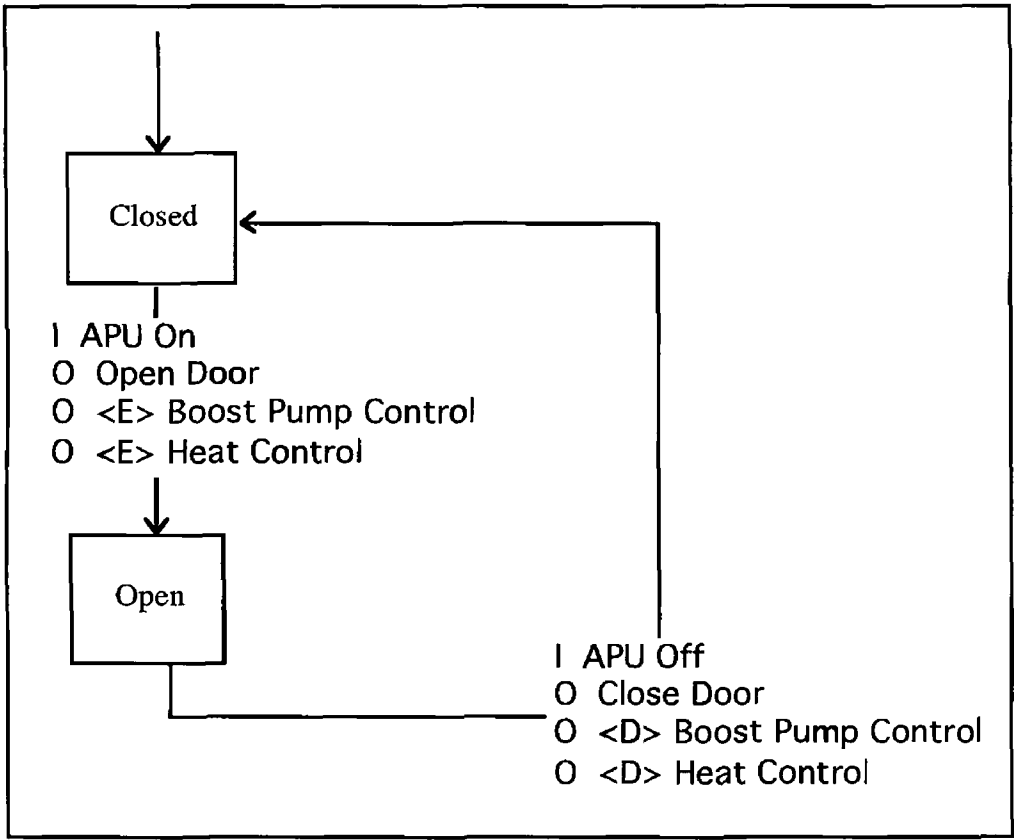


Figure 6 3 STD for Fuel Control Valve

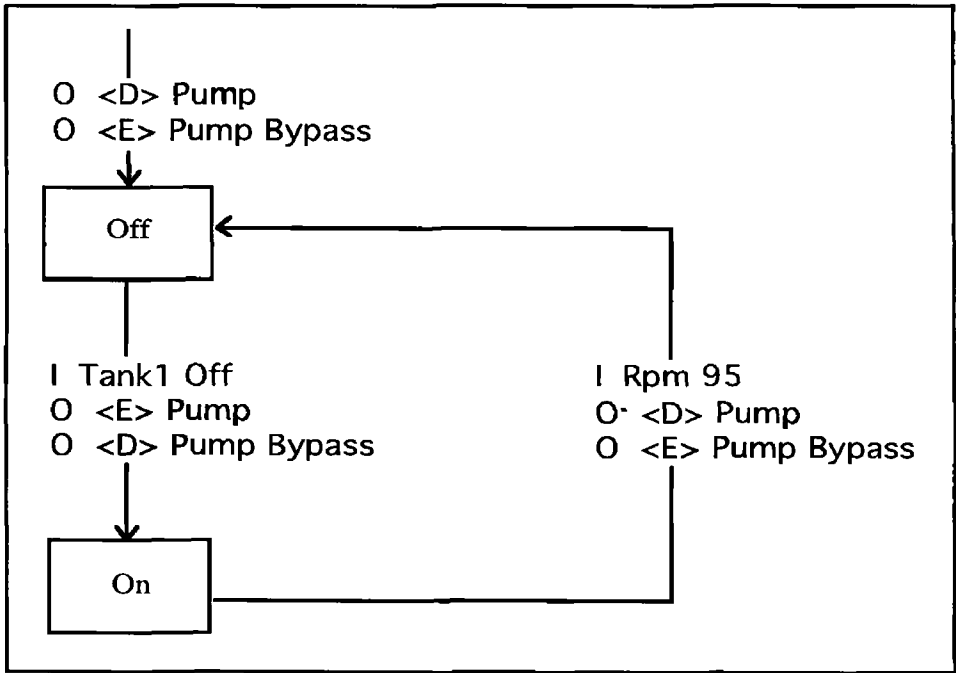


Figure 6 4 STD for Boost Pump Control

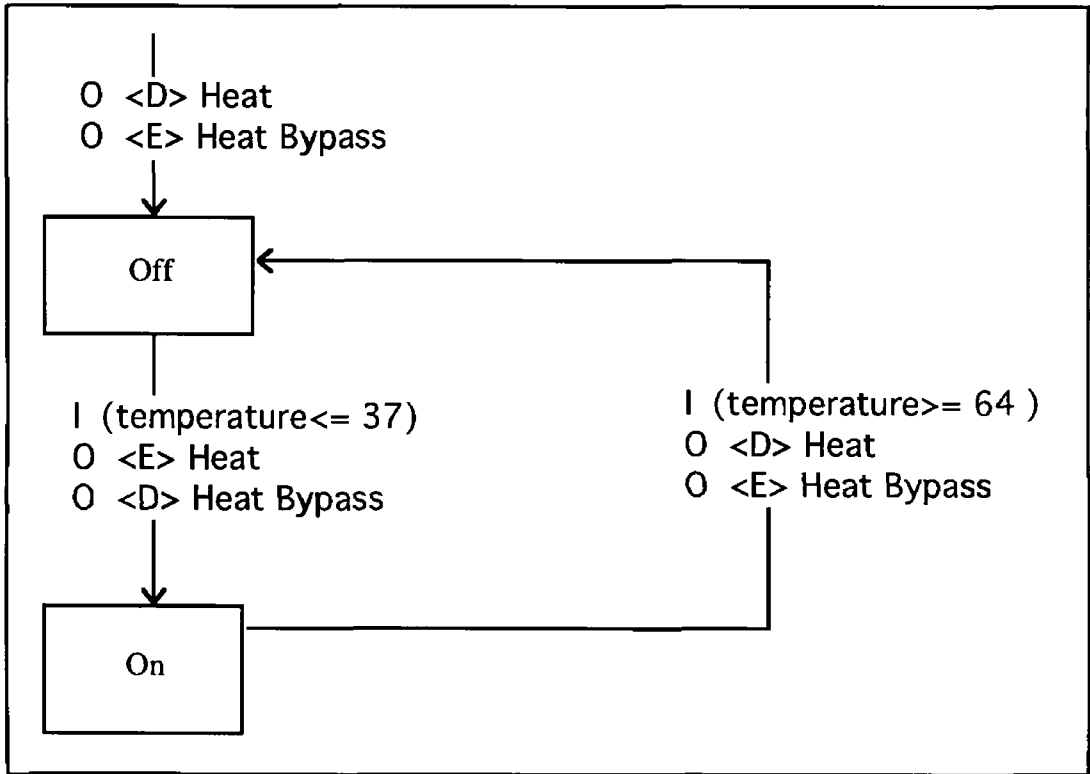


Figure 6 5 STD for Heat Control

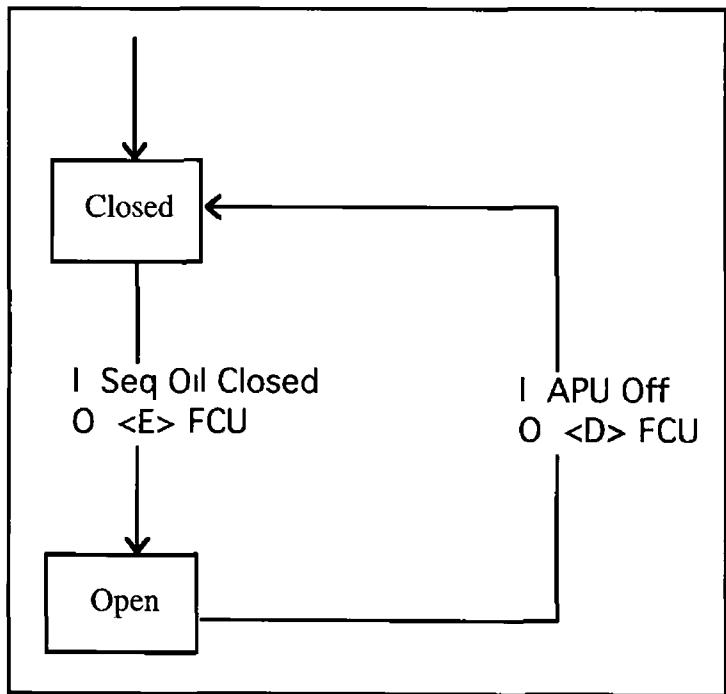


Figure 6 6 STD for Fuel Solenoid Valve

6.4 Prototype Specification in STATEMATE

For comparison purposes a prototype of the case study has been specified in STATEMATE using the Activity-chart and Statechart languages

Figure 6 7 shows the Activity-chart for the APU Fuel Subsystem The chart has been built using the STATEMATE graphical editors It consists of one control activity, four activities, and eight external activities The box-is-chart operator (@) has been used to structure the chart Unlike the ESML specification, all data flows within the model are discrete approximations of real-world quantities, such as fuel and air The data item definitions are as in the ESML specification

The control activity, *FUEL_CONTROL*, is described by the Statechart of Figure 6 8 It uses AND-lines to model the concurrent operation of the fuel shutoff and solenoid valves, which have been modelled separately in the ESML specification

The fuel shutoff valve is initially closed, but opens on receipt of an APU_On signal from Panel_P5, and activates the Pump, Heat and Filter activities This activation is defined in the forms dictionary, a "throughout" relationship exists between the *Open* state of the shutoff valve and the Pump, Heat and Filter activities In opening, the fuel shutoff valve generates the *Open_Door* event which is sent to the AIS The valve closes on receipt of an APU_Off signal, deactivates its activities and sends a *Close_Door* signal to the AIS The fuel solenoid valve works similarly, its *Open* state has a "throughout" relation on the FCU activity

The Filter activity is a basic activity which sets the filtered component of the fuel data item to true The FCU activity is an activity which has not been further described, it is non-basic

The BP Activity-chart, which details the Pump activity, is shown in Figure 6 9 It consists of two basic activities, a *Pump_F* which increases the pressure component of the fuel data item, and a *Bypass_F* which duplicates its input to output The control activity BPC is described using the Statechart shown in Figure 6 10 While in the *Off* state it keeps the *Bypass_F* activity active (via a "throughout" relation in the forms dictionary), while in the *On* state, the Pump is active, the *Bypass_F* is no longer active

The FH Activity-chart, which details the Heat activity, is shown in Figure 6 11, and is very similar to the Activity-chart of Figure 6 9 The control activity FHC is described by the Statechart in Figure 6 12 FHC changes states by testing the value of the temperature component of the input fuel flow, and activates the bypass if the temperature is normal ($\geq 64^{\circ}$ F), but activates the heater if the fuel temperature is too low ($\leq 37^{\circ}$ F)

Figure 6 7 Activity-chart for the APU Fuel Subsystem

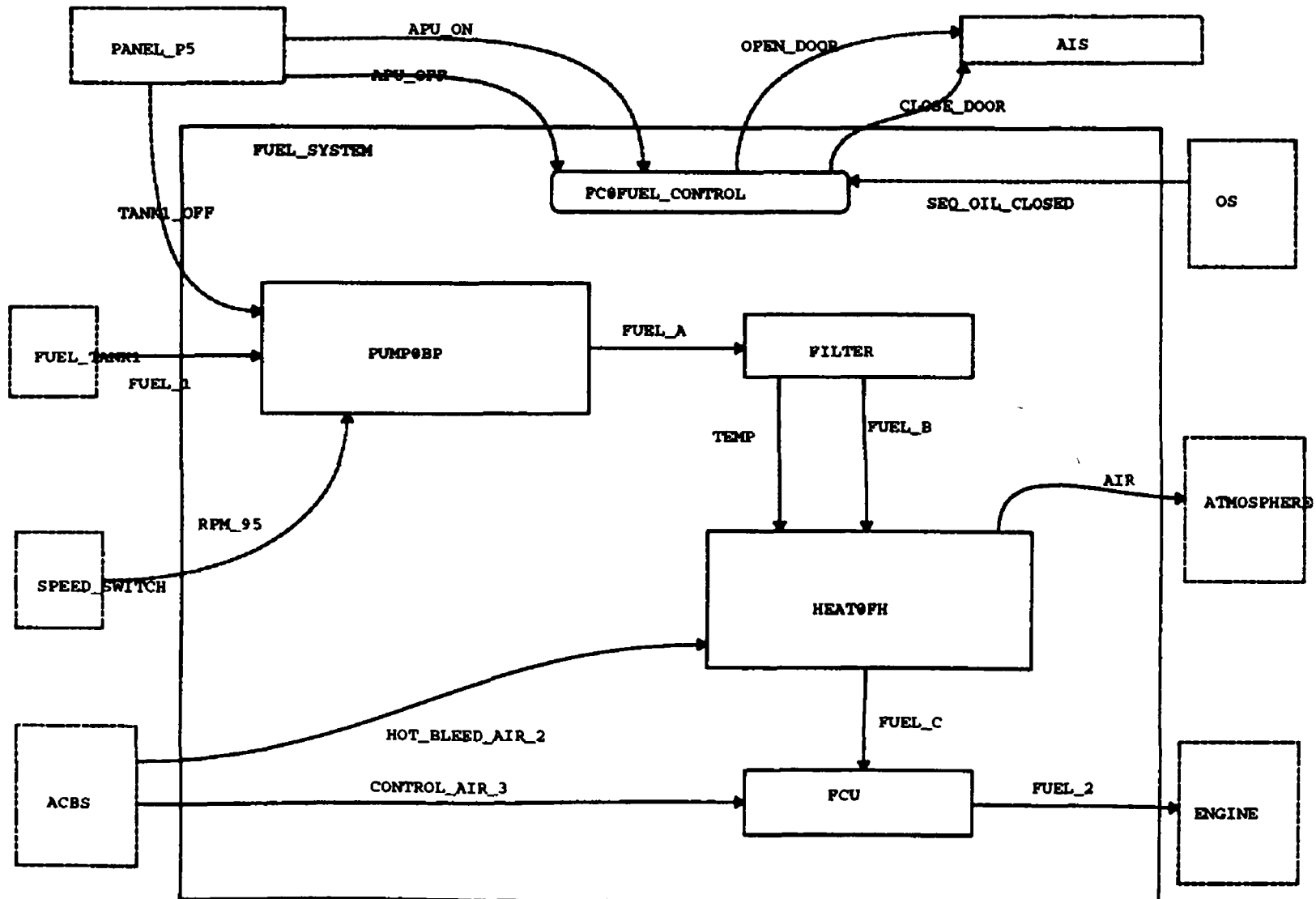


Figure 6 8 Statechart for FUEL_CONTROL

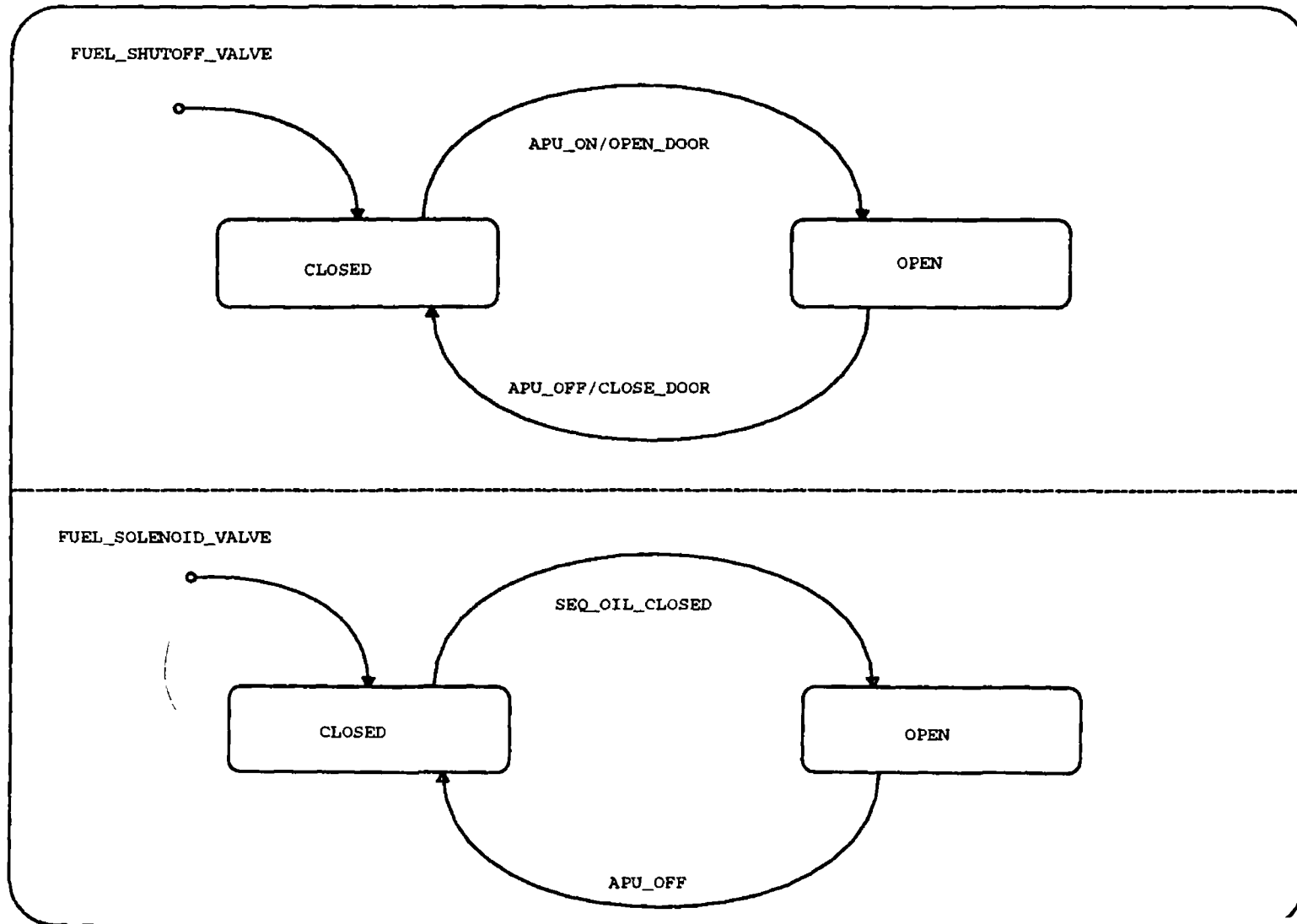


Figure 6 9 Activity-chart for BP

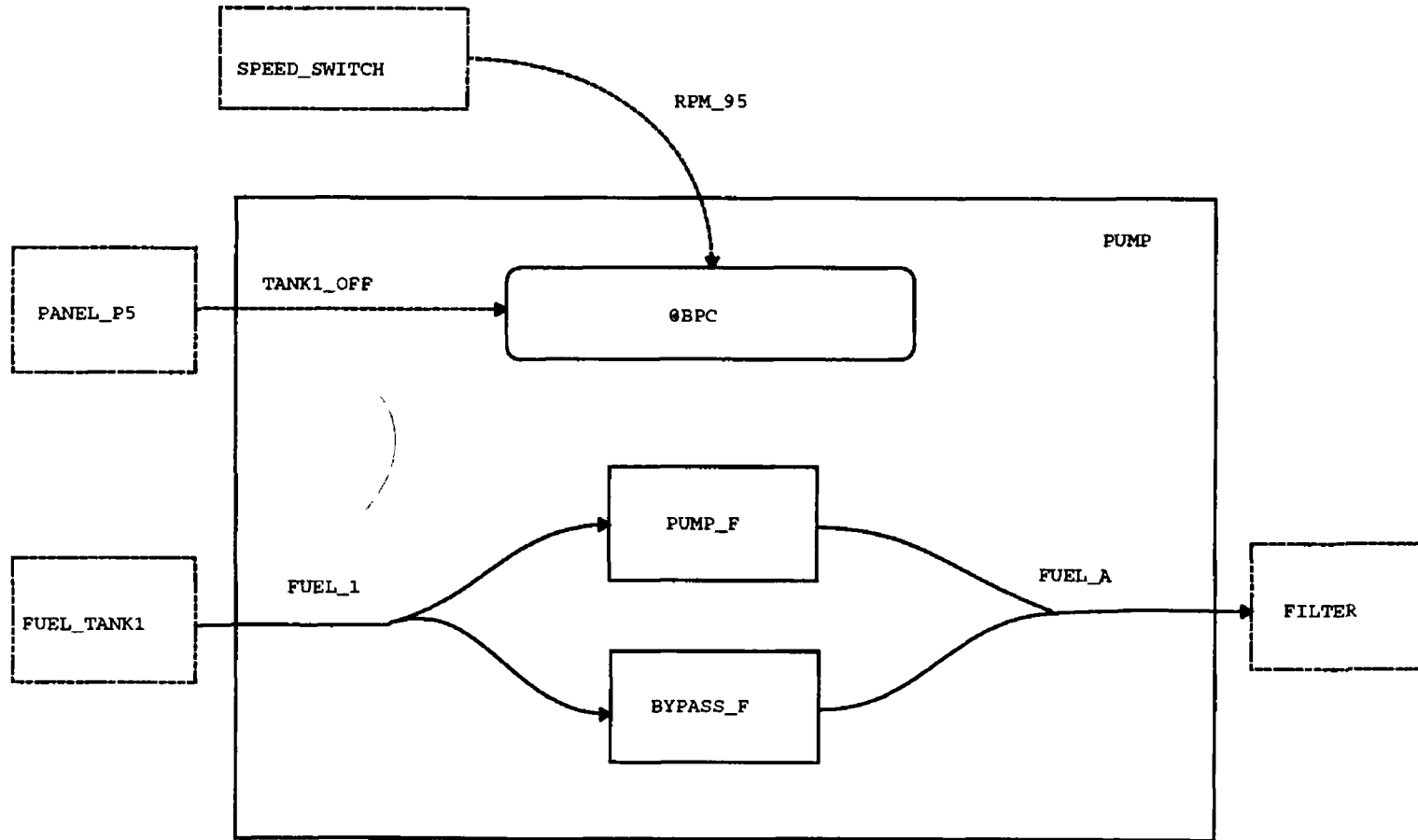


Figure 6 10 Statechart for BPC

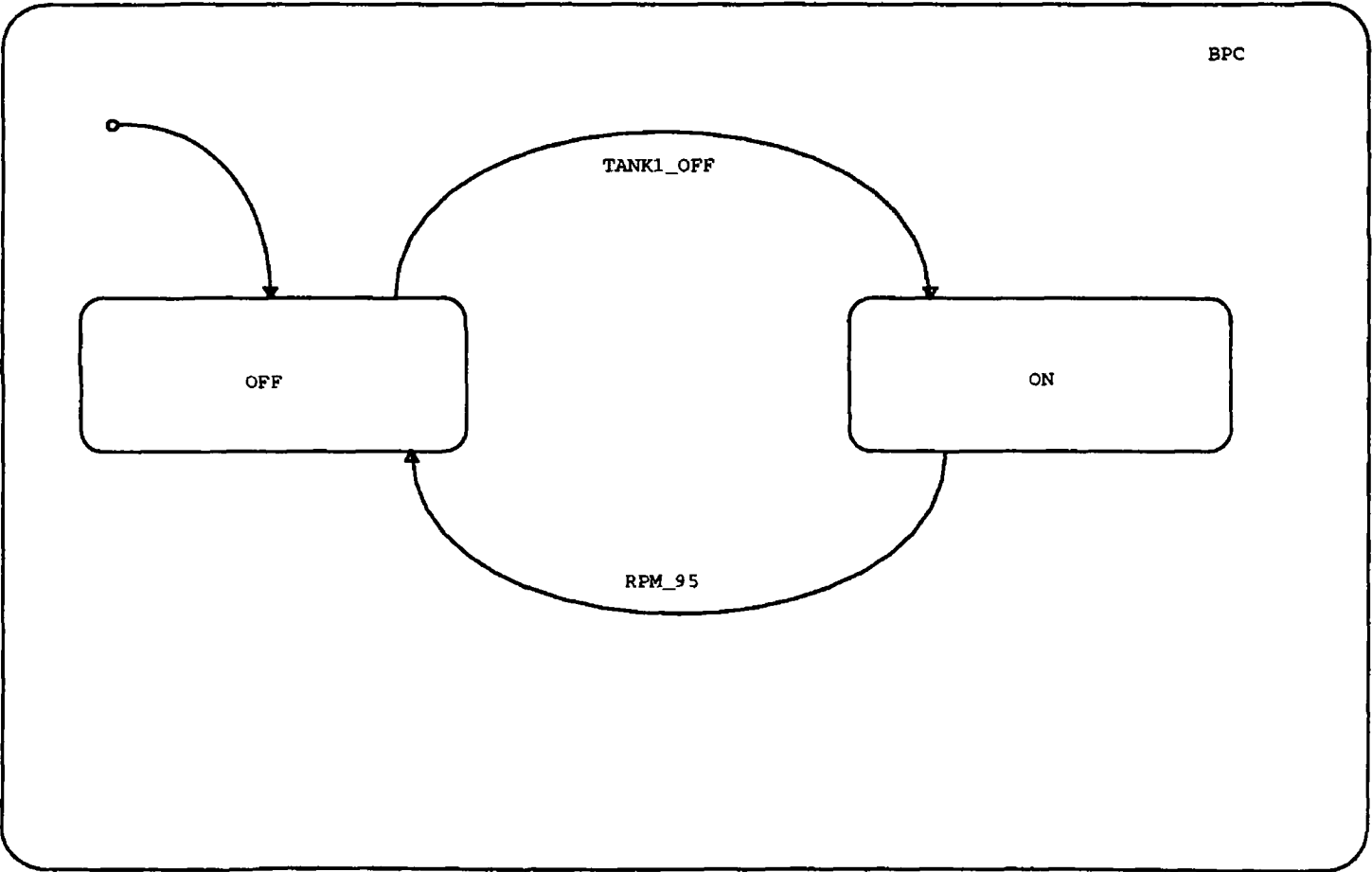


Figure 6 11 Activity-chart for FH

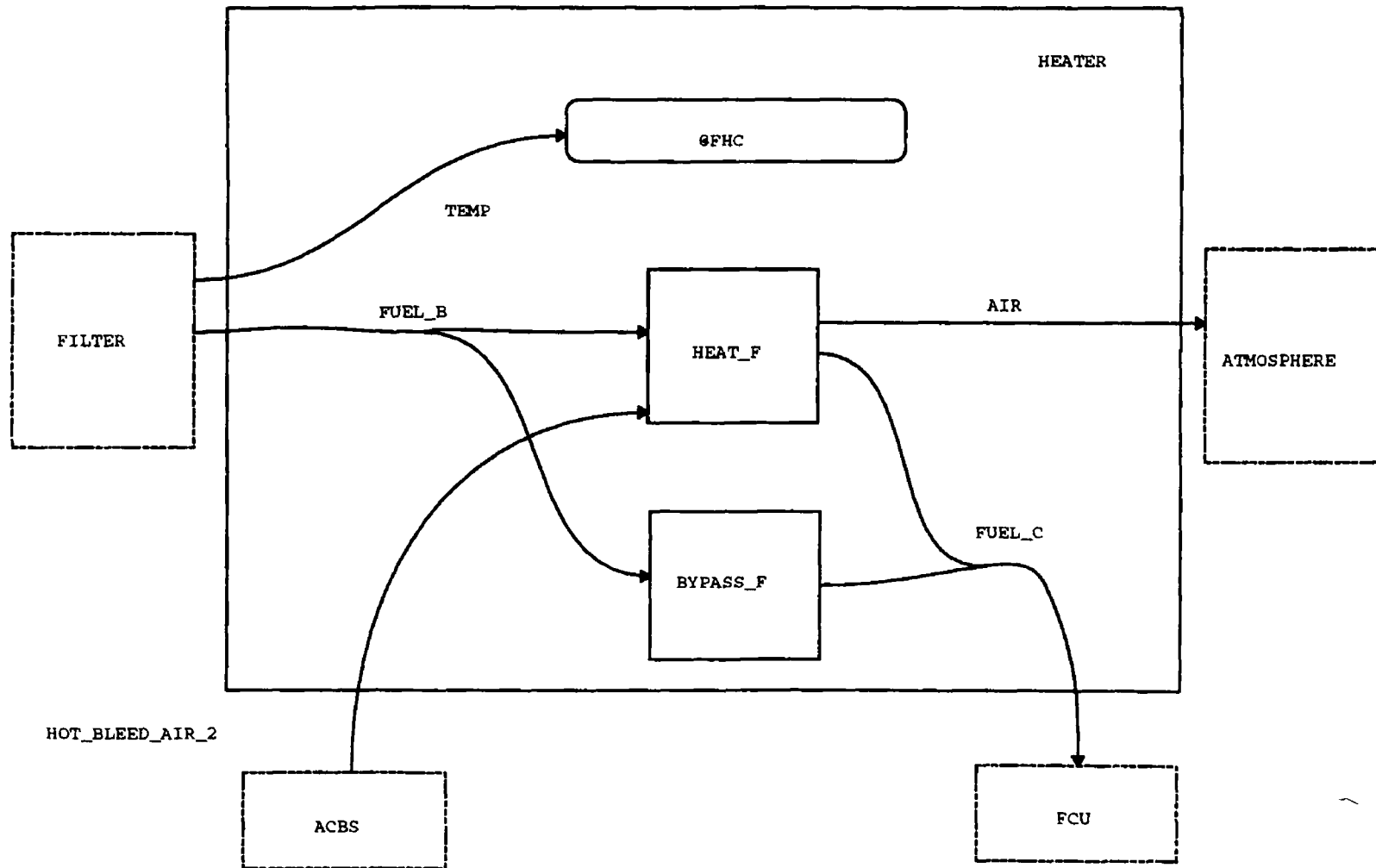
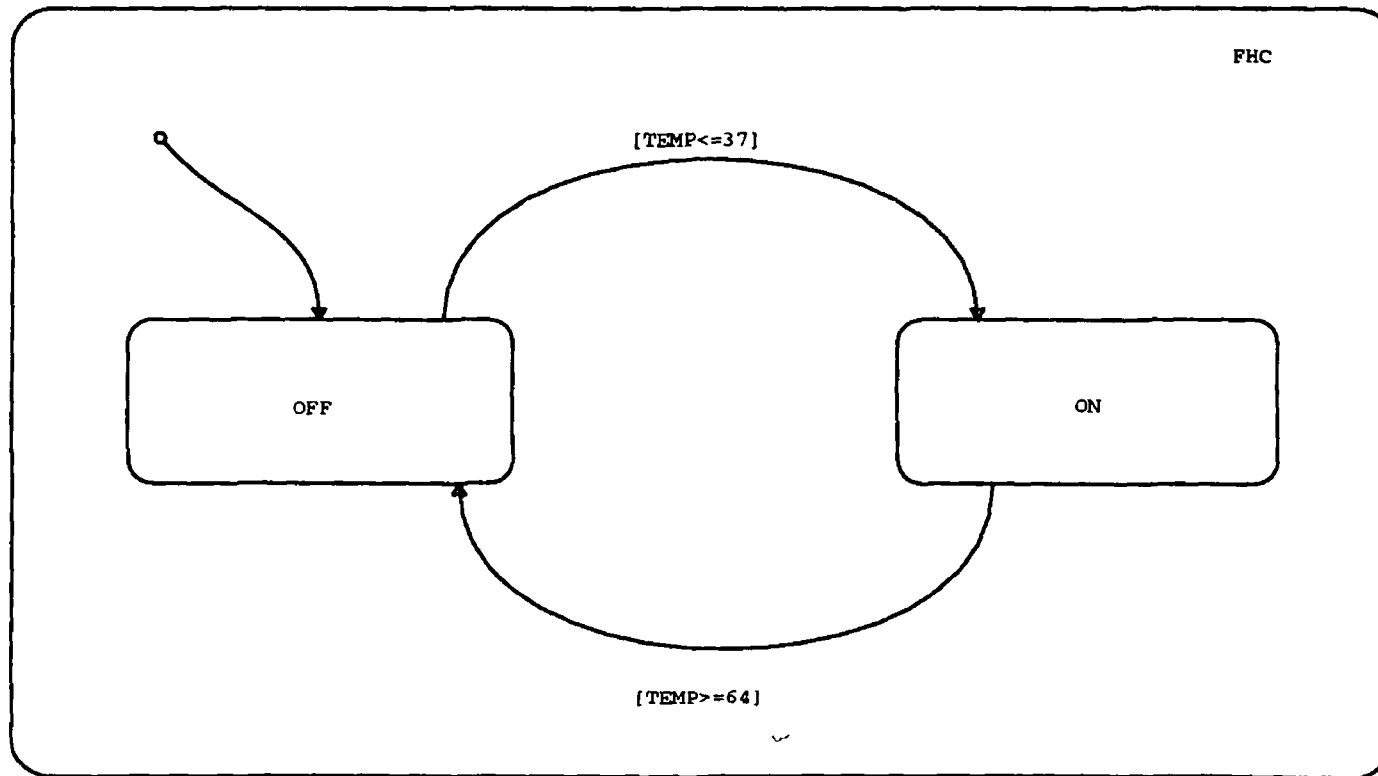


Figure 6 12 Statechart for FHC



6.5 Comparing ESML with STATEMATE

This section compares and contrasts ESML with the languages of STATEMATE by considering the prototype specifications of the case study

The ESML transformation schema and Activity-charts of STATEMATE have been used to describe the system from a functional viewpoint. Activities in Activity-charts correspond to flow transformations in ESML, both represent the functional entities of systems. Control activities in Activity-charts correspond to control transformations in ESML, both represent the control logic of systems. There are some notations in the ESML transformation schema which do not have equivalents in Activity-charts. Activity-charts do not include a notation for modelling continuous data flows, i.e. flows whose content is defined at every point in time. In the STATEMATE version of the case study, fuel and air, though real-world quantities, had to be approximated using discrete data flows. Activity-charts also have no notation for modelling depletable data stores, i.e. stores whose contents are consumed on being read.

ESML allows a transformation schema to contain several control transformations, so reflecting the distributed control nature of the system. In the ESML version of the case study, the fuel control valve and fuel solenoid valve, which are modelled using control transformations, execute concurrently according to the ESML execution rules. In Activity-charts only one control activity can exist per level. The FUEL_CONTROL control activity uses AND-lines to describe the concurrent operation of its orthogonal components, the fuel control (shutoff) and solenoid valves. By allowing multiple control activities per level, each described using separate Statecharts, AND decomposition of states, which has complicated the Statechart semantics, would be redundant.

There is an equivalence between the types of control that can be exerted in STATEMATE and ESML, start, stop, suspend and resume corresponding to enable, disable, suspend and resume respectively. Simple inspection of an ESML transformation schema reveals the types of control exerted by control transformations on flow transformations. It is clear that the fuel solenoid valve can enable and disable the FCU, but cannot suspend or resume it. In the STATEMATE version we cannot determine by inspection the type of control exerted by control activities on the activities. By looking at the activity chart it cannot be determined if the FCU is suspended and resumed by some component within the control activity. Activity charts have no graphical representation of control imposed, which would be equivalent to the prompts of ESML.

The use of static reactions, and "within" and "throughout" relationships, detracts from STATEMATE model visibility. These relationships are specified in the forms dictionary, and have no graphical representation.

From the above discussion it is clear that the ESML transformation schema are a better language for describing system functionality than the Activity-charts of STATEMATE. The former provides a more comprehensive set of modelling components, offering better model visualisation.

The STDs of ESML and the Statecharts of ESML have been used to describe the control logic of systems. The STDs used within ESML are an extension of the STD used in the Ward transformation schema to include prompts and the testing of continuous inputs. They provide a set of intuitive constructs for modelling the reactive behaviour of systems, and provide excellent model visualisation.

Statecharts are an extension of STDs to facilitate the modelling of complex reactive behaviour, and provide a richer set of modelling components than STDs. Statecharts are therefore a superior language to STDs. Statecharts allow the hierarchical decomposition of states, no such facility exists in STDs. The transition labelling syntax of Statecharts provides much flexibility. However transitions, which move across state boundaries, lower model clarity.

STATEMATE lacks a well accepted development method for its languages. Israel Aircraft Industries have developed a specialised analysis method, *ECSAM* (Embedded Computer Systems Analysis Method) [Winokur89] [Winokur90] for use on a special project, an electro-optical embedded computer system designed for operation on helicopter platforms. *ECSAM* is a specialist analysis method, and not suited to general real-time systems development, indeed STATEMATE does not support *ECSAM* in full. ESML does not suffer any such problem, the Ward/Mellor RTSA/SD method is a widely accepted method which can guide its use.

A disadvantage of ESML is that the only way time can be modelled is by the use of fixed time delays for flow transformations. Statecharts provide more comprehensive constructs for modelling time, timeout events and scheduled actions.

6.6 Translating the ESML specification into a LOOPN net specification

This section follows the translation process described in section 5.4 to generate an executable LOOPN specification from a non-executable ESML specification of the APU Fuel Subsystem case study.

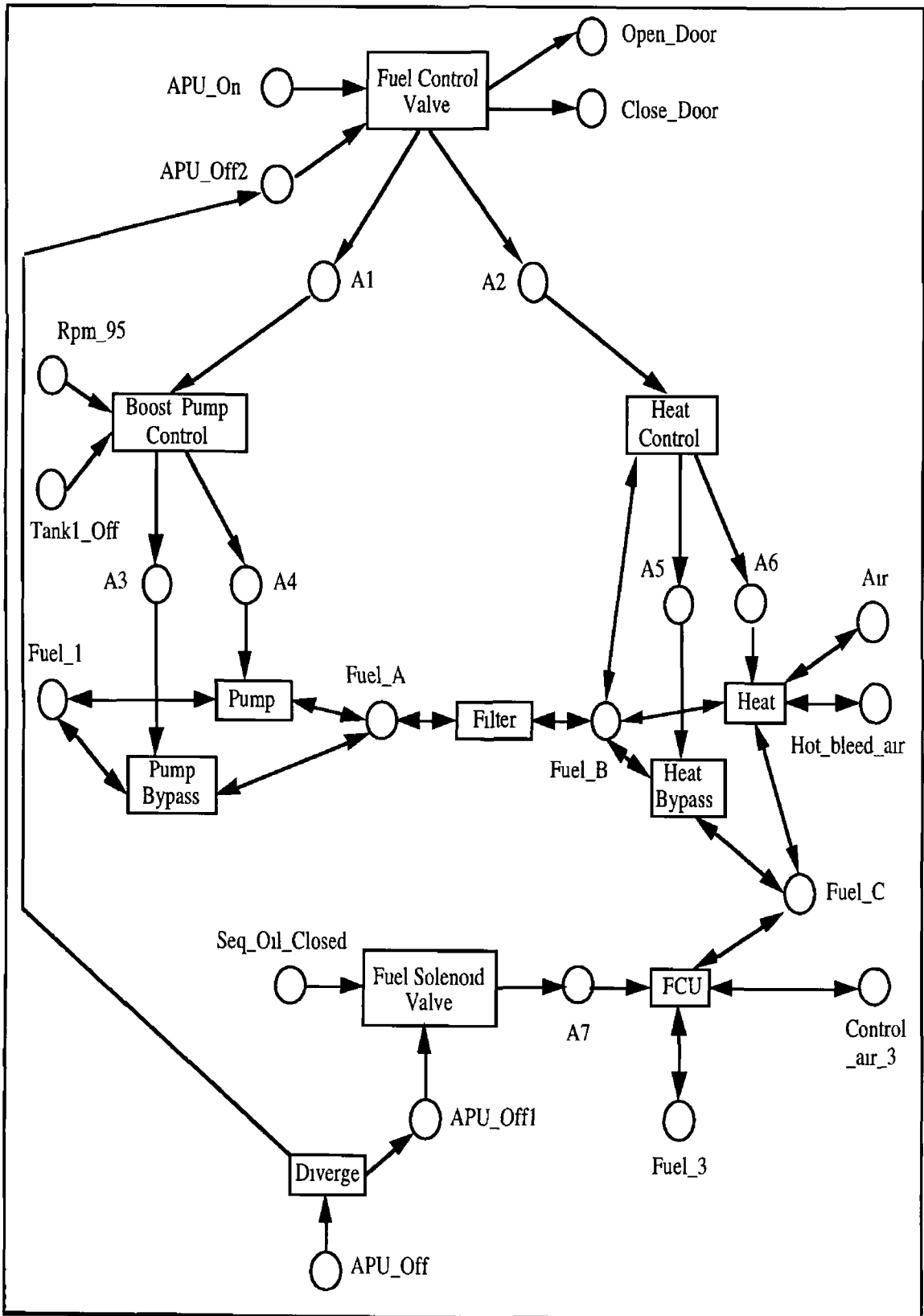


Figure 6 13 Top-level LOOPN net equivalent of the APU Fuel Subsystem

Step 1 requires the identification of flow types and the application of the templates shown in Figure 5.2. This step also requires the user to consider convergence/divergence of flows and the connection of flows to terminators. The LOOPN net produced by this step is shown in Figure 6.13, it depicts all transformations as super-transitions (to be further refined as in steps 3 and 4). Figure 6.13 is "top-level" since it describes the overall connection of flows and transformations.

Figure 6.13 contains continuous data flows, prompts and signals. An example of a continuous data flow is "Fuel_I", a place that holds tokens of type "fueltype", each token contains "filtered", "temperature" and "pressure" attributes. An example prompt is "AI", a place which holds tokens of type "prompttype", each token contains an "E", "D", "S", "R" or "T". An example signal is "Open_Door", the place is unstructured ("null").

The three places "APU_On", "APU_Off" and "Tank1_Off" model interaction with the "Panel_P5" terminator. These correspond to the three flows that connect the terminator to the system. The places which represent these three flows are delayed so that their tokens become visible as execution proceeds. Another example of a terminator is "Fuel_Tank1", which is modelled by one place, "Fuel_I", which is marked in the initial marking, with the "filtered" attribute set to false, the "temperature" attribute set to 10, the "pressure" attribute set to 0, and the undefined attribute set to false.

The templates shown in Figure 5.4 has been used to generate a LOOPN net to model the divergence of the "APU_Off" signal flow. Since there are no stores in the ESML specification, step 2 is skipped.

6.6.1 Special Modules

The translation process has described the need for two special modules. They are described in the following paragraphs.

6.6.1.1 Types Module

The Types module is a special module used to define the place types used within the LOOPN net model. It defines the type of prompts ("prompttype"), and types for fuel ("fueltype") and air ("airtype") continuous flows. Fuel contains "filtered", "temperature", and "pressure" attributes. Air contains just "temperature" and "pressure" components.

The Types module uses token type inheritance to include the "undefined" attribute in the type definition of continuous flows. The Types module is inherited by all other modules within the system so that the type definitions can be used. The LOOPN source program for the Types module is shown in Appendix A.

6.6.1.2 APU Module

The APU module is the main driver module for the prototype. It defines places for signals, continuous flows, and prompts, and links instances of modules. It defines the time delays to be associated with signals from terminators, and the initial values of continuous flows. The initial marking is set-up by the special "initial" transition. The "diverge" transition is defined to create a duplicate of the "APU_On" signal. The module instances are created by filling in actual places for module parameter places. The APU module specifies the top-level LOOPN net of Figure 6.13. The LOOPN source program for the APU module is shown in Appendix A.

6.6.2 Flow Transformations

Step 3 specifies the identification of flow transformation types and the application of the appropriate template. The following paragraphs describe each flow transformation and the LOOPN net modules generated.

6.6.2.1 Pump

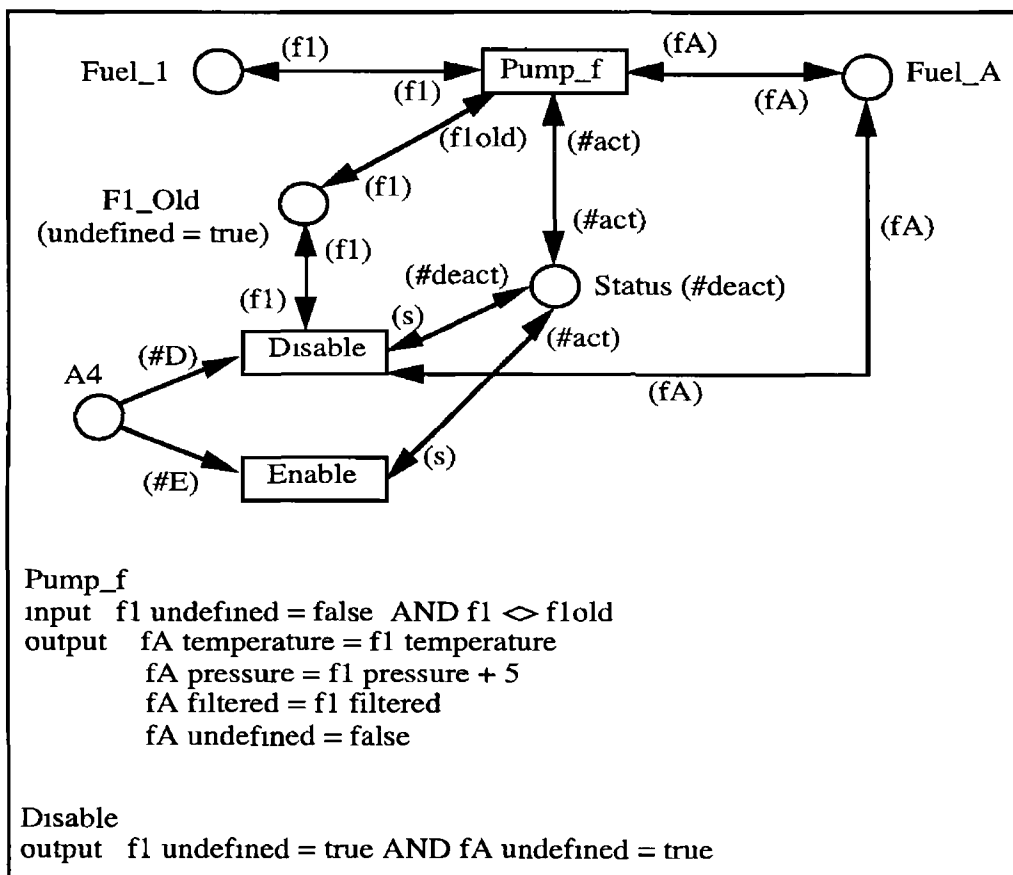


Figure 6.14 LOOPN net module for Pump

The fuel pump is responsible for supplying pressurised fuel to the APU engine. It is a primitive flow transformation with an Activate input prompt, "A4", from the boost pump controller, one continuous input, "Fuel_I", and one continuous output, "Fuel_A".

The "Pump_f" transition implements the pump mini-specification. It increases the "pressure" attribute of the incoming fuel flow by 5 units. This is implemented in the output token generation section of the transition.

The translation is achieved by considering the template of Figure 5.11. The LOOPN net module is shown in Figure 6.14, its LOOPN source language specification is shown in Appendix A.

6.6.2.2 Pump Bypass

The pump bypass is responsible for conveying fuel from its input to the fuel filter. It is a primitive flow transformation with an Activate input prompt, "A3", from the boost pump controller, one continuous input, "Fuel_I", and one continuous output, "Fuel_A". The "Bypass_f" transition simply duplicates its input at the output stage.

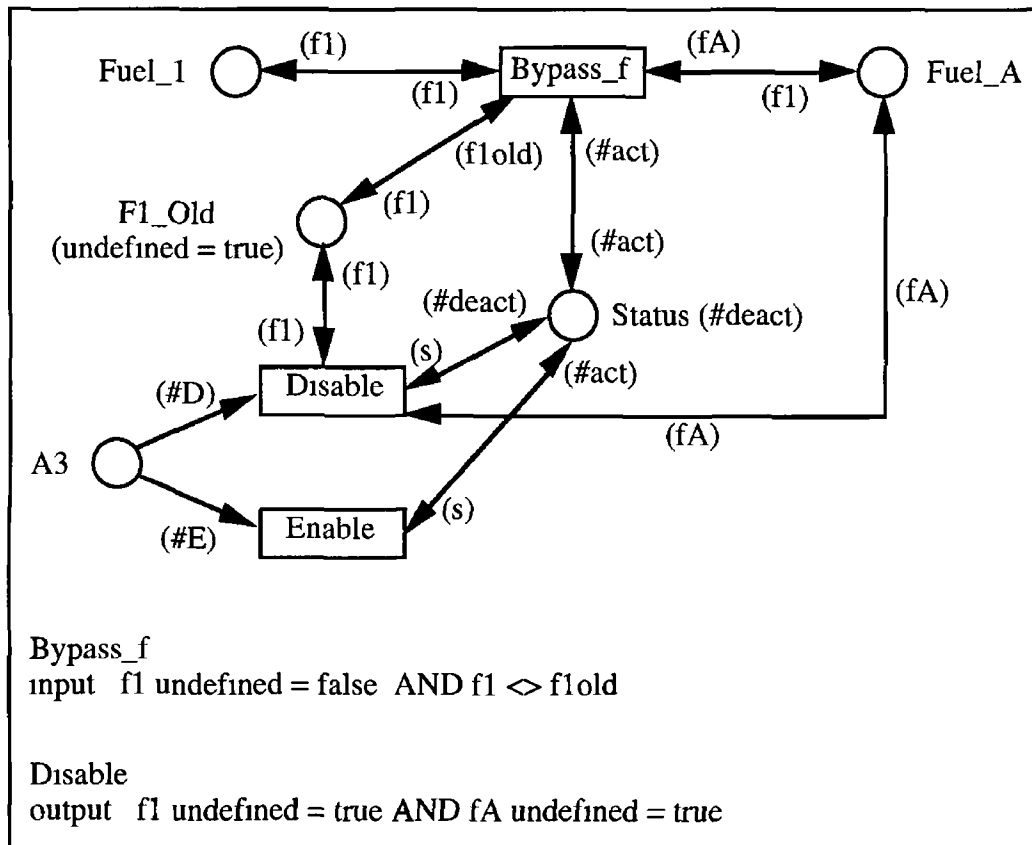


Figure 6.15 LOOPN net module for Pump Bypass

The translation is achieved by considering the template of Figure 5 11 The LOOPN net module is shown in Figure 6 15, its LOOPN source language specification is shown in Appendix A

6 6.2.3 Filter

The filter is responsible for filtering fuel passing through it It is a primitive flow transformation with no input prompt, one continuous input, "Fuel_A", and one continuous output, "Fuel_B" The "Bypass_f" transition sets the "filtered" attribute of the outgoing fuel to true

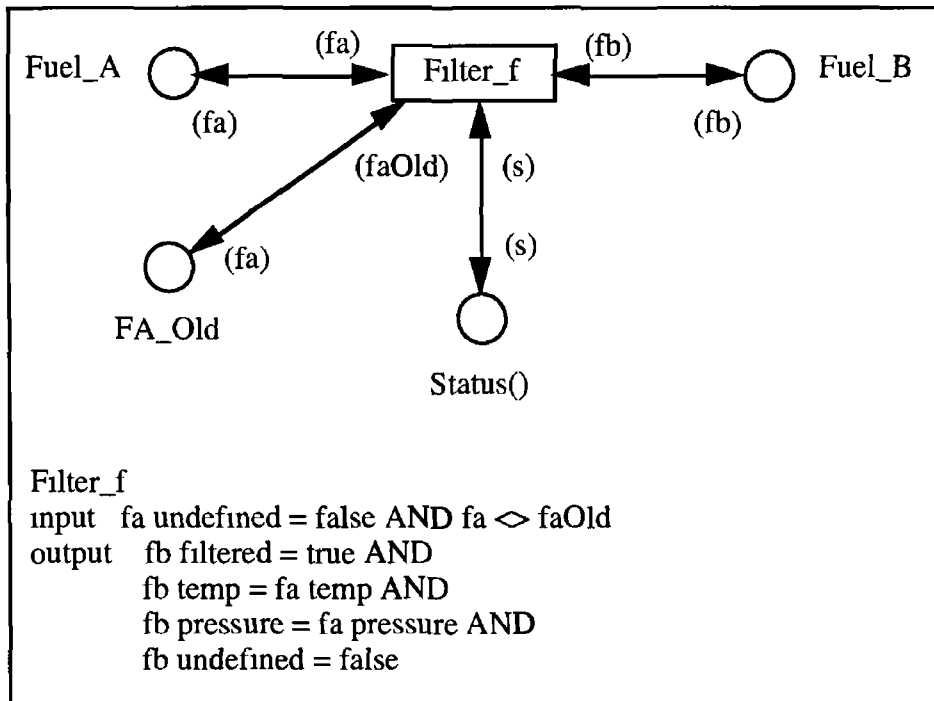


Figure 6 16 LOOPN net module for Filter

The translation is achieved by considering the template of Figure 5 8 The LOOPN net module is shown in Figure 6 16, its LOOPN source language specification is shown in Appendix A

6.6.2 4 Heat

The heater is responsible for heating fuel passing through it It is a non-primitive flow transformation with an Activate input prompt, "A6", two continuous inputs, "Fuel_B" and "Hot_bleed_air_2", and two continuous outputs, "Air", and "Fuel_C" The heater uses the hot bleed air to heat incoming fuel, and expels the resultant air into the

atmosphere The transformation has not been refined to a primitive level, and is therefore not described using a mini-specification

The "Heat_f" transition, since it implements a non-primitive transformation, just its defines its continuous outputs, and does not consider the other token attributes

The translation is achieved by considering the template of Figure 5 11 The LOOPN net module is shown in Figure 6 17, its LOOPN source language specification is shown in Appendix A

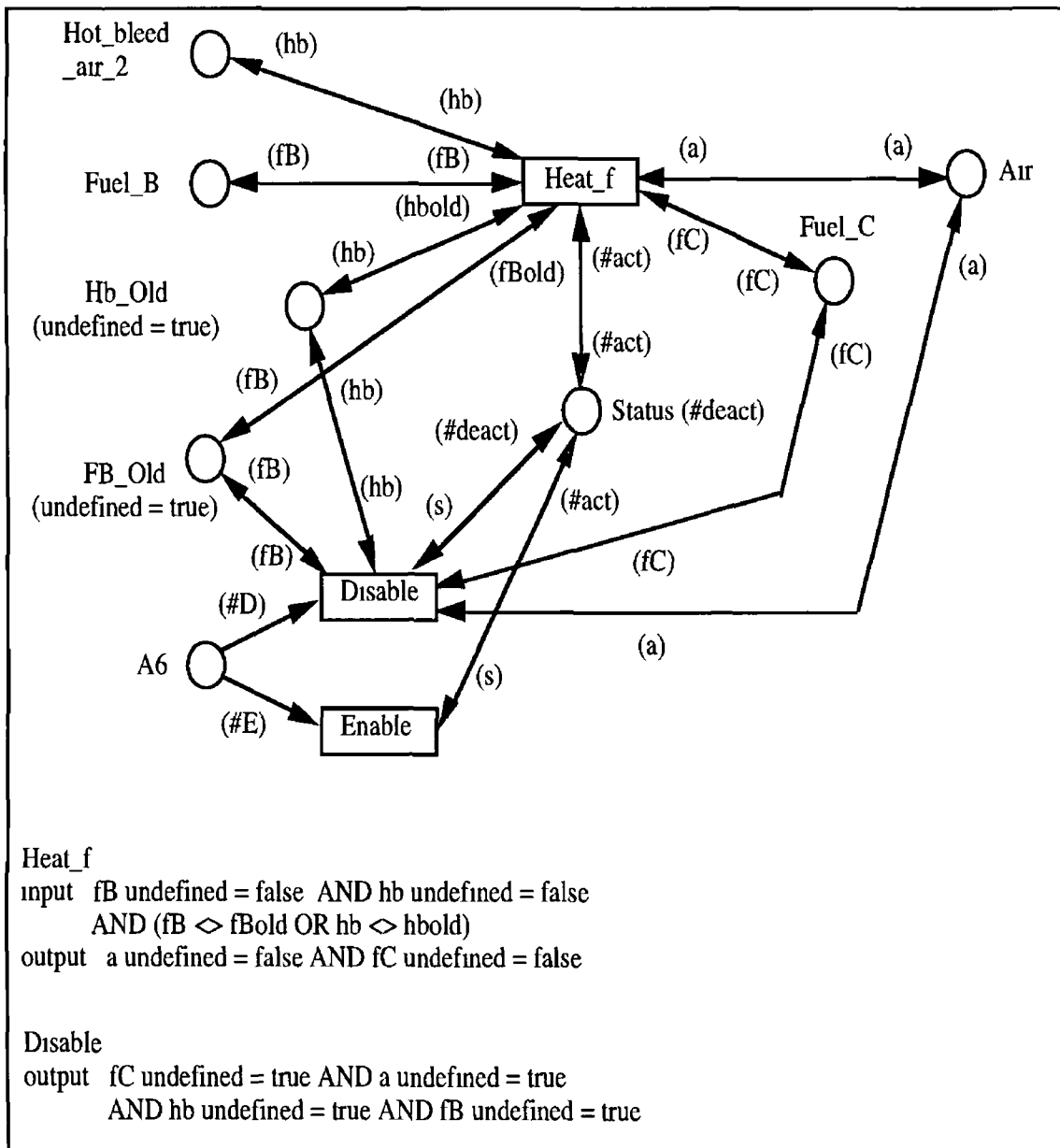


Figure 6 17 LOOPN net module for Heat

6.6.2.5 Heat Bypass

The heat bypass is responsible for conveying fuel from its input to the FCU. It is a primitive flow transformation with an Activate input prompt, "A5", from the heat controller, one continuous input, "Fuel_B", and one continuous output, "Fuel_C". The "Bypass_f" transition simply duplicates its input at the output stage.

The translation is achieved by considering the template of Figure 5.11. The LOOPN net module is shown in Figure 6.18, its LOOPN source language specification is shown in Appendix A.

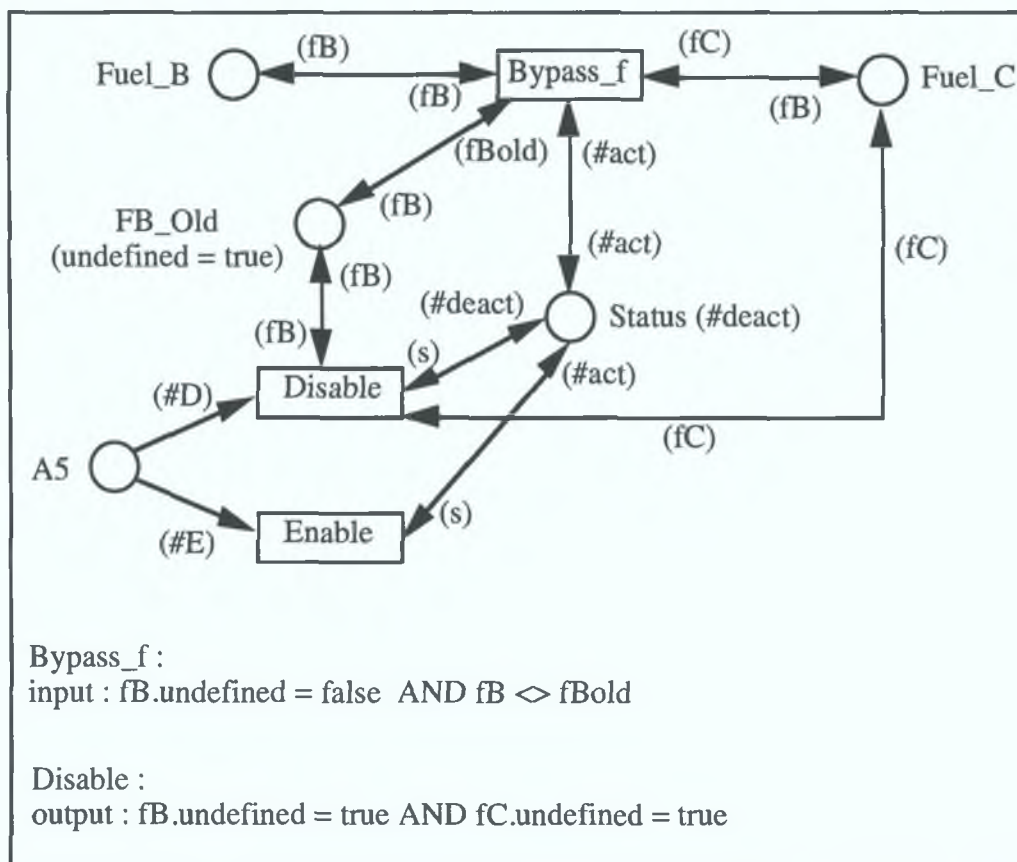


Figure 6.18 LOOPN net module for Heat Bypass

6.6.2.6 Fuel Control Unit

The FCU is responsible for regulating the fuel flow to the APU engine. It is a non-primitive flow transformation with an Activate input prompt, "A7", two continuous inputs, "Fuel_C" and "Control_air_3", and one continuous output, "Fuel_3".

The "Heat_f" transition, since it implements a non-primitive transformation, just defines its continuous output and does not consider the other token attributes.

The translation is achieved by considering the template of Figure 5 11 The LOOPN net module is shown in Figure 6 19, its LOOPN source language specification is shown in Appendix A

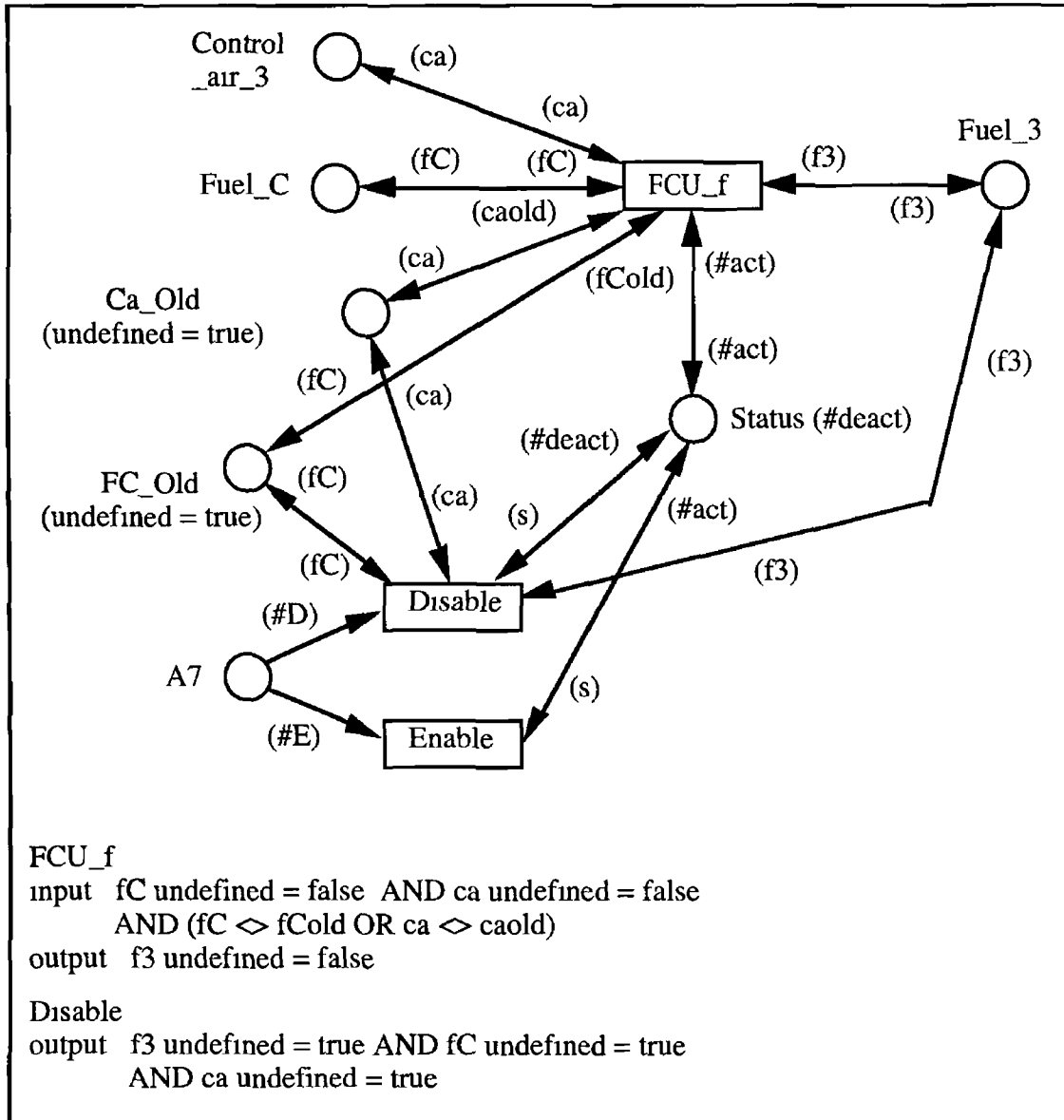


Figure 6 19 LOOPN net module FCU

6.6.3 Control Transformations

Step 4 specifies the identification of control transformation type and the application of the appropriate template The following paragraphs describe each control transformation and the LOOPN net modules generated

6.6.3.1 Fuel Control Valve

The fuel control valve is responsible for controlling the boost pump control and heat control modules. It is a control transformation with no input prompt, two input signals, "APU_On" and "APU_Off2", two output signals, "Open_Door" and "Close_Door", and two Activate output prompts, "A1" and "A2". The fuel control valve is either closed ("#closed") or open ("#open").

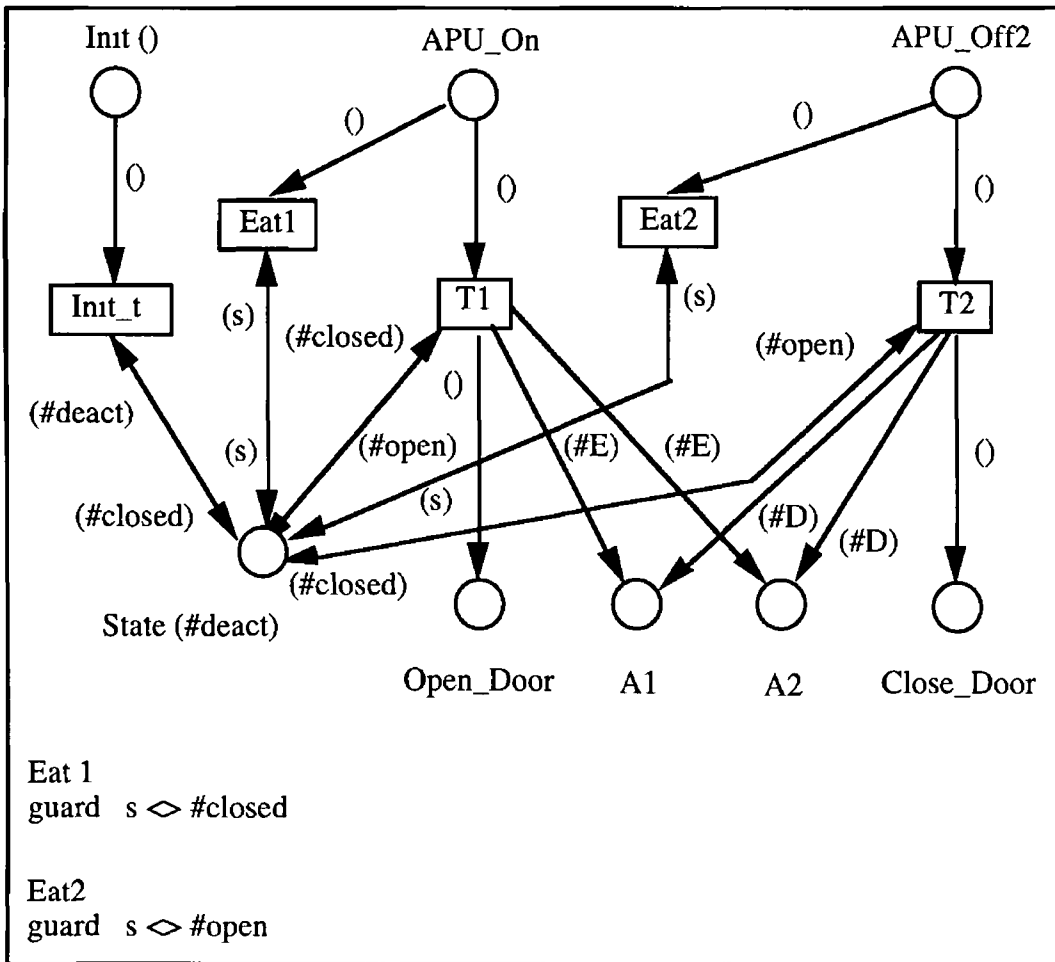


Figure 6 20 LOOPN net module for Fuel Control Valve

The translation is achieved by considering the template of Figure 5 13. The LOOPN net module is shown in Figure 6 20, its LOOPN source language specification is shown in Appendix A.

6.6.3.2 Boost Pump Control

The boost pump control is responsible for controlling the boost pump and pump bypass. It is a control transformation with an Activate input prompt, "A1", from the fuel control valve, two input signals, "Rpm_95", and "Tank1_Off", and two Activate output prompts, "A3" and "A4". The boost pump controller is either deactivated, on (the boost pump is on), or off (the pump bypass is on).

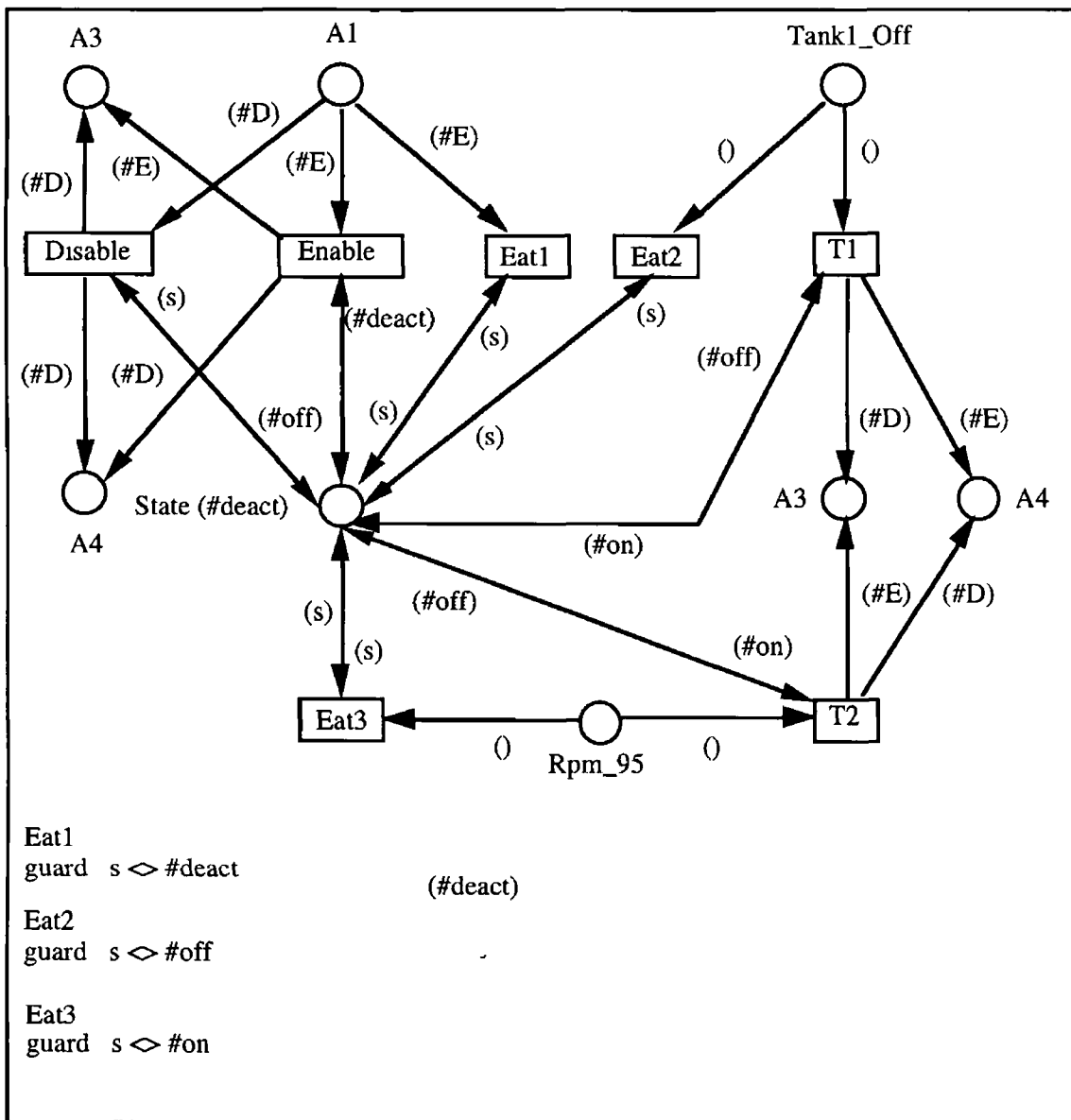


Figure 6 21 LOOPN net module for Boost Pump Control

The translation is achieved by considering the template of Figure 5 14. The LOOPN net module is shown in Figure 6 21, its LOOPN source language specification is

shown in Appendix A Note that the LOOPN net contains two identically name places
 This is a drawing convenience, they represent the same place

6 6.3.3 Heat Control

The heat control is responsible for controlling the heater and heater bypass It is a control transformation with an Activate input prompt, "A2", from the fuel control valve, an input continuous flow, "Fuel_B", and two Activate output prompts, "A5" and "A6"

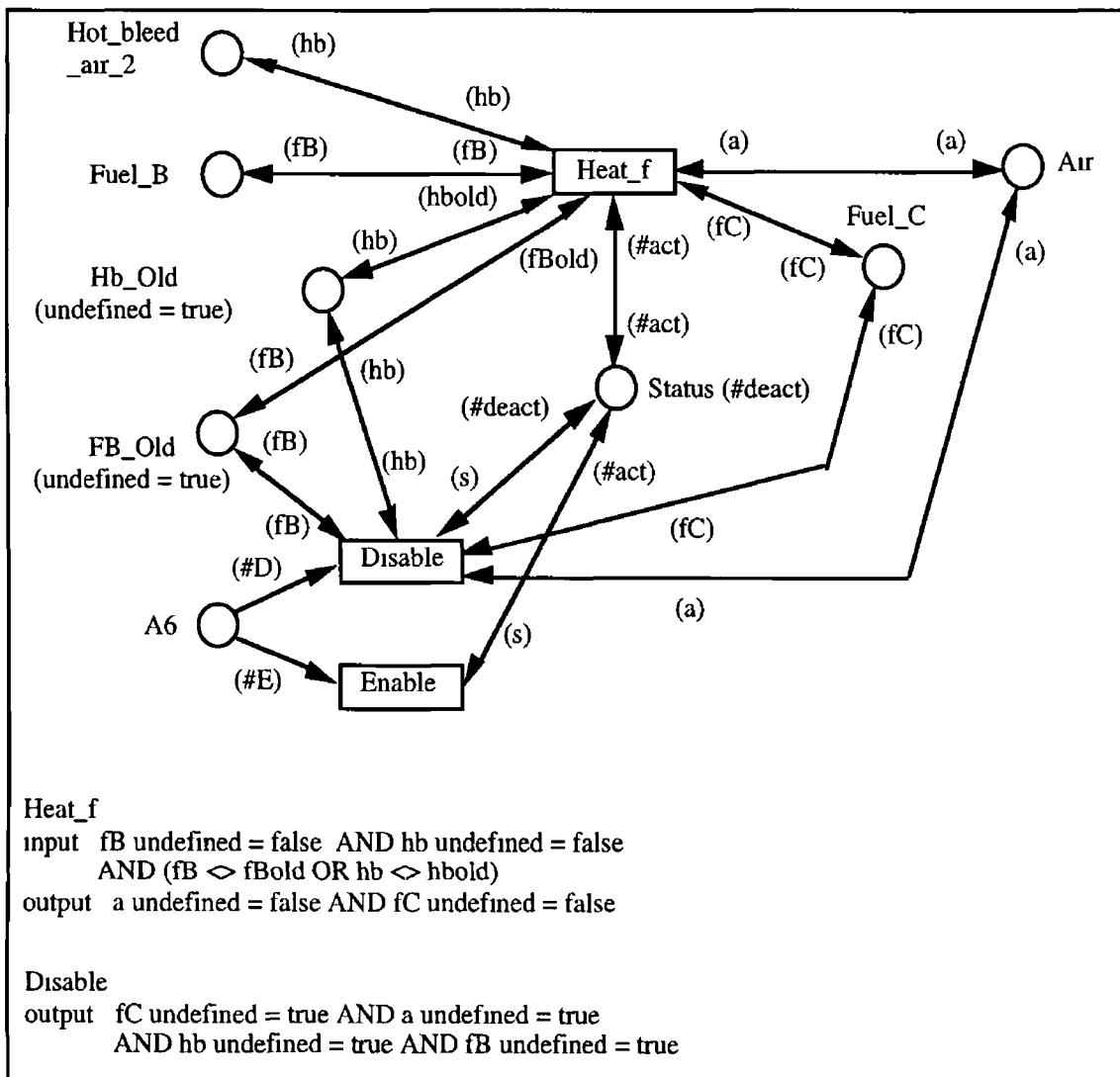


Figure 6 22 LOOPN net module for Heat Control

The "temperature" attribute of "Fuel_B" is used to trigger state transitions and cause the enabling and disabling of the heat and heat-bypass flow transformations Transitions "T1" and "T2" operate by checking the value of this attribute The heat controller is either deactivated, off (the heat bypass is on), or on (the heater is on)

The translation is achieved by considering the template of Figure 5 14 The LOOPN net module is shown in Figure 6 22, its LOOPN source language specification is shown in Appendix A

6.6.3.4 Fuel Solenoid Valve

The fuel solenoid valve is responsible for controlling the (FCU) It is a control transformation with no input prompt, two input signals, "Seq_Oil_Closed" and "APU_Off1", and an Activate output prompt, "A7", to the FCU The fuel solenoid is either closed (the FCU is off), or open (the FCU is on)

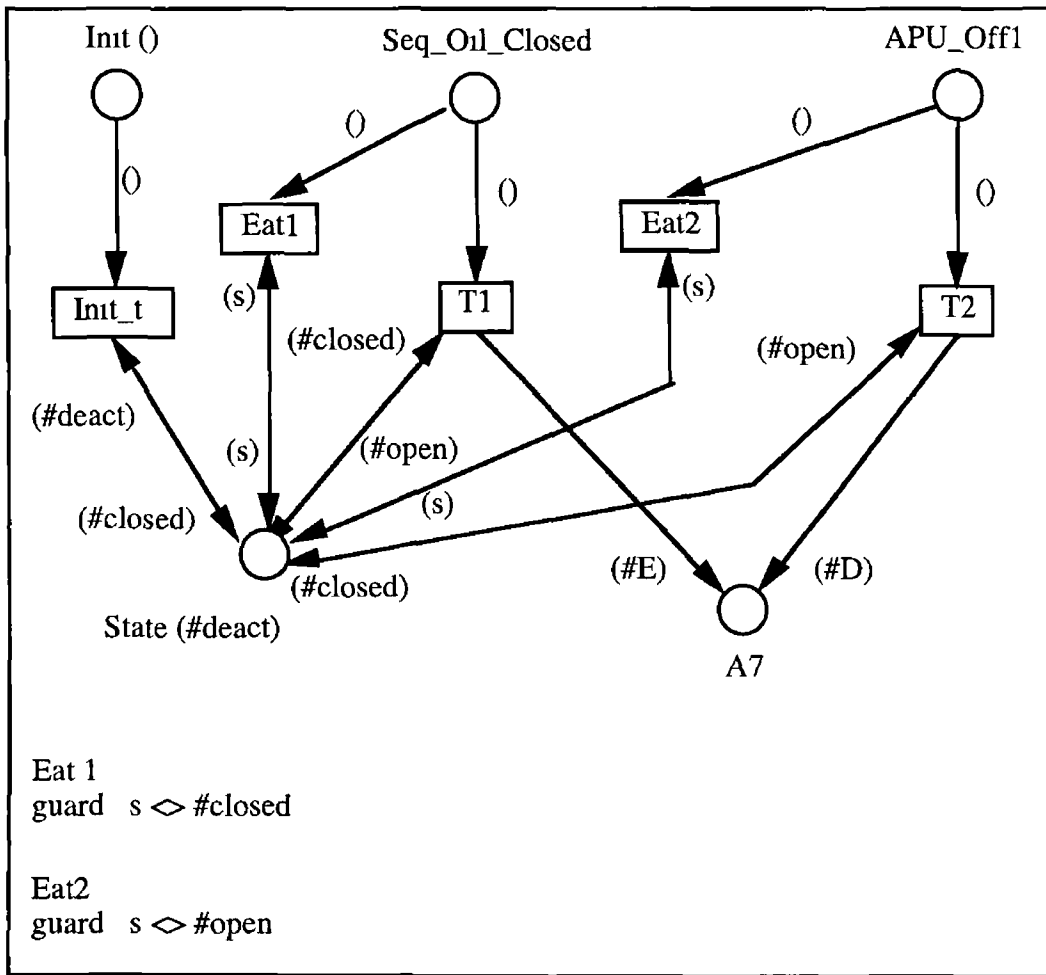


Figure 6 23 LOOPN net module for Fuel Solenoid Valve

The translation is achieved by considering the template of Figure 5 13 The LOOPN net module is shown in Figure 6 23, its LOOPN source language specification is shown in Appendix A

6.7 C Language Prototype Generation

The LOOPN tool has been used to generate a C program which has been compiled and run as a prototype of the case study. The program is long and difficult to understand and therefore has not been included as an appendix.

Appendix B shows the output produced by the running prototype. The output, which is produced by the auxiliary action sections of the LOOPN transitions, reports the simulation progress and the simulation time. It also indicates which transition has fired, this was used as a debugging aid which would be removed when the translation process becomes automatic and the user unaware of the LOOPN formalism. The output reflects the distributed execution strategy defined by the ESML execution rules. Observation of the prototype can be used to study indeterminism within the system and possible event concurrency. The prototype can be used to aid the requirements specification effort. The output is used to revise the prototype for future iterations in the prototyping approach.

The prototype of the case study built using STATEMATE has been interactively simulated using the STATEMATE simulation tool component. Appendix C shows the output produced by the prototype. The output, like the above, is textual, though STATEMATE facilitates the connection of specifications to simulated interfaces.

6.8 Summary

This chapter has discussed the APU Fuel Subsystem case study, and applied the ESML/LOOPN prototyping system to produce a running prototype of it. The prototype has been specified in ESML, an equivalent specification has been built in STATEMATE for comparison purposes. It has been noted that ESML is a more superior language for describing the functional viewpoint of a system than Activity-charts, but it lacks the intricacies provided by Statecharts for modelling complex reactive behaviour. The translation templates have been applied to translate a non-executable prototype specification of the case study into an executable one. The LOOPN nets have been specified in the LOOPN source language. The prototype has been exercised in the target environment. The prototyping system has provided a framework for using ESML as a graphical executable specification language. The LOOPN code generator has been used to generate a C program from the LOOPN specification which can be compiled and run as the prototype in the target environment. The following chapter offers conclusions, analysis, and ideas about further work.

Chapter 7

Conclusions and Further Work

7.1 Introduction

This chapter summarises the research which has resulted in this thesis, critically analyses the ESML/LOOPN prototyping system, and offers some ideas about further work in the area.

7.2 Research Summary

The research forming the basis of this thesis has resulted in definition of the ESML/LOOPN prototyping system which facilitates the use of ESML as a graphical executable specification language, i.e. it can be used to build prototypes. Prototyping is a new paradigm for software development which has been developed to overcome the deficiencies of the traditional software development paradigm, i.e. the waterfall life cycle model. This thesis has described how prototyping is applicable to real-time systems. The ESML/LOOPN prototyping system currently facilitates the construction of exploratory prototypes of real-time systems for use within a keep-it prototyping approach.

The ESML/LOOPN prototyping provides a set of translation templates which can be used to translate non-executable ESML specifications into executable LOOPN specifications. An unambiguous dialect of ESML has been defined for use with the Ward/Mellor RTSA/SD method. The translation templates are based on a set of guidelines (the ESML execution rules) which have been defined for ESML to allow prediction of the behaviour of an ESML specification over time. The execution rules have been specified in terms of Petri net tokens to allow quantitative, rather than qualitative, specification execution. LOOPN nets, an object-oriented high-level Petri net formalism, have been therefore used to define an execution semantics for ESML, they provide a rigorous interpretation of the ESML constructs and their combinations. The ESML/LOOPN prototyping system uses the LOOPN code generator to automatically produce a C program from a LOOPN net specification. This program can be compiled and run as the prototype in the target environment. The LOOPN code generator is just one of a number of Petri net code generators surveyed in the thesis. The prototyping system has been applied to the case study, i.e. the APU Fuel Subsystem STATEMATE, a popular CASE tool for real-time systems, has been used as a benchmark for evaluating the ESML/LOOPN system.

In summary, the thesis has

- described the waterfall life cycle model and its deficiencies
- described an alternative paradigm, i.e. prototyping, and how it can be applied to real-time systems
- described an unambiguous dialect of ESML and its use with the Ward/Mellor RTSA/SD method
- defined a set of execution rules (guidelines) for ESML in terms of Petri net tokens which facilitate quantitative specification execution
- described LOOPN nets and the operation of a number of Petri net code generators (including the LOOPN code generator)
- defined the ESML/LOOPN prototyping system which facilitates the translation of non-executable ESML specifications into executable LOOPN specifications
- defined the translation templates used in the translation process
- applied the prototyping system to generate a prototype of the APU Fuel Subsystem the case study
- compared ESML with the languages of STATEMATE by considering the prototypes constructed of the case study using both

7.3 Analysis

The following paragraphs provide a critical analysis of the ESML/LOOPN prototyping system

The ESML/LOOPN prototyping system provides a simple framework for using ESML as a graphical executable specification for the prototyping of real-time systems. The system has bound together existing tools (TurboCASE and LOOPN) and a method (Ward and Mellor RTSA/SD) to form a complete approach.

The ESML structured language is an intuitive and easy to use language for the specification of prototypes of real-time systems. It encompasses the best features of previous structured languages. Its constructs for modelling data and control flow are particularly rich. It has been noted that STDs do not provide as many facilities for modelling complex reactive behaviour as do the Statecharts of STATEMATE. Currently the ESML transformation schema addresses the logical modelling of systems (the construction of behavioural models), and so can only be used to specify exploratory prototypes of systems.

The prototype specification effort requires the specification of complex mini-specifications in the target language, this may be a little premature at this stage of the

development. An executable formal language, such as VDM-SL, might be more appropriate for the definition of mini-specifications.

The translation templates are complex since they have LOOPN nets as the target language. The automation of the translation process is needed to fully exploit the prototyping system, the presence of LOOPN is then oblivious to the user.

The program which is output automatically from the LOOPN code generator can be compiled and run in the target environment. This is especially important for real-time systems, their prototypes need to execute at realistic speeds. As an intermediate step it might be better to animate the ESML specification initially before generating any code. Such an animation would show the flow of tokens (as defined by the execution rules) as the prototype is exercised, this would allow debugging of the prototype before it is exercised by the user in the field.

The format of the prototype output is currently unimpressive. The user requires more realistic output, i.e. connection of the prototype to a graphical interface (e.g. a graphical mock-up of the cockpit panel in the case of the APU Fuel Subsystem). Such an interface would allow the user to manipulate objects and view result (e.g. press switched and view resultant dial readings). This would aid the user in evaluating the prototype and proposing amendments.

7.4 Further Work

The following paragraphs consider various extensions of the ESML/LOOPN prototyping systems and further work in the whole area.

The most attainable extension of the ESML/LOOPN prototyping system would be one which incorporates an automatic translation process and which utilises extensions of the LOOPN tool which are being proposed. Figure 7.1 shows a possible future version of the ESML/LOOPN prototyping system after such a scenario. The future version has an automatic translation process which has a graphical specification of the resultant LOOPN net specification as its output. The XLOOPN tool [Lakos93a] is under development which will allow the graphical specification of LOOPN nets. The output from the translation process is in XLOOPN format. A set of Petri net analysis tools [Lakos93b] for LOOPN are also under construction which will allow formal model analysis. Formal analysis of LOOPN nets can be used to mathematically prove certain properties (such as absence from deadlock) of the prototype. The LOOPN code generator is also being modified to produce different target languages, such as OCCAM, which would allow the prototype to execute on a transputer network. This will allow the prototyping of distributed real-time systems in a natural environment. To generate prototypes to run on

parallel architectures the LOOPN tool needs to implement a distributed implementation strategy to allow the concurrent firing of transitions, such as the PPO strategy extended to LOOPN nets (section 4.4.3.1.2). At the moment the prototype generated by the LOOPN code generator interacts with the environment through simple print statements which report execution progress. In the future it will be possible to interface the prototype with a graphical mock-up of its environment, e.g. gauges, switches, dials etc., so as to provide a more realistic execution output.

The ESML/LOOPN prototyping system could also be expanded to allow the heterogeneous prototyping (section 2.3.4) of real-time systems. This would first require the definition of a new set of ESML execution rules which would concern Implementation Model abstraction levels such as Processor Environment Models (PEMs), Software Environment Models (SEMs), and possibly Code Organisation Models (COMs). These execution rules would have to consider the allocation of portions of the system model to different processors, how execution would proceed at the boundary between processors, and how execution within individual processors is determined by software architecture. The execution rules would also have to consider Structure Charts as used within the COM. Once a full set of execution rules have been defined, the ESML/LOOPN prototyping system could use LOOPN nets to provide a rigorous interpretation of them through a new set of translation templates. The prototyping system would then facilitate the construction of heterogeneous prototypes.

It would be also possible to replace LOOPN as the code generator component and use a different high-level Petri net language for defining the execution semantics for ESML. A possibility would be the use of TAPIOCA and CP-nets. The TAPIOCA code generator generates an OCCAM program to run on a transputer network, it supports the distributed implementation of CP-nets. The translation templates would have to be modified to produce CP-nets as output. The modifications would however be minimal since the various high-level net languages differ only in the inscription language used, the basic concepts are the same. The main advantage of TAPIOCA is the clever use of invariant analysis in its code generation process. The CPN code generator could be used if the user wished to run the prototype on a multiprocessor platform, indeed any of the code generators surveyed in Chapter 4 could be used to define new ESML/??? prototyping systems (e.g. ESML/TAPIOCA, ESML/CPN, ESML/PROTOB etc.). The choice would depend on the target language and architecture required.

An extension of the ESML language to allow the modelling of complex reactive behaviour would be beneficial. Such a facility is provided by the Statecharts language through its use of hierarchy, concurrency, and broadcast communication. Features such as

these could be imported into ESML to augment the current method of specifying control logic using simple STDs ESML could even use Statecharts in their entirety to replace STDs The ESML execution rules would have to be updated to cope with the improvements, the translation templates would then be modified to implement the execution rule changes

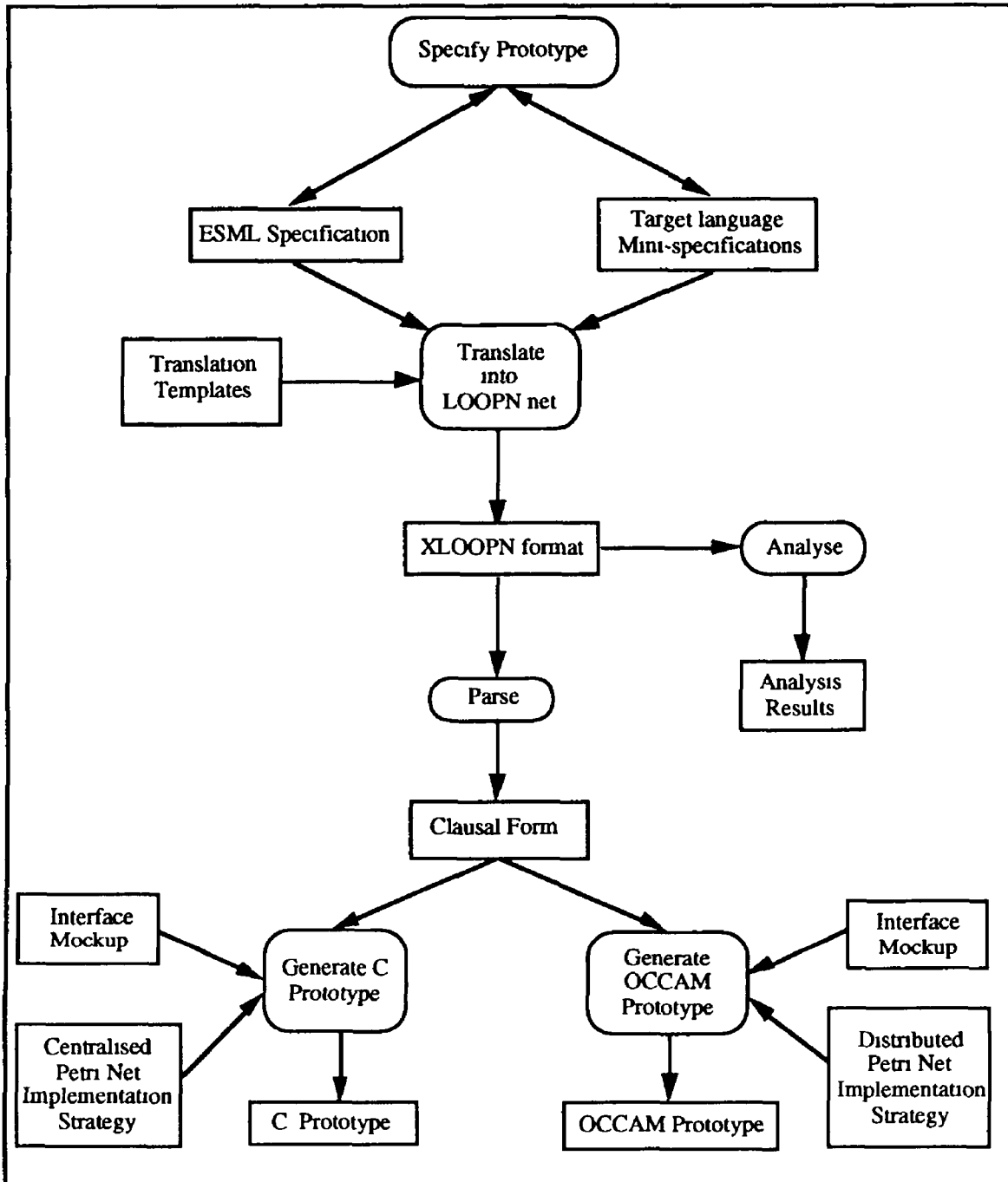


Figure 7 1 Future Version of the ESML/LOOPN Prototyping System

It would also be a useful exercise to examine how the translation templates could be arranged into some sort of object-oriented class hierarchy so as to fully exploit the object-oriented features provided by LOOPN, especially the definition of module subclasses with inheritance

7.5 Conclusion

The ESML/LOOPN prototyping system provides a simple framework for using ESML as a graphical executable specification language. In such a capacity, ESML can be used to build exploratory prototypes of real-time systems for use within a keep-it prototyping approach. ESML can therefore be used for prototyping real-time systems, an approach which is characterised by executable models at each stage of development and closer co-operation with users.

References

- [Agrawala92] A Agrawala and W Levi, Real-Time System Design, McGraw-Hill, 1992
- [Agresti86] W Agresti, "The Conventional Software Life-Cycle Model Its Evolution and Assumptions", New Paradigms for Software Development, IEEE Computer Society Press
- [Alavi84] M Alavi, "An Assessment of the Prototyping approach to Information Systems development", Communications of the ACM 27, 6, 1984
- [Auer88] A Auer, "Automated Code Generation of Embedded Real-time Systems", Microprocessing and Microprogramming 24, 1988
- [Baldassari91] M Baldassari and G Bruno, "PROTOB An Object Oriented Methodology for Developing Discrete Event Dynamic Systems", Computer Languages 16, 1, 1991
- [Battiston88] E Battiston, "OBJSA Nets A Class of High-level Nets having Objects as Domains", Advances in Petri Nets 1988, edited by G Rozenberg
- [Blumofe88] R Blumofe and A Hecht, "Executing Real-Time Structured Analysis Specifications", ACM Sigsoft, Software Engineering Notes, Vol 13, No 3
- [Boehm76] B Boehm, "Software Engineering Economics", Englewood Cliffs, Prentice Hall, 1976
- [Booch91] G Booch, Object-Oriented Design with Applications, Benjamin/Cummings Pub Co , 1991
- [Brackett87] J Brackett and E Reilly, "An Experimental System for Executing Real-Time Structured Analysis Models", Proceedings of the 12th Structured Methods Conference, August 1987, Chicago, USA

- [Breant90] F Breant, "TAPIOCA OCCAM Rapid Prototyping from Petri-net", 5th Jerusalem Conference on Information Technology, October 1990
- [Breant91] F Breant, "Rapid Prototyping from Petri-net on a Loosely Coupled Parallel Architecture", Proceedings of the 3rd International Conference on the Applications of Transputers, Glasgow, Scotland, 1991
- [Bruno86] G Bruno and G Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems", IEEE Transactions on Software Engineering, Vol SE-12, No 2, February 1986
- [Bruyn88] W Bruyn, R Jensen, D Keskar, P Ward, "ESML An Extended Systems Modelling Language based on the Data Flow Diagram", ACM SIGSOFT Software Engineering Notes, Vol 13, No 1, January 1988
- [Butler90] B Butler, R Esser, R Mattmann, "A Distributed Simulator for High Order Petri Nets", Advances in Petri Nets 1990, Springer-Verlag, edited by G Rozenberg
- [Chen76] P Chen, "The Entity-Relationship Model - Towards a Unified View of Data", ACM Transactions on Database Systems, Vol 1, No 1, March 1976
- [Coomber90] C Coomber and R Childs, "A Graphical Tool for the Prototyping of Real-Time Systems", in ACM SIGSOFT Software Engineering Notes, Vol 15, No 2, April 1990
- [Dahler87] J Dahler et al , "A Graphical Tool for the Design and Prototyping of Distributed Systems", ACM Software Engineering Notes, Vol 12, No 3, July 1987
- [DeMarco78] T De Marco, "Structured Analysis and System Specification", Yourdon Press, New York, 1978
- [[DiGiovanni91] R Di Giovanni, "HOOD nets", Advances in Petri Nets 1991, Springer-Verlag, edited by G Rozenberg,

[Elmstrom92] R Elmstrom, R Lintulampi, M Pezzi, "Giving Semantics to SA/RT by Means of High-Level Timed Petri Nets", Institute of Applied Computer Science (IFAD), Denmark, 1992

[Ferrentino77] A Ferrentino and H Mills, "State Machines and their semantics in Software Engineering", Proc IEEE COMPSAC 77

[Fleming88] P Fleming, Parallel Processing in control the Transputer and other architectures, IEE Control Engineering Series, 1988

[Floyd84] C Floyd, "A Systematic look at Prototyping", in Approaches to Prototyping, edited by R Budde et al , Springer-Verlag, 1984

[France92] R France, "Semantically Extended Data Flow Diagrams A Formal Specification Tool", IEEE Transactions on Software Engineering, Vol 18, No 4, April 1992

[Fuchs92] N Fuchs, "Specifications are (preferably) executable", Software Engineering Journal, September 1992

[Gane79] C Gane and T Sarson, Structured Systems Analysis and Design, Prentice Hall, Englewood Cliffs, NJ, USA, 1979

[Genrich81] H Genrich and K Lautenbach, "System Modelling with High-level Petri Nets", Theoretical Computer Science 13, 1981

[Harel87] D Harel, "Statecharts A Visual Formalism for Complex Systems", Science of Computer Programming, Vol 8, 1987

[Harel88] D Harel, "STATEMATE A Working Environment for the Development of Complex Reactive Systems", 10th International Conference on Software Engineering, Washington, USA, 1988

[Harel92] D Harel, "Biting the Silver Bullet Toward a Brighter Future for System Development", IEEE Computer, January 1992

- [Henderson86] P Henderson, "Functional Programming, Formal Specification, and Rapid Prototyping", IEEE Transactions on Software Engineering, Vol Se-12, No 2, February 1986
- [Hekmatpour88] S Hekmatpour and D Ince, "Software Prototyping, Formal Methods and VDM", Addison-Wesley, International Computer Science Series
- [Hartung88] G Hartung, "Programming a closely coupled Multiprocessor system with high-level Petri Nets", Advances in Petri Nets 1988, Springer-Verlag, edited by G Rozenberg
- [Hatley87] D Hatley and I Pirbhai, "Strategies for Real-Time System Specification", Dorset House Publishing, New York, 1987
- [Hayes89] I Hayes and C Jones, "Specifications are not (necessarily) executable", Software Engineering Journal, November 1989
- [Hughes89] T Hughes and J Cooling, "The Emergence of Rapid Prototyping as a Real-Time Software Development Tool", IEE Colloquium on the Specification of Complex Systems, London, 1989
- [INMOSS88] Transputer Reference Manual, INMOS, Prentice-Hall, 1988
- [1-Logix91] "The Languages of STATEMATE", Technical Report, 1-Logix Inc , Burlington, MA, 1991
- [Jensen81] K Jensen, "Coloured Petri-Nets and the Invariant Method", Theoretical Computer Science 14, 1981
- [Jensen81] K Jensen, "Coloured Petri-Nets and the Invariant Method", Theoretical Computer Science 14, 1981
- [Jensen90] K Jensen, "Coloured Petri Nets A High Level Language for System Design and Analysis", Advances in Petri Nets 1990, edited by G Rozenberg

- [Jensen92] K Jensen, "Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use", EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992
- [Kordon90] F Kordon, P Estrailhier, R Card, "Rapid Ada Prototyping Principles and example of a complex application", 9th Annual International Phoenix Conference on Computers and Communications, Scottsdale, USA, March 1990
- [Lai91] R Lai and D O'Connor, "Automatic Implementation of Communication Protocols based on Petri Nets", Proceedings of the 1991 Singapore International Conference on Networks
- [Laplante92] P Laplante, "Real-Time System Design and Analysis - an Engineer's Handbook", IEEE Press, 1992
- [Lakos90] C Lakos and C Keen, "Simulation with Object-Oriented Petri Nets", Australian Software Engineering Conference, Sydney, 1991
- [Lakos92] C Lakos et al , "A Flexible Distributed simulator for Object-Oriented Petri Nets", Transputer and Parallel Applications Conference, Melbourne, Australia, 1992
- [Lakos93a] C Lakos, "LOOPN System Manual", Department of Computer Science, University of Tasmania, Hobart, Australia
- [Lakos93b] C Lakos and C Keen, "Applying Invariant Analysis to Modular Petri Nets", Australian Computer Science Conference, 1993, Brisbane
- [Ledru90] Y Ledru, "Hierarchical Specification of Reactive System a Case Study", COMPEURO '90, Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering
- [Lee92] P Lee and K Tan, "Modelling of Visualised Data-Flow Diagrams using Petri Net model", Software Engineering Journal, January 1992
- [Marca88] D Marca and C McGowan, SADT, McGraw-Hill, New York, 1988

- [Martin93] D Martin, "Introducing the Formal Specification Language LOTOS into Structured Analysis based Development", M Sc Thesis, School of Computer Applications, Dublin City University, 1993
- [Mortensen90] B Mortensen, "Incremental Prototyping of Embedded Real-Time Systems (IPTES), part 1 project summary and consortium information", Esprit II project proposal, Odense, Denmark, Institute of Applied Computer Science (IFAD), 1990
- [Munemori88] J Munemori et al , "An Extension of SDL for Executable Specification of Communication Systems based upon Petri nets", ISIIS 88, Proceedings of the 2nd International Symposium on Interoperable Information Systems, Tokyo, Japan, 1988
- [Murata89] T Murata, "Petri Nets Properties, Analysis and Applications", Proceedings of the IEEE, Vol 77, No 4, April 1989
- [Parker90] K Parker, "The PROMPT Automatic Implementation Tool - Initial Impressions", 2rd International Conference on Formal Description Techniques, FORTE 90, Madrid, Spain
- [Peterson81] J Peterson, "Petri Net Theory and the Modelling of Systems", Prentice Hall
- [Petri66] C A Petri, "Communication with Automata", New York Griffis Air Force Base Technical Report, RADC-TR-66-377, 1966
- [Pinci91] V Pinci and R Shapiro, "An Integrated Software Development Methodology Based on Hierarchical Coloured Petri nets", in Advances in Petri Nets 1991, edited by G Rozenberg, Springer-Verlag
- [Pulli86a] P Pulli, "Execution of Structured Analysis Specifications with an Object Oriented Petri Net Approach", Proceedings of the 1986 International Conference on Computer Languages, Miami, USA
- [Pulli86b] P Pulli, "Execution of Structured Analysis Specifications with an Object-Oriented Petri Net Approach", P Pulli, Proceedings of the 1986 International Conference on Computer Languages, Miami, USA

- [Reisig91] W Reisig, "Petri Nets and Algebraic Specifications", Theoretical Computer Science 80, 1991
- [Ross77] D Ross, "Structured Analysis A Language for Communicating Ideas", IEEE Transactions on Software Engineering, Vol 3, No 1, January 1977
- [Royce70] W Royce, "Managing the development of large software systems Concepts and techniques", WESCON, August 1970
- [Sacha91] K Sacha, "Transnet A method for Transformational Development of Embedded Software", Microprocessing and Microprogramming, Vol 32, 1991
- [Sacha92] K Sacha, "Transformational Implementation of PAISLey using Petri nets", Software Engineering Journal, May 1992
- [Shapiro90] R Shapiro et al , "Modelling a NORAD Command Post using SADT and Coloured Petri Nets", Proceedings of the IDEF Users Group, Washington DC, May 1990
- [Silva86] M Silva et al , "On the Software Implementation of Petri Nets and Coloured Petri Nets using high-level concurrent languages", 7th European Workshop on the Application and Theory of Petri Nets, Oxford, England, 1986
- [Silva89] M Silva, "Petri Nets and Flexible Manufacturing", Advances in Petri Nets 1989, Springer-Verlag, edited by G Rozenberg
- [Taubner88] D Taubner, "On the Implementation of Petri Nets", Advances in Petri Nets 1988, Springer-Verlag, edited by G Rozenberg
- [Tsaï89] J Tsaï and T Weigert, "Exploratory Prototyping through the use of Frames and Production Rules", Proceedings of the 13th International Computer Software and Applications Conference
- [Tse88] T Tse and L Pong, "An Application of Petri nets in Structured Analysis", ACM SIGSOFT Software Engineering Notes, Vol 11, No 5, October 1988

- [Valette91] V Valette, "Software Implementation of Petri Nets and Compilation of Rule-based Systems", Advances in Petri Nets 1991, Springer-Verlag, edited by G Rozenberg
- [Vonk90] R Vonk, "Prototyping The effective use of CASE technology", Prentice Hall
- [Ward85a] P Ward and S Mellor, Structured Development for Real-Time Systems, Volume 1 Introduction and Tools, Prentice Hall, NJ, USA, 1985
- [Ward85b] P Ward and S Mellor, Structured Development for Real-Time Systems, Volume 2 Essential Modelling Techniques, Prentice Hall, NJ, USA, 1985
- [Ward86a] P Ward and S Mellor, Structured Development for Real-Time Systems, Volume 3 Implementation Modelling Techniques, Prentice Hall, NJ, USA, 1986
- [Ward86b] P Ward, "The Transformation Schema An Extension of the Data Flow Diagram to Represent Control and Timing", IEEE Transactions on Software Engineering, Vol SE-12, No 2, February 1986
- [Ward89] P Ward, "How to Integrate Object Orientation with Structured Analysis and Design", IEEE Software, March 1989
- [Winokur89] W Winokur and J Levi, "ECSAM A Method for the Analysis of Complex Embedded Computer Systems and their Software", Proceedings of the 5th Structured Techniques Association Conference, Chicago, May 1989
- [Winokur90] W Winokur et al , "Requirements Analysis and Specification of Embedded Systems with ECSAM - a case study", COMPEURO '90, Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering
- [Yourdon89] E Yourdon, Modern Structured Analysis, Prentice Hall, Englewood Cliffs, NJ, USA, 1989

Appendix A

LOOPN Source Program Code for the APU Fuel Subsystem Prototype

```
{////////////////////////////////////}
{ Types Defintion Module (types l) }
{////////////////////////////////////}

MODULE types,

TYPE
  { enable, disable, suspend, resume, trigger }
  prompts = ( E, D, S, R, T ),

  prompttype = TOKEN null WITH
    pr prompts,
  end,

  defaulttype = TOKEN null WITH      { token defined or not }
    undefined boolean,
  end,

  fueltype = TOKEN defaulttype WITH  { fuel token type }
    filtered boolean,
    temperature real,
    pressure real,
  end,

  airtype = TOKEN defaulttype WITH   { air token type }
    temperature real,
    pressure real,
  end,

TRANSITION initial,

END MODULE

{////////////////////////////////////}
{ Fuel Control Valve (fcv l) }
{ Control Transformation, no input prompt, 2 output activations, }
{ 2 input signals, 2 output signals }
{////////////////////////////////////}

MODULE fcv = types( INOUT APU_On, APU_Off2, Open_door, Close_door null,
  INOUT A1, A2 prompttype ),

TYPE states = ( DEACT, CLOSED, OPEN ), { valve is closed or open }
```

```

stateype = TOKEN null WITH      { inherit from null }
  st states,
end,

PLACE Init null,                { S-elements }
  State stateype,

TRANSITION initial,            { setup initial marking }
  OUTPUT
  Init <- t0,
  State <- s0 = [ st DEACT ], { deactivated initially }

  ACTION
  WriteString("Initial transition fired - fuel control valve",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

TRANSITION Init_t,            { valve is initially closed }
  INPUT
  t <- Init,
  sin <- State | sin st = DEACT,

  OUTPUT
  State <- sout = [ st CLOSED ], { valve closed }

  ACTION
  WriteString("Init_t fired - fuel control valve",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

TRANSITION T1,                { valve becomes open }
  INPUT
  d <- APU_On,                { Apu On switch pressed }
  sin <- State | sin st = CLOSED,

  OUTPUT
  Open_door <- d,             { signal door to open }
  A1 <- p1 = [ pr E ],        { enable boost pump control }
  A2 <- p2 = [ pr E ],        { enable heat control }
  State <- sout = [ st OPEN ],

  ACTION
  WriteString("T1 fired, valve opens - fuel control valve",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

TRANSITION T2,                { valve becomes open }
  INPUT
  d <- APU_Off2,              { Apu Off switch pressed }

```

```
sin <- State | sin st = OPEN,
```

OUTPUT

```
Close_door <- d,      { signal door to close }  
A1 <- p1 = [ pr D ], { disable boost pump control }  
A2 <- p2 = [ pr D ], { disable heat control }  
State <- sout = [ st CLOSED ],
```

ACTION

```
WriteString("T2 fired, valve closes - fuel control valve",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

```
TRANSITION Eat1,      { remove APU_On if wrong state }
```

INPUT

```
d <- APU_On,          { Apu On switch pressed }  
s <- State | s st <> CLOSED, { not closed }
```

OUTPUT

```
State <- s,
```

ACTION

```
WriteString("Eat1 fired, APU_On eaten - fuel control valve",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

```
TRANSITION Eat2,      { remove APU_Off2 if wrong state }
```

INPUT

```
d <- APU_Off2,        { Apu Off switch pressed }  
s <- State | s st <> OPEN, { not open }
```

OUTPUT

```
State <- s,
```

ACTION

```
WriteString("Eat2 fired, APU_Off2 eaten - fuel control valve",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

```
END MODULE
```

```
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
{ Boost Pump Control (bpc 1) }
{ Control Transformation, 1 input Activation prompt, 2 output }
{ Activation prompts, 2 input signals }
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
```

```
MODULE bpc = types( INOUT Tank1_off, Rpm_95 null,  
                  INOUT A1, A3, A4 prompttype ),
```

```
TYPE states = ( DEACT, OFF, ON ),
```

```

statetype = TOKEN null WITH      { inherit from null }
  st states,
end,

```

```

PLACE State statetype,          { S-element }

```

```

TRANSITION initial,            { setup initial marking }
  OUTPUT
  State <- s0 = [ st DEACT ], { deactivated }

```

```

ACTION
  WriteString("Initial Transition fired - boost pump control",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

```

```

TRANSITION Disable,           { disable self }
  INPUT
  p1 <- A1 | p1 pr = D,       { from fuel control valve }
  sin <- State,

```

```

OUTPUT
  A3 <- p2 = [ pr D ],      { disable pump bypass }
  A4 <- p3 = [ pr D ],      { disable pump }
  State <- sout = [ st DEACT ], { deactivated }

```

```

ACTION
  WriteString("Disable fired - boost pump control",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

```

```

TRANSITION Enable,           { enable self }
  INPUT
  p1 <- A1 | p1 pr = E,       { from fuel control valve }
  sin <- State | sin st = DEACT,

```

```

OUTPUT
  A3 <- p2 = [ pr E ],      { enable pump bypass }
  A4 <- p3 = [ pr D ],      { disable pump }
  State <- sout = [ st OFF ], { off is initial State }

```

```

ACTION
  WriteString("Enable fired - boost pump control",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

```

```

TRANSITION Eat1,              { superflous E received }
  INPUT
  p <- A1 | p pr = E,         { from fuel control valve }

```

```

s <- State | s st <> DEACT,

OUTPUT
State <- s,

ACTION
WriteString("Eat1 fired, enable signal eaten - boost pump control",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION T1,                { turn on pump if tank no 1 is off }
INPUT
d <- Tank1_off,                { fuel tank no 1 is off }
sin <- State | sin st = OFF,    { pump off }

OUTPUT
A3 <- p1 = [ pr D ],           { disable pump bypass }
A4 <- p2 = [ pr E ],           { enable pump }
State <- sout = [ st ON ],      { pump on }

ACTION
WriteString("T1 fired, pump goes on - boost pump control",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION T2,                { turn off pump if rpm_95 reached }
INPUT
d <- Rpm_95,                   { 95% engine rpm }
sin <- State | sin st = ON,     { pump on }

OUTPUT
A3 <- p1 = [ pr E ],           { enable pump bypass }
A4 <- p2 = [ pr D ],           { disable pump }
State <- sout = [ st OFF ],     { pump off }

ACTION
WriteString("T2 fires, pump goes off - boost pump control",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Eat2,              { eat Tank1_off if wrong state }
INPUT
d <- Tank1_off,
s <- State | s st <> OFF,      { pump not off }

OUTPUT
State <- s,

ACTION
WriteString("Eat2 fired, Tank1_off eaten - boost pump control",0),
WriteString(", t =",0),

```



```

WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Eat3,          { eat Rpm_95 if wrong state }
INPUT
  d <- Rpm_95,
  s <- State | s st <= ON,  { pump not on }

OUTPUT
  State <- s,

ACTION
  WriteString("Eat3 fired, Rpm_95 eaten - boost pump control",0),
  WriteString(" t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

END MODULE

```

```

{//////////////////////////////////////////////////////////////////}
{ Fuel Heater Control (fhc 1) }
{   Control Transformation, 1 input Activation prompt, 2 output }
{   Activation prompts, 1 continuous input }
{//////////////////////////////////////////////////////////////////}

```

```

MODULE fhc = types( INOUT Fuel_B    fueltype,
                   INOUT A2, A5, A6 prompttype ),

```

```

TYPE states = ( DEACT, OFF, ON ),

```

```

  statetype = TOKEN null WITH    { inherit from null }
  st states,
end,

```

```

PLACE State  statetype,          { S-element }

```

```

TRANSITION initial,          { setup initial marking }

```

```

  OUTPUT
  State <- s0 = [ st DEACT ], { deactivated initially }

```

```

  ACTION
  WriteString("Initial Transition fired - fuel heater control",0),
  WriteString(" t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

```

```

TRANSITION Disable,          { disable self }

```

```

  INPUT
  p1 <- A2 | p1 pr = D,      { from fuel control valve }
  sin <- State,

```

```

  OUTPUT

```

```

A5 <- p2 = [ pr D ], { disable heat bypass }
A6 <- p3 = [ pr D ], { disable heat }
State <- sout = [ st DEACT ], { deactivated }

```

ACTION

```

WriteString("Disable fired - fuel heater control",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

TRANSITION Enable, { enable self }

INPUT

```

p1 <- A2 | p1 pr = E, { from fuel control valve }
sin <- State | sin st = DEACT,

```

OUTPUT

```

A5 <- p2 = [ pr E ], { enable heat bypass }
A6 <- p3 = [ pr D ], { disable heat }
State <- sout = [ st OFF ], { off is initial state }

```

ACTION

```

WriteString("Enable fired - fuel heater control",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

TRANSITION Eat1, { already enabled }

INPUT

```

p <- A2 | p pr = E, { from fuel control valve }
s <- State | s st <> DEACT,

```

OUTPUT

```

State <- s,

```

ACTION

```

WriteString("Eat1 fired, Enable eaten - fuel heater control",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

TRANSITION T1, { turn on heat if temp <= 37 }

INPUT

```

f <- Fuel_B | ( f undefined = false and f temperature <= 37 ),
sin <- State | sin st = OFF, { heat off }

```

OUTPUT

```

Fuel_B <- f,
A5 <- p1 = [ pr D ], { disable heat bypass }
A6 <- p2 = [ pr E ], { enable heat }
State <- sout = [ st ON ], { heat on }

```

ACTION

```

WriteString("T1 fired, Heater goes on - fuel heater control",0),
WriteString(", t =",0),

```

```

WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION T2,                { turn off heat if temp >= 64 }
INPUT
  f <- Fuel_B | ( f undefined = false and f temperature >= 64 ),
  sin <- State | sin st = ON, { heat on }

OUTPUT
  Fuel_B <- f,
  A5 <- p1 = [ pr E ], { enable heat bypass }
  A6 <- p2 = [ pr D ], { disable heat }
  State <- sout = [ st OFF ], { heat off }

ACTION
  WriteString("T2 fired, Heater goes off - fuel heater control",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

END MODULE

{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
{ Fuel Solenoid Valve (fsv l) }
{ Control Transformation, no input prompt, 1 output Activation }
{ prompt, 2 input signals }
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}

MODULE fsv = types( INOUT Seq_oil_closed, APU_Off1 null,
                   INOUT A7 promptype ),

TYPE states = ( DEACT, CLOSED, OPEN ), { valve is closed or open }

  statetype = TOKEN null WITH { inherit from null }
  st states,
  end,

PLACE Init null, { S-elements }
  State statetype,

TRANSITION initial, { setup initial marking }
OUTPUT
  Init <- t0,
  State <- s0 = [ st DEACT ], { deactivated initially }

ACTION
  WriteString("Initial transition fired - fuel solenoid valve",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

TRANSITION Init_t, { setup initial state }
INPUT

```

```

t <- Init,
sin <- State | sin st = DEACT,

OUTPUT
State <- sout = [ st CLOSED ],           { closed is initial state }

ACTION
WriteString("Init_t fired - fuel solenoid valve",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION T1,                           { valve becomes open }
INPUT
d <- Seq_oil_closed,                       { sequencing oil pres switch }
sin <- State | sin st = CLOSED,

OUTPUT
A7 <- p = [ pr E ],                         { enable fcU }
State <- sout = [ st OPEN ],

ACTION
WriteString("T1 fired, valve opens - fuel solenoid valve",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION T2,                           { valve becomes closed }
INPUT
d <- APU_Off1,                             { Apu Off switch pressed }
sin <- State | sin st = OPEN,

OUTPUT
A7 <- p = [ pr D ],   { disable fcU }
State <- sout = [ st CLOSED ],

ACTION
WriteString("T2 fired, valve closes, fuel solenoid valve",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Eat1,                          { eat Seq_oil_closed if wrong state }
INPUT
d <- Seq_oil_closed,                       { pressure switch closed }
s <- State | s st <> CLOSED,                { not closed }

OUTPUT
State <- s,

ACTION
WriteString("Eat1 fired, Seq_oil_closed eaten - fuel solenoid valve",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),

```

```

WriteLn(),

TRANSITION Eat2,                                { remove APU_Off1 if wrong state }
INPUT
  d <- APU_Off1,                                { Apu Off switch pressed }
  s <- State | s st <> OPEN,                     { not open }

OUTPUT
  State <- s,

ACTION
  WriteString("Eat2 fired, Apu_Off1 eaten - fuel solenoid valve",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

END MODULE

{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
{ Pump (pump 1) }
{ Primitive Flow Transformation, 1 continuous input, 1 continuous }
{ output, 1 input activation prompt }
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}

MODULE pump = types( INOUT Fuel_1, Fuel_A fueltype,
                    INOUT A4 prompttype ),

TYPE statuses = ( DEACT, ACT ),                 { visualization of transformation }

  statustype = TOKEN null WITH                  { inherit from null }
  st statuses,
  end,

PLACE Status      statustype,                  { S-elements }
      F1_Old      fueltype,

TRANSITION initial,                            { setup initial marking }
OUTPUT
  Status <- s0 = [ st DEACT ],

      { dummy token }

  F1_Old <- f0 = [ filtered false,
                  temperature 0,
                  pressure 0,
                  undefined true ],

ACTION
  WriteString("Initial Transition fired - pump",0),
  WriteString(", t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

TRANSITION Pump_f,                             { pump fuel, increase pressure }

```

INPUT

```
{ input defined }
f1 <- Fuel_1 | f1 undefined = false,

{ input changed }
f1old <- F1_Old | f1old filtered <> f1 filtered OR
           f1old pressure <> f1 filtered OR
           f1old temperature <> f1 temperature OR
           f1old undefined <> f1 undefined,

fA <- Fuel_A,
s <- Status | s st = ACT,           { active }
```

OUTPUT

```
Fuel_1 <- f1,           { return input }
F1_Old <- f1,           { copy old f1 to F1_Old }

Status <- s,

Fuel_A <- fAnew = [ filtered      f1 filtered,
                  pressure      (f1 pressure + 10),
                  temperature    f1 temperature,
                  undefined      false ],
           { fuel pressure is increased 10 by transition }
```

ACTION

```
WriteString("Pump_f fired - pump",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),
```

```
TRANSITION Enable,           { enable self }
```

INPUT

```
p <- A4 | p pr = E,           { from boost pump control }
sin <- Status,
```

OUTPUT

```
Status <- sout = [ st ACT ], { active }
```

ACTION

```
WriteString("Enable fired - pump",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),
```

```
TRANSITION Disable,         { disable self }
```

INPUT

```
p <- A4 | p pr = D,           { disable from boost pump control }
sin <- Status,
f1old <- F1_Old,
fA <- Fuel_A,
```

OUTPUT

```

Status <- sout = [ st DEACT ],

{ undefine output }
Fuel_A <- fAnew = [ temperature fA temperature,
                  pressure      fA pressure,
                  filtered       fA filtered,
                  undefined      true ],

F1_Old <- f1new = [ temperature f1old temperature,
                  pressure       f1old pressure,
                  filtered        f1old filtered,
                  undefined       true ],

```

```

ACTION
WriteString("Disable fired - pump",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

END MODULE

```

{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
{ Pump Bypass (pump_b 1) }
{ Primitive Flow Transformation, 1 continuous input, 1 continuous }
{ Primitive Flow Transformation, 1 continuous output, 1 input }
{ Activation prompt, 1 continuous output }
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}

```

```

MODULE pump_b = types( INOUT Fuel_1, Fuel_A fueltype,
                     INOUT A3 prompttype ),

```

```

TYPE statuses = ( DEACT, ACT ),

```

```

    statustype = TOKEN null WITH { inherit from null }
    st statuses,
end,

```

```

PLACE Status statustype, { S-elements }
      F1_Old fueltype,

```

```

TRANSITION initial, { setup initial marking }
OUTPUT
Status <- s0 = [ st DEACT ],

```

```

      F1_Old <- f0 = [ filtered false, { dummy token }
                    temperature 0,
                    pressure 0,
                    undefined true ],

```

```

ACTION
WriteString("Initial Transition fired - pump bypass",0),
WriteString(", t =",0),

```

```

WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Bypass_f,                                { bypass fuel }
INPUT

{ input defined and changed }
f1 <- Fuel_1 | f1 undefined = false,
f1old <- F1_Old | f1old filtered <> f1 filtered OR
      f1old temperature <> f1 temperature OR
      f1old pressure <> f1 pressure OR
      f1old undefined <> f1 undefined,

fA <- Fuel_A,
s <- Status | s st = ACT,                            { active }

OUTPUT
Fuel_1 <- f1,                                         { return input }
F1_Old <- f1,                                         { copy old f1 to F1_Old }
Status <- s,
Fuel_A <- f1,                                         { copy input to output }

ACTION
WriteString("Bypass_f fired - pump bypass",0),
WriteString(" t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Enable,                                  { enable self }
INPUT
p <- A3 | p pr = E,                                   { from boost pump control }
sin <- Status,

OUTPUT
Status <- sout = [ st ACT ], { active }

ACTION
WriteString("Enable fired - pump bypass",0),
WriteString(" t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Disable,                                 { disable self }
INPUT
p <- A3 | p pr = D,                                   { disable from boost pump control }
sin <- Status,
f1old <- F1_Old,
fA <- Fuel_A,

OUTPUT
Status <- sout = [ st DEACT ],

{ undefine output }
Fuel_A <- fAnew = [ temperature fA temperature,

```



```

        pressure  fA pressure,
        filtered  fA filtered,
        undefined true ],

```

```

F1_Old  <- f1new = [ temperature f1old temperature,
                    pressure  f1old pressure,
                    filtered  f1old filtered,
                    undefined  true ],

```

ACTION

```

WriteString("Disable fired - pump bypass",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

END MODULE

```

{//////////////////////////////////////////////////////////////////}
{ Filter (filter 1) }
{ Primitive Flow Transformation, 1 continuous input, 1 continuous }
{ output, no input prompts }
{//////////////////////////////////////////////////////////////////}

```

```

MODULE filter = types( INOUT Fuel_A, Fuel_B  fueltype ),

```

```

PLACE Status  null, { S-elements }
      Fuel_A_Old fueltype,

```

```

TRANSITION initial, { setup initial marking }
OUTPUT
      Status  <- s0,

```

```

{ dummy token }
      Fuel_A_Old <- f0 = [ filtered false,
                          temperature 0,
                          pressure 0,
                          undefined true ],

```

ACTION

```

WriteString("Initial Transition fired - filter",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

```

TRANSITION Filter_f, { filter fuel }
INPUT

```

```

{ continuous input defined and changed }
      fA  <- Fuel_A  | fA undefined = false,
      fAold <- Fuel_A_Old | fAold filtered <> fA filtered OR
                          fAold temperature <> fA temperature OR
                          fAold pressure <> fA pressure OR
                          fAold undefined <> fA undefined,

```

```

      fB  <- Fuel_B,

```

```
s <- Status,
```

OUTPUT

```
Fuel_A <- fA, { return continuous input }  
Fuel_A_Old <- fA, { copy old fA to Fuel_A_Old }
```

```
Status <- s,
```

```
{ set filtered = true }
```

```
Fuel_B <- fBnew = [ filtered true,  
                  pressure fA pressure,  
                  temperature fA temperature,  
                  undefined false ],
```

ACTION

```
WriteString("Filter_f fired - filter",0),  
WriteString(" t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

END MODULE

```
{//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}  
{ Heat (heat 1) }  
{ Non-Primitive Flow Transformation, 2 continuous inputs, }  
{ 2 continuous outputs, 1 input Activation prompt }  
{//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
```

```
MODULE heat = types( INOUT Fuel_B, Fuel_C fueltype,  
                   INOUT Hot_bleed_air, Air airtype,  
                   INOUT A6 prompttype ),
```

```
TYPE statuses = ( DEACT, ACT ),
```

```
    statustype = TOKEN null WITH { inherit from null }  
    st statuses,  
end,
```

```
PLACE Status statustype, { S-elements }  
    FB_Old fueltype,  
    Hb_Old airtype,
```

```
TRANSITION initial, { setup initial marking }
```

OUTPUT

```
Status <- s0 = [ st DEACT ],
```

```
{ dummy tokens }
```

```
FB_Old <- f0 = [ filtered false,  
               temperature 0,  
               pressure 0,  
               undefined true ],
```

```
Hb_Old <- a0 = [ temperature 0,
```

```
pressure 0,  
undefined true ],
```

ACTION

```
WriteString("Initial Transition fired - heat",0),  
WriteString(" t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

```
TRANSITION Heat_f, { heat fuel }  
INPUT
```

```
{ continuous input must be defined }
```

```
fB <- Fuel_B | fB undefined = false,  
hb <- Hot_bleed_air | hb undefined = false,
```

```
{ continuous outputs }
```

```
a <- Air,  
fC <- Fuel_C,
```

```
{ only fire if continuous inputs have changed }
```

```
hbold <- Hb_Old,  
fBold <- FB_Old | ( (fB filtered <> fBold filtered OR  
fB temperature <> fBold temperature OR  
fB pressure <> fBold pressure OR  
fB undefined <> fBold undefined) OR  
(hb temperature <> hbold temperature OR  
hb pressure <> hbold pressure OR  
hb undefined <> hbold undefined) ),
```

```
{ must be active }
```

```
s <- Status | s st = ACT,
```

OUTPUT

```
{ continuous inputs }
```

```
Fuel_B <- fB,  
Hot_bleed_air <- hb,
```

```
{ continuous outputs are defined }
```

```
Air <- anew = [ temperature a temperature,  
pressure a pressure,  
undefined false ],
```

```
Fuel_C <- fCnew = [ temperature fC temperature,  
pressure fC pressure,  
filtered fC filtered,  
undefined false ],
```

```
{ store old values of continuous inputs }
```

```
Hb_Old <- hb,  
FB_Old <- fB,
```

```
{ status still active }
```

```
Status <- s;
```

ACTION

```
WriteString("Heat_f fired - heat",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

TRANSITION Enable,

{ enable self }

INPUT

```
p <- A6 | p pr = E,  
sin <- Status,
```

{ from heat_control }

OUTPUT

```
Status <- sout = [ st ACT ], { active }
```

ACTION

```
WriteString("Enable fired - heat",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

TRANSITION Disable,

{ disable self }

INPUT

```
p <- A6 | p pr = D,  
sin <- Status,
```

{ disable from heat_control }

{ continuous outputs }

```
a <- Air,  
fC <- Fuel_C,
```

```
hbold <- Hb_Old,  
fBold <- FB_Old,
```

OUTPUT

```
Status <- sout = [ st DEACT ],
```

{ continuous outputs become undefined }

```
Fuel_C <- fCnew = [ temperature fC temperature,  
pressure fC pressure,  
filtered fC filtered,  
undefined true ],
```

```
Air <- anew = [ temperature a temperature,  
pressure a pressure,  
undefined true ],
```

{ undefine remembered inputs }

```
Hb_Old <- hbnew = [ temperature hbold temperature,  
pressure hbold pressure,  
undefined true ],
```

```
FB_Old <- fBnew = [ temperature fBold temperature,  
pressure fBold pressure,  
filtered fBold filtered,
```

```
undefined true ],
```

```
ACTION
```

```
WriteString("Disable fired - heat",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

```
END MODULE
```

```
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
{ Heat Bypass (heat_b l) }
{ Primitive Flow Transformation, 1 continuous input, 1 continuous }
{ output, 1 input Activation prompt }
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
```

```
MODULE heat_b = types( INOUT Fuel_B, Fuel_C fueltype,  
INOUT A5 prompttype ),
```

```
TYPE statuses = ( DEACT, ACT ),
```

```
statustype = TOKEN null WITH { inherit from null }  
st statuses,  
end,
```

```
PLACE Status statustype, { S-elements }  
FB_Old fueltype,
```

```
TRANSITION initial, { setup initial marking }  
OUTPUT  
Status <- s0 = [ st DEACT ],
```

```
{ dummy token }  
FB_Old <- f0 = [ filtered false,  
temperature 0,  
pressure 0,  
undefined true ],
```

```
ACTION
```

```
WriteString("Initial Transition fired - heat bypass",0),  
WriteString(", t =",0),  
WriteReal(globaltime(),5,2),  
WriteLn(),
```

```
TRANSITION Bypass_f, { bypass fuel }  
INPUT
```

```
{ continuous input must be defined and have changed }  
fB <- Fuel_B | fB undefined = false,  
fBold <- FB_Old | fBold filtered <> fB filtered OR  
fBold temperature <> fB temperature OR  
fBold pressure <> fB pressure OR  
fBold undefined <> fB undcfined
```

```
fC <- Fuel_C,
s <- Status | s st = ACT, { active }
```

OUTPUT

```
Fuel_B <- fB, { return input }
FB_Old <- fB, { copy old fB to FB_Old }
Status <- s,
Fuel_C <- fB, { copy input to output }
```

ACTION

```
WriteString("Bypass_f fired - heat bypass",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),
```

```
TRANSITION Enable, { enable self }
```

INPUT

```
p <- A5 | p pr = E, { from heat control }
sin <- Status,
```

OUTPUT

```
Status <- sout = [ st ACT ], { active }
```

ACTION

```
WriteString("Enable fired - heat bypass",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),
```

```
TRANSITION Disable, { disable self }
```

INPUT

```
p <- A5 | p pr = D, { disable from heat control }
sin <- Status,
fBold <- FB_Old,
fC <- Fuel_C,
```

OUTPUT

```
Status <- sout = [ st DEACT ],
```

```
{ continuous output becomes undefined }
```

```
Fuel_C <- fCnew = [ temperature fC temperature,
                  pressure fC pressure,
                  filtered fC filtered,
                  undefined true ],
```

```
{ undefine remembered input }
```

```
FB_Old <- fBnew = [ temperature fBold temperature,
                  pressure fBold pressure,
                  filtered fBold filtered,
                  undefined true ],
```

ACTION

```

WriteString("Disable fired - heat bypass",0),
WriteString(" t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

END MODULE

```

{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}
{ Fuel Control Unit (fcu l) }
{ Non-Primitive Flow Transformation, 2 continous inputs, }
{ 1 continous output, 1 input Activation prompt }
{////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////}

```

```

MODULE fcu = types( INOUT Fuel_C, Fuel_3 fueltype,
                   INOUT Control_ar_3 airtype,
                   INOUT A7 prompttype ),

```

```

TYPE statuses = ( DEACT, ACT ),

```

```

    statustype = TOKEN null WITH { inherit from null }
    st statuses,
end,

```

```

PLACE Status statustype, { S-elements }
      FC_Old fueltype,
      Ca_Old airtype,

```

```

TRANSITION initial, { setup initial marking }
  OUTPUT
  Status <- s0 = [ st DEACT ],

```

```

{ dummy tokens }
FC_Old <- f0 = [ filtered false,
                temperature 0,
                pressure 0,
                undefined true ],

Ca_Old <- a0 = [ temperature 0,
                pressure 0,
                undefined true ],

```

```

ACTION
  WriteString("Initial Transition fired - fuel control unit",0),
  WriteString(" t =",0),
  WriteReal(globaltime(),5,2),
  WriteLn(),

```

```

TRANSITION Fcu_f, { fuel control unit }
  INPUT

```

```

{ contionuos input must be defined and have changed }
fc <- Fuel_C | fc undefined = false,
ca <- Control_ar_3 | ca undefined = false,

```

```

caold <- Ca_Old,

fCold <- FC_Old | ( (fCold filtered <> fC filtered OR
                    fCold temperature <> fC temperature OR
                    fCold pressure <> fC pressure OR
                    fCold undefined <> fC undefined) OR
                    (caold temperature <> ca temperature OR
                    caold pressure <> ca pressure OR
                    caold undefined <> ca undefined) ),

{ continuous output }
f3 <- Fuel_3,

s <- Status | s st = ACT,           { active }

OUTPUT
Fuel_C <- fC,                       { return input }
Control_ar_3 <- ca,

{ remember old input values }
FC_Old <- fC,
Ca_Old <- ca,

{ define output based on input }
Fuel_3 <- fout = [ temperature f3 temperature,
                  pressure f3 pressure,
                  filtered f3 filtered,
                  undefined false ],

Status <- s,

ACTION
WriteString("Fcu_f fired - fuel control unit",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Enable,                   { enable self }
INPUT
p <- A7 | p pr = E,                 { from fuel solenoid valve }
sin <- Status,

OUTPUT
Status <- sout = [ st ACT ],        { active }

ACTION
WriteString("Enable fired - fuel control umt",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),

TRANSITION Disable,                 { disable self }
INPUT
p <- A7 | p pr = D,                 { disable from solenoid valve }

```



```

sin <- Status,

fC <- FC_Old,
ca <- Ca_Old,

f3 <- Fuel_3,

```

OUTPUT

```

Status <- sout = [ st DEACT ],

{ continuous output becomes undefined }
Fuel_3 <- f3new = [ temperature f3 temperature,
                  pressure f3 pressure,
                  filtered f3 filtered,
                  undefined true ],

{ undefine remembered input }
FC_Old <- fCnew = [ temperature fC temperature,
                  pressure fC pressure,
                  filtered fC filtered,
                  undefined true ],

Ca_Old <- canew = [ temperature ca temperature,
                  pressure ca pressure,
                  undefined true ],

```

ACTION

```

WriteString("Disable fired - fuel control unit",0),
WriteString(" t=",0),
WriteReal(globaltime(),5,2),
WriteLn(),

```

END MODULE

```

{////////////////////}
{ Driver Module (apu 1) }
{////////////////////}

```

```
MODULE apu = types(),
```

PLACE

```

{ signals from terminators }
APU_On      null | self delay(1),      { APU ON swtich pressed }
APU_Off     null | self delay(20),     { APU Off swtich pressed }
Rpm_95      null | self delay(15),     { 95% engine rpm reached }
Tank1_off   null | self delay(5),      { Fuel tank no 1 switch off }
Seq_oil_closed null | self delay(10),  { sequencing oil pressure switch
                                       closes }

{ signals to terminators }
Open_door   null,                      { signal to open inlet door }
Close_door  null,                      { signal to close inlet door }

```

```

{ diverged APU_Off }
  APU_Off1      null,
  APU_Off2      null,

{ continous data flows }
  Fuel_1, Fuel_3      fueltypes,      { fuel from/to terminators }
  Fuel_A, Fuel_B, Fuel_C fueltypes,    { internal fuel flows }
  Hot_bleed_air_2    airtype,        { bleed air for fuel heating }
  Air                airtype,        { exhasut air from heater }
  Control_air_3      airtype,        { air for FCU regulation }

{ activation prompts }
  A1 prompttype | self first(),
  A2 prompttype | self first(),
  A3 prompttype | self first(),
  A4 prompttype | self first(),
  A5 prompttype | self first(),
  A6 prompttype | self first(),
  A7 prompttype | self first(),

```

```

TRANSITION initial,      { setup initial marking }
  OUTPUT

```

```

  APU_On      <- e0,      { events }
  APU_Off     <- e0,
  Rpm_95      <- e0,
  Tank1_off   <- e0,
  Seq_oil_closed <- e0,

```

```

  Fuel_1      <- f0 = [ { dummy values for continous flows }
                        filtered  false,
                        temperature 10,
                        pressure    0,
                        undefined   false ],

```

```

  Hot_bleed_air_2 <- a0 = [ temperature 90,
                           pressure    20,
                           undefined   false ],

```

```

  Control_air_3 <- a1 = [ temperature 20,
                           pressure    20,
                           undefined   false ],

```

```

  Fuel_A <- u0 = [ filtered  false,
                  temperature 0,
                  pressure    0,
                  undefined   true ],

```

```

  Fuel_B <- u1 = [ filtered  false,
                  temperature 0,
                  presure    0,
                  undefined   true ],

```

```
Fuel_C <- u2 = [ filtered   false,
                temperature 0,
                pressure    0,
                undefined   true ],
```

```
Fuel_3 <- u3 = [ filtered   false,
                temperature 0,
                pressure    0,
                undefined   true ],
```

```
Air   <- a3 = [ temperature 0,
                pressure    0,
                undefined   true ],
```

ACTION

```
WriteString("Initial Transition fired - APU driver",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),
```

```
TRANSITION Diverge,      { diverge APU_Off to APU_Off1 and APU_Off2 }
```

```
INPUT
e <- APU_Off,
```

OUTPUT

```
APU_Off1 <- e,
APU_Off2 <- e,
```

ACTION

```
WriteString("Diverge fired - APU driver",0),
WriteString(", t =",0),
WriteReal(globaltime(),5,2),
WriteLn(),
```

```
{ Module instances }
```

```
INSTANCE fcv1      fcv  ( APU_On, APU_Off2, Open_door, Close_door, A1, A2 ),
INSTANCE bpc1      bpc  ( Tank1_off, Rpm_95, A1, A3, A4 ),
INSTANCE pump1     pump ( Fuel_1, Fuel_A, A4 ),
INSTANCE pump_b1   pump_b ( Fuel_1, Fuel_A, A3 ),
INSTANCE filter1   filter ( Fuel_A, Fuel_B ),
```

```
INSTANCE fhc1      fhc  ( Fuel_B, A2, A5, A6 ),
INSTANCE heat1     heat ( Fuel_B, Fuel_C, Hot_bleed_air_2, Air, A6 ),
INSTANCE feat_b    heat_b( Fuel_B, Fuel_C, A5 ),
```

```
INSTANCE fsv1      fsv  ( Seq_oil_closed, APU_Off1, A7 ),
INSTANCE fcu1      fcu  ( Fuel_C, Fuel_3, Control_air_3, A7 ),
```

```
END MODULE
```

Appendix B

ESML/LOOPN Prototype Execution Output

Initial Transition fired - APU driver, t = 0.00
Initial transition fired - fuel control valve, t = 0.00
Initial Transition fired - boost pump control, t = 0.00
Initial Transition fired - pump, t = 0.00
Initial Transition fired - pump bypass, t = 0.00
Initial Transition fired - filter, t = 0.00
Initial Transition fired - fuel heater control, t = 0.00
Initial Transition fired - heat, t = 0.00
Initial Transition fired - heat bypass, t = 0.00
Initial transition fired - fuel solenoid valve, t = 0.00
Initial Transition fired - fuel control unit, t = 0.00
Init_t fired - fuel solenoid valve, t = 0.00
Init_t fired - fuel control valve, t = 0.00
T1 fired, valve opens - fuel control valve, t = 1.00
Enable fired - fuel heater control, t = 1.00
Enable fired - boost pump control, t = 1.00
Disable fired - pump, t = 1.00
Enable fired - pump bypass, t = 1.00
Bypass_f fired - pump bypass, t = 1.00
Filter_f fired - filter, t = 1.00
Enable fired - heat bypass, t = 1.00
T1 fired, Heater goes on - fuel heater control, t = 1.00
Disable fired - heat, t = 1.00
Disable fired - heat bypass, t = 1.00
Enable fired - heat, t = 1.00
Heat_f fired - heat, t = 1.00
T1 fired, pump goes on - boost pump control, t = 5.00
Enable fired - pump, t = 5.00
Pump_f fired - pump, t = 5.00
Disable fired - pump bypass, t = 5.00
T1 fired, valve opens - fuel solenoid valve, t = 10.00

Enable fired - fuel control unit, t =10 00
Fcu_f fired - fuel control unit, t =10 00
T2 fires, pump goes off - boost pump control, t =15 00
Disable fired - pump, t =15 00
Enable fired - pump bypass, t =15 00
Bypass_f fired - pump bypass, t =15 00
Diverge fired - APU driver, t =20 00
T2 fired, valve closes, fuel solenoid valve, t =20 00
T2 fired, valve closes - fuel control valve, t =20 00
Disable fired - fuel heater control, t =20 00
Disable fired - heat, t =20 00
Disable fired - boost pump control, t =20 00
Disable fired - heat bypass, t =20 00
Disable fired - pump, t =20 00
Disable fired - pump bypass, t =20 00
Disable fired - fuel control unit, t =20 00

Appendix C

STATEMATE Prototype Execution Output

SIMULATION TRACE REPORT

Simulation name trace1
Project APU_CASE_STUDY
Work area /home/1/capg/gclynch/workarea
Operator gclynch

SCOPE

=====

Statechart name	State name	Watchdog
-----	-----	-----
BPC		No
FHC		No
FUEL_CONTROL		No

CLOCKS DEFINITION

=====

GLOBAL CLOCK	Clock unit	1 SECONDS
Statechart BPC	Clock unit	1 SECONDS
Statechart FHC	Clock unit	1 SECONDS
Statechart FUEL_CONTROL	Clock unit	1 SECONDS

=====
=== Step 1 Phase 1 Time 0 clock units ===
=====

Changes caused by SUD

Activated activities FUEL_SYSTEM

States entered FUEL_SHUTOFF_VALVE,
FUEL_SHUTOFF_VALVE CLOSED, FUEL_SOLENOID_VALVE,
FUEL_SOLENOID_VALVE CLOSED

Basic states configuration

FUEL_SHUTOFF_VALVE CLOSED, FUEL_SOLENOID_VALVE CLOSED

=====
=== Step 2 Phase 2 Time 0 clock units ===
=====

Changes caused by environment

Generated events APU_ON

Changes caused by SUD

Generated events OPEN_DOOR

Activated activities FILTER, HEAT, PUMP, BP BYPASS_F, FH BYPASS_F

States exited FUEL_SHUTOFF_VALVE CLOSED

States entered FUEL_SHUTOFF_VALVE OPEN, BPC, BPC OFF, FHC,
FHC OFF

Basic states configuration

BPC OFF, FHC OFF, FUEL_SHUTOFF_VALVE OPEN,
FUEL_SOLENOID_VALVE CLOSED

=====
=== Step 3 Phase 3 Time 0 clock units ===
=====

Changes caused by environment

Generated events TANK1_OFF

Activated activities PUMP_F, HEAT_F

Stopped activities BP BYPASS_F, FH BYPASS_F

States exited BPC OFF, FHC OFF

States entered BPC ON, FHC ON

Basic states configuration

BPC ON, FHC ON, FUEL_SHUTOFF_VALVE OPEN,
FUEL_SOLENOID_VALVE CLOSED

=====
=== Step 4 Phase 4 Time 0 clock units ===
=====

Changes caused by environment

Changed data_items FUEL_TEMP changed to 70

Changes caused by SUD

Activated activities FH BYPASS_F

Stopped activities HEAT_F

States exited FHC ON

States entered FHC OFF

Basic states configuration

BPC ON, FHC OFF, FUEL_SHUTOFF_VALVE OPEN,
FUEL_SOLENOID_VALVE CLOSED

=====
=== Step 5 Phase 5 Time 0 clock units ===
=====

Changes caused by environment

Generated events SEQ_OIL_CLOSED

Changes caused by SUD

Activated activities FCU

States exited FUEL_SOLENOID_VALVE CLOSED

States entered FUEL_SOLENOID_VALVE OPEN

Basic states configuration

BPC ON, FHC OFF, FUEL_SHUTOFF_VALVE OPEN,
FUEL_SOLENOID_VALVE OPEN

=====
=== Step 6 Phase 6 Time 0 clock units ===
=====

Changes caused by environment

Generated events RPM_95

Changes caused by SUD

Activated activities BP BYPASS_F

Stopped activities PUMP_F

States exited BPC ON

States entered BPC OFF

Basic states configuration

BPC OFF, FHC OFF, FUEL_SHUTOFF_VALVE OPEN,
FUEL_SOLENOID_VALVE OPEN

=====
=== Step 7 Phase 7 Time 0 clock units ===
=====

Changes caused by environment

Generated events APU_OFF

Changes caused by SUD

Generated events CLOSE_DOOR

Stopped activities FCU, PUMP, HEAT, FILTER

States exited FUEL_SHUTOFF_VALVE OPEN,
FUEL_SOLENOID_VALVE OPEN

States entered FUEL_SHUTOFF_VALVE CLOSED,
FUEL_SOLENOID_VALVE CLOSED

Basic states configuration

FUEL_SHUTOFF_VALVE CLOSED, FUEL_SOLENOID_VALVE CLOSED