Dialogue Classification and Specification

Bollin Andreas
bollin@ist.tu-graz.ac.at

October 1999

Diplomarbeit in 874 Telematik

durchgeführt am

Ordinariat für Softwaretechnologie
Technische Universität Graz

Begutachter: O.Univ.Prof. Dipl.-Ing. Peter Lucas
Betreuer: Univ.Ass. Dipl.-Ing. Andreas Ausserhofer

I assure you that I have written this work by myself and that I did not use any sources or aids other than what I have declared in this work.

Dedication

Many thanks to my wife, who supported me on the long way of research with all her love. Without her and her knowledge about education this work would not be as complete as it is.
The next person I want to express my deepest thanks to is my mentor, professor Peter Lucas. He made it possible for me to visit the WebNet'98 conference in Orlando, which gave me new inputs in writing this work.
This would indeed be an incomplete dedication if I did not thank my family, too. Without their patience and understanding (I did not always have as much time as I wanted to spend with them) this work would never have been completed.
This work is dedicated to my beloved father who departed life too early.

ABSTRACT

At the moment no accepted way (or notation) exists for describing dialogues. In the last years, especially in the 1980s, many approaches came up, but no method, no tool or no idea brought a breakthrough in user interface and dialogue design. As even the meaning of phrases concerning the design process is not always clear, it is important to define and explain such common words as user interface, dialogue or user interface management systems.
This work gives a summary of the approaches which have come up in the last twenty years. Not only the more or less text oriented approaches like context free grammars, the user action notation or transition diagrams will be explained, but also the different graphical methods like Cards, Prototypers and Toolkits will be presented. To help the reader in comparing the approaches, 24 of them have been analysed, classified and their results have been graphically presented.
With all those different ways of specification in mind, Coloured Petri Nets, an approach in describing complex dialogue systems, will be presented. Coloured Petri Nets (CPNs) are a variation (or development) of ordinary Petri Nets, which were invented in the 1960s by Carl Adam Petri.
The use of CPNs to describe ordinary dialogue systems will lead to a classification of dialogues. With this classification it is possible to separate several basic dialogue elements, which will then be used to form the basis for a general dialogue system. Another major advantage of this classification is that the type of dialogue can be separated from its form, and therefore it is possible to use style files to generate different appearances of the same dialogue.
With basic and well specified dialogue forms available, it is also possible to generate dialogues automatically (from a dialogue description). As an example the basic idea of LaTeXES, (LaTeX for Educational Systems) will be presented which could then be the basis for the generation of larger educational systems.

ZUSAMMENFASSUNG

Im Augenblick existiert noch kein allgemeiner Ansatz oder eine Notation um Dialoge eindeutig und ausführlich genug zu beschreiben und dann aus dieser Beschreibung automatisch ein dynamisches Dialogsystem zu generieren. In den letzten Jahren, vor allem in den späten 80er Jahren, gab es viele Ansätze. Keine Methode oder Werkzeug, keine Idee erzielte einen Durchbruch im Design von User Interfaces.
Diese Arbeit versucht einen Überblick über Ansätze der letzten 20 Jahre zu liefern. Dabei werden verschiedene grafisch orientierte Ansätze (wie Cards, Prototypers und Toolkits) aber auch textorientierte Ansätze (wie XYacc oder die Task Action Grammar) vorgestellt. Um dem Leser eine Entscheidungshilfe zu bieten, wurden sämtliche Ansätze klassifiziert und in grafischer Form zusammengefaßt.
Aus dieser Fülle von Ansätzten wurden Coloured Petri Netzte (CP Netze) als Grundlage für die Analyse von Dialog Systemen ausgewählt. CP Netze sind eine Erweiterung von gewöhnlichen Petri Netzen, welche von Carl Adam Petri im Jahre 1962 erfunden wurden. Die Analyse der Dialoge führt zu einer Klassifikation von Dialogen und deren Struktur, die dann als Grundlage zum Aufbau eines Dialogsystems herangezogen werden kann.
Mit einer Menge an Basisdialogen und einer Klassifikation von Dialogstrukturen ist es dann möglich, komplexere Dialogsysteme zu spezifizieren und Dialoge automatisch zu generieren. Da bei der Klassifikation der Inhalt der Dialogelemente von deren Form getrennt wird, ist es möglich, das Aussehen des Dialoges über ein Stylefile zu steueren, ja sogar dynamisch zu ändern. Die Idee von LaTeXES (LaTeX for Educational Systems) liefert die Grundlage für ein derartiges System.

CONTENTS

1. Introduction . . . 1

2. Overview of Dialogue Design . . . 3
- 2.1 Definitions . . . 3
- 2.2 Guidelines for Comparison . . . 7
- 2.3 Specification Methods and Approaches . . . 10
  - 2.3.1 Formal and Semiformal [2] . . . 11
  - 2.3.2 Informal Methods - Building Tools [3] . . . 34
- 2.4 Summary . . . 38

3. Petri Nets . . . 41
- 3.1 Motivation . . . 41
- 3.2 The World of Petri Nets . . . 42
- 3.3 Informal Introduction . . . 44
- 3.4 Mathematical Model of PT Nets . . . 47
- 3.5 Coloured Petri Nets . . . 51
  - 3.5.1 Motivation . . . 51
  - 3.5.2 Mathematical Background . . . 55
  - 3.5.3 Analysis . . . 58
  - 3.5.4 Dynamic Properties . . . 60
- 3.6 Design/CPN . . . 62

4. Dialogue Classification . . . 67
- 4.1 The Problem of Dialogues . . . 67
- 4.2 Dialogue System . . . 69
- 4.3 General Classification Method . . . 71
- 4.4 Examples of Dialogues . . . 74
  - 4.4.1 Standard ML Quiz Example . . . 74
  - 4.4.2 Mathematica Notebook . . . 76
- 4.5 Dialogue Types . . . 78
- 4.6 Dialogue Structures . . . 80
- 4.7 Summary . . . 83

5. Future Work . . . 85
5.1 Ideas for Implementing a WWW-based Educational System . . . 85
5.2 Helper Applications . . . 87
5.3 Summary . . . 88

6. Conclusion . . . 91

Appendix 93

A. Assessment of Methods . . . 95

B. Graphical Comparison . . . 109
B.1 Abbreviations . . . 109
B.2 Graphical Comparison based on Assessment . . . 109

C. Petri Net Classifications . . . 113

D. Design/CPN Examples . . . 117
D.1 Design CPN - Screen Shot . . . 117
D.2 Producer/Consumer System - Statistics . . . 117
D.3 Quiz Model - Statistics . . . 118
D.4 Quiz Model - Occurrence Set . . . 119

E. Analyses Results . . . 121
E.1 Quiz Example - General View . . . 121
E.2 Quiz Example - Detailed View . . . 125
E.3 Mathematica Notebook - General View . . . 130

F. Syntax of CPN/ML . . . 141

LIST OF FIGURES

2.1 The Seeheim model of human-computer interaction (1985), including the unspecified secret box responsible for the control of communication . . . 3
2.2 Interactive software: involved users or roles . . . 6
2.3 Assessment projection: graphical and mathematical level . . . 9
2.4 Specification methods, general view . . . 10
2.5 Formal and semiformal methods, general classification . . . 11
2.6 Formal methods, detailed view . . . 12
2.7 YACC: YACC dialogue control . . . 14
2.8 YACC: dialogue specification for hangman . . . 15
2.9 XYACC: combining XYACC with an X-Windows application . . . 16
2.10 XYACC: part of xhangman specification . . . 17
2.11 Denotational approach: interaction between user and system . . . 18
2.12 Semiformal methods, detailed view . . . 21
2.13 ITS: architecture of applications . . . 23
2.14 ITS: typical dialogue frame using a simple menu . . . 24
2.15 ITS: frame of an airline reservation system . . . 24
2.16 ITS: tree of units describing a simple menu . . . 25
2.17 ITS: style definition of a simple menu . . . 26
2.18 Transition diagram . . . 27
2.19 Transition diagram: extended form indicating frequency of transitions (from [Shn97], p.163) . . . 28
2.20 Simple USE transition diagram . . . 29
2.21 State chart: simple bank transaction system (from [Shn97], p.164) . 30
2.22 Event automata: simple selection of items . . . 31
2.23 Event automata: NSD sub-diagram for an event automata . . . 31
2.24 Menu tree: Lycos search service called Sitemap (from [Shn97], p.251) 34
2.25 Interface building tools . . . 34
2.26 Hyper-Talk: event specification . . . 35

3.1 Petri Nets: a general classification . . . 43
3.2 Examples of simple PT nets . . . 45

3.3 PT net describing a simple resource allocation system (with the initial Marking M_0) (from [Jen97], p.4) . . . 46
3.4 PT net describing a simple resource allocation system with the marking M_1, reached from M_0 by T2p (from [Jen97], p.6) . . . 47
3.5 PT net: producer consumer system . . . 49
3.6 PT net describing a simple resource allocation system with the marking M_2, reached from M_0 by transition T1q (from [Jen97], p.7) . . . 50
3.7 CP net: net describing the simple resource allocation system of Fig. 3.3 (from [Jen97], p.10.) . . . 53
3.8 CP net: very compact net describing the simple resource allocation system of Fig. 3.3 (from [Jen97], p.20) . . . 55
3.9 CP net: occurrence set for the produce/consumer system of Fig. 3.8. Node and arc descriptors are toggled on for nodes 1 and 3 . . . 58
3.10 Design/CPN: quiz model, general structure . . . 63
3.11 Design/CPN: quiz model, hierarchy overview . . . 63
3.12 Design/CPN: quiz model, detailed view . . . 64
3.13 Design/CPN: quiz model, colour set definition . . . 65

4.1 Natural language dialogues forming subsets . . . 67
4.2 Basis of dialogue systems . . . 68
4.3 Dialogue system architecture, detailed view . . . 70
4.4 Dialogue system, ADI model . . . 71
4.5 Dialogue classification I, based on fields of application . . . 72
4.6 Combination of the Seeheim and the ADI model . . . 73
4.7 Block structure of the quiz example . . . 76
4.8 Notebook dialogue structure block diagram . . . 79
4.9 Basic dialogue elements . . . 80
4.10 Dialogue unit properties . . . 81
4.11 Dialogue classification III: based on different structures . . . 82

5.1 General structure of WWW-based educational systems . . . 86
5.2 General structure of a WWW-based educational system using S-Dialogues . . . 88

B.1 Comparison via separation and abstraction level . . . 110
B.2 Comparison via graphical and mathematical level . . . 110
B.3 Comparison via description and portability level . . . 111
B.4 Comparison via development and extension level . . . 111
B.5 Comparison via usability and readability level . . . 112
B.6 Comparison via mathematical and description level . . . 112

C.1 State machine: simple example . . . 114

C.2 FC Nets: simple net structures 114
C.3 ALG Net: sender/receiver model 115

D.1 Design/CPN: screen shot 117
D.2 Design/CPN: quiz model, the occurrence set 120

E.1 Analysis: Quiz v1.0 hierarchy 121
E.2 Analysis: Quiz v1.0 declarations 122
E.3 Analysis: Quiz v1.0 structure 123
E.4 Analysis: Quiz v1.0 occurrence set 124
E.5 Analysis: Quiz v2.0 hierarchy 125
E.6 Analysis: Quiz v2.0 behaviour declaration 126
E.7 Analysis: Quiz v2.0 definitions declaration 127
E.8 Analysis: Quiz v2.0 structure 128
E.9 Analysis: Quiz v2.0 occurrence set 129
E.10 Analysis: Notebook v1.0 hierarchy 130
E.11 Analysis: Notebook v1.0 regions 131
E.12 Analysis: Notebook v1.0 general structure 132
E.13 Analysis: Notebook v1.0 starting Mathematica 133
E.14 Analysis: Notebook v1.0 yes/no behaviour 134
E.15 Analysis: Notebook v1.0 menu 135
E.16 Analysis: Notebook v1.0 menu options 136
E.17 Analysis: Notebook v1.0 help system 137
E.18 Analysis: Notebook v1.0 cell behaviour 138
E.19 Analysis: Notebook v1.0 solver 139
E.20 Analysis: Notebook v1.0 NB work 140

LIST OF TABLES

2.1 24 approaches in user interface and dialogue design 7
2.2 TAG definition of cursor control: directory of tasks 13
2.3 TAG definition of cursor control: syntax of the commands 13
2.4 TAG definition of cursor control: consistent grammar 13
2.5 Multiparty BNF: login dialogue 18
2.6 Denotational Approach: domain specification for E_S 19
2.7 Denotational Approach: syntax specification of the dialogue 19
2.8 Denotational Approach: semantic specification of the dialogue 20
2.9 UAN: Selection of an icon 22
2.10 UAN: Deleting a file 22

3.1 PT net: definition of a simple producer/consumer system 49
3.2 CPN: tuple Σ representing the Coloured Petri Net specification of the resource allocation system in Fig. 3.8. 56
3.3 Simple Quiz: typical human computer dialogue 62

4.1 Quiz Example, basic components 75
4.2 Mathematica Notebook: basic components 77
4.3 Mathematica Notebook: interaction elements grouped by common application fields 78
4.4 Dialogue classification II: based on dialogue types 81

A.1 Assessment: BNF 95
A.2 Assessment: TAG 95
A.3 Assessment: Lex-yacc 96
A.4 Assessment: multiparty BNF 96
A.5 Assessment: UAN 97
A.6 Assessment: transition diagrams 97
A.7 Assessment: USE diagrams 98
A.8 Assessment: state charts 98
A.9 Assessment: place transition Petri Nets 99
A.10 Assessment: Coloured Petri Nets 99
A.11 Assessment: event languages 100
A.12 Assessment: declarative languages 100

A.13 Assessment: constraint languages 101
A.14 Assessment: application frameworks 101
A.15 Assessment: automatic generation 102
A.16 Assessment: denotational approach 102
A.17 Assessment: graphical editor 103
A.18 Assessment: menu trees 103
A.19 Assessment: prototypes 104
A.20 Assessment: cards 104
A.21 Assessment: interface builders 105
A.22 Assessment: editing tools 106
A.23 Assessment: ordinary toolkits 106
A.24 Assessment: virtual toolkits 107

B.1 Abbreviations used in assessment diagrams 109

1. INTRODUCTION

> We want principles, not only developed-
> the work of the closet-
> but applied,
> which is the work of life.
>
> Horace Mann, Thoughts [1867]

The design and implementation of intelligent educational systems include both the structuring of knowledge and the design of dialogues within the system and between the user and the system. The most important part is personal freedom in the design of both areas to achieve the best results in didactics and methodology.

The expansion of WWW (World Wide Web) provides a good means for presenting educational data. Because of its spread wide use and its variety in presenting data and interaction with the user, it is a good candidate to fulfil the demands of didactics and methodology. The language describing the content of a page is called HTML (Hyper Text Meta Language).

Since May 1996, the standardized version HTML 3.2 [Eis96] has provided the possibility to create pages dynamically and to interact with the user in a more complex way. With HTML 3.2 it is also possible to implement states and to include applications in the page. The content of the educational session can now be displayed in a pleasant way and various kinds of interactions lead to a greater extent of satisfaction by the end user. "Illustration is the basis of all cognitions" (Pestalozzi).

Using the above described techniques for future educational systems, all the claims in I.A. Comenius' "Didaktika Magna", which are the basis of didactics, can be fulfilled. This includes a demand for "... all differences to be distinguished, subsequent things to be based on previous ones and related things to be connected".

The good news is that major parts of methodology can be implemented throughout the WWW. Methodology can be split into three parts:

1. presentation,
2. common covering and
3. independent covering of facts.

The first part is ideally implemented by the WWW because the presentation of information was the basic idea behind the WWW. Common covering of facts can be realized through guided tours. Guided tours are not directly part of HTML 3.2, but can be implemented as linked pages. Systems like HM-Card or Hyper Wave [MS95] or WebCT [MJ98] have provided these features for several years. The third part (which deals with teaching unrelated and different topics) is the most difficult one to implement in WWW systems, because there is no direct support by HTML. The problem can be solved if applications (like Java Applets or Java), XML (eXtensible Markup Language) and dynamically generated pages can be used. The description of complex interactive systems is not standardized - there are many approaches and the major part of this thesis will put all of them into order.

- **Chapter 1.** The introduction sums up the structure of this work.
- **Chapter 2.** The design of user interfaces and the specification of dialogues belong together. Chapter two will describe the mainstream in the design of interfaces and in the specification of dialogues.
- **Chapter 3.** After summing up the advantages and disadvantages of the different methods, Petri Nets, especially Coloured Petri Nets, will be introduced. They will form the basis for the description of dialogues in chapters four and five.
- **Chapter 4.** The main focus of chapter four will be on dialogue classification. It turns out that it is important to look at existing systems to recognise the kinds and behaviours of typical applications. Based on the experience collected at this phase, it is possible to classify and (later) specify dialogues which will lead to a system that fulfils all demands described earlier.
- **Chapter 5.** Using formal techniques in describing dialogues, it is also possible to generate dialogue systems automatically and to prove their correctness. The last chapter concludes with the introduction of LaTeXES, an idea to extend LaTeX ([Kop99]) classes to describe and generate general dialogue systems.
- **Chapter 6.** The conclusion sums up the previous chapters and presents some ideas and problems for future research.

2. OVERVIEW OF DIALOGUE DESIGN

Good conversation is a compromise
between speaking and listening.

Ernst Jünger [* 1895]

The design of dialogues leads sooner or later to the question of how dialogues between the user and the system can be specified. This chapter starts with the definition of such common words like user-interface, dialogue element, toolkit or even UIMS. It describes the mainstream in specification methods and software tools that support design and software engineering. The chapter ends with a motivation for the use of Coloured Petri Nets.

2.1 Definitions

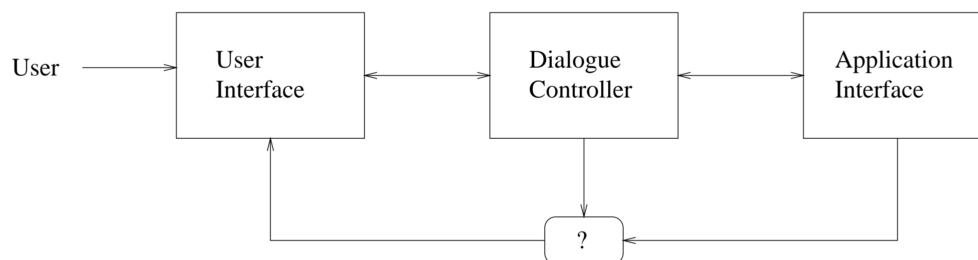


Fig. 2.1: The Seeheim model of human-computer interaction (1985), including the unspecified secret box responsible for the control of communication

Window technology has been improved dramatically since 1985 leading to rich featured user interfaces. On the other hand application-interface technology has barely changed during this period. It is still difficult to separate the application from the user-interface in an elegant way ([Nym95]). In 1985, at a workshop in Seeheim, the Seeheim Model (see Fig. 2.1) was defined. It divides an interactive application into three components: a presentation component (the user interface), a dialogue controller, and an interface to the application itself. The model itself evolved from a linguistic model which assumed the interface to be a dialogue between the computer and the user. It was the basis for research on semantic, syntactic and lexical levels

of dialogue, and encouraged many modifications. The original description of the model encounters three components:

1. The presentation manager directly communicates with the input and output devices (draws images and reads from physical input devices). This layer represents the lexical level.

2. The dialogue control component defines the structure between the application program and the user. This layer represents the syntactic level.

3. The application interface enables an access to the functionality of the application. This layer represents semantics.

The dialogue controller itself handles the communication between the application (data and procedures) and the presentation layer (interaction objects) and communicates with both layers. As this communication and the kind of control is not specified in detail, this part is represented as a small box with a question mark in the middle.

The little question mark in Fig. 2.1 represents the most important problem concerning the model. The simple way of decomposition is not useful without giving any guidelines to the programmer. The model itself tells nothing about the real structure of the system. Furthermore, as the user interface and the application are often tightly linked (in means of data and interaction), the separation could raise more problems than advantages. Brad Myers ([Mye97]) argues about the secret box with the question mark:

> "Next, as with most other systems conforming to the Seeheim model, the designer specifies essentially the entire user interface, by writing code for the little box..."

More criticism (which is partly deduced from the criticism above and from [Nic94]) can be summarized as follows:

- The model lacks the description of its relation to modern graphical environments (window management systems). There is no information about behaviour in distributed applications and concurrent systems. The general opinion seems to be that the Seeheim model works well for simple, small and static interfaces. Highly dynamic interfaces suggest that the specification should not follow the linguistic structure.

- The model itself lacks the definition of boundaries at the three levels and does not assist in the problems which may arise due to the different levels of abstraction at their boundaries. (This is a direct consequence of the problem mentioned before.)

- Performance issues are not described and concerned. As input has to be passed through all levels, the computation is expensive and requires the generation of code for every level. In fact, this problem might arise if all components are modules of the runtime system and performance is of importance. If the code is generated (and optimised) automatically (from an abstract description), this problem is of less importance.
- There is a lack of semantic feedback from the application to the dialogue manager and the presentation components.

Nevertheless, the Seeheim model is still useful. It provides the basis for further analysis and gives an abstract view of interactive systems. Chapter four uses a model of the system which is (as many other models) directly derived from the Seeheim model. (It presents an approach by describing the small box using Coloured Petri Nets.)

The enigma between abstraction and implementation is enlarged as the meaning of application and interface is not clearly separated in the literature. Some approaches use two or more layers of abstraction. They will be presented later in this chapter.

Human-computer dialogue.

It it important to mention that the term **human-computer monologue** can be used instead of the term **human-computer dialogue**. Albert Nymeyer (1994) states that the term "dialogue" is taken from real-world. He argues that human-human dialogue which consists of two (symmetric) parties (and where context plays a crucial role), is essentially different from human-computer dialogues, where the response of the computer is only a side effect. Nevertheless, in this work the term dialogue (or dialogue component) will be used for describing the interaction between the user and the application.

The user interface.

In general, the **user interface** of a computer program handles the input from the user and the output to the display (and other devices). The rest of the program is called **application** or the **application semantics**.

Types of users.

For reasons of terminology and for a better understanding of this material it is important to recognise that there are four different types (roles) of users involved in the software system (see Fig. 2.2). The user who uses the resulting program is called the end user. The next user creates the user interface and is called the user interface designer. The writer of the software, who works together with the designer,

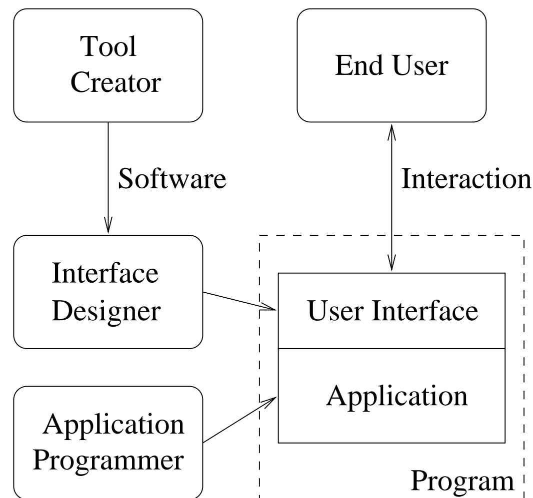


Fig. 2.2: Interactive software: involved users or roles

is called the application programmer. The designer itself may use special tools to create the user interfaces. And it is the tool creator who creates such tools.

UIMS and Toolkits

Historically two types of layered architectures (in analogy to Data Base Management Systems) have been developed in an attempt to provide the acquired flexibility in designing applications:

1. User Interface Management Systems and
2. Toolkits.

A UIMS (User Interface Management System) provides tools to design and develop the presentation and dialogue components, and to integrate these components into an application. Following the Seeheim model, the UIMS compiles a user interface, the dialogue specification and the application into an interactive application. In this analogy the dialogue specification between the user and the application forms the source program. UIMS separate the application from its interface.

On the other hand there are toolkits which separate the style from the application. Dialogue control remains at the back-end while the implementation of interaction techniques is hidden in a code-library. ([WB90])

Many of the tools are increasingly graphical in their user interface, enabling designers and programmers to build interfaces rapidly by dragging components and linking functions together. Productivity then raises from 50 to 500 percent ([Shn97]) as compared to other methods. The following sections will give an overview of existing approaches in the design and development of human-computer interfaces.

2.2 Guidelines for Comparison

It has turned out that the comparison and classification of the different approaches are not always simple and straight forward. To get an overview of and a feeling about the research in the field of user interface and dialogue design, 24 different methods, which can be found in literature, are examined. A short nickname (the name the author or the research group was using) is assigned to each method (see Tab. 2.1) in order to refer to them in a simple way.

The biggest problem is that many approaches use extensions of one ore more different methods. Some of them also hide their methods from the user. Another problem in comparing those approaches is that the same approach looks completely different for different users.

Abbreviations
BNF, Menu Trees, Context Free Grammars, Multiparty BNF
UAN, Transition Diagrams, USE Diagrams, Petri Nets
Coloured Petri Nets, State Charts, Event Languages
Declarative Languages, Constraint Languages, TAG
Automatic Generation, Denotational Approach
Graphical Editors, Application Frameworks, Prototypes
Cards, Interface Builders, Editing Tools, Toolkits
Virtual Toolkits

Tab. 2.1: 24 approaches in user interface and dialogue design

A solution to this problem is the introduction of a n-dimensional feature-space as a criteria for comparing the different dialogue specification approaches. Users can select some dimensions from this feature space and find the most suitable approach for their problem. (Projections to the first, second and third dimensions are possible and will provide a good basis for their decisions).

Ten characteristic features have been analysed (that is the possible level of ratings that can be reached) leading to a ten-dimensional feature space. As some approaches cover more than one level, it is possible for them to get more than one characteristic mark.

It is also important to mention that the ratings of dimensions five (readability) and six (development) are the writer's own opinion. For a (statistically) more objective rating, it is necessary to perform assessments with several users.

1. Dimension 1: Separation level.
   This level concerns the place in the Seeheim model: it answers the question

whether the approach is merely at the presentation level (9-7), at the dialogue control (6-4) or the functional core interface level (3-1).

2. Dimension 2: Mathematical level.
   This level describes the mathematical background. Fully formal (9,8), formal (7-3) and informal (2,1).

3. Dimension 3: Description level.
   This level answers the question whether merely the semantic or the syntax of the dialogue can be described. The values range from purely semantic (9) to purely syntactic (1). A value of 5 indicates that both could be described.

4. Dimension 4: Usability level.
   This level represents the skill of the user. Values range from novice (9), average user (5) to an expert user (1).

5. Dimension 5: Readability level.
   This level is tightly coupled with the usability level. This level gives an idea of how easy or difficult it is to read (or to work with) the notation (or working environment) by an average user. Values range from highly readable (9) to unreadable (1).

6. Dimension 6: Development level.
   This level concerns the development speed, an important factor in user interface development: values range from high (9) to low (1).

7. Dimension 7: Portability level.
   This level describes how easily transferable the dialogue system is between the different operating systems. Values range from easy (9) to difficult (1).

8. Dimension 8: Graphical level.
   To characterise the way of working with the approach, this level describes the kind of working environment. Values range from a direct manipulation environment (9) to a graphical environment (5) or a text based environment (1).

9. Dimension 9: Abstraction level.
   This level represents the distance to the implemented system. The level could range from a very abstract description (9) to a description which is near to the implemented system (1), meaning that in this case there is a machine-dependent code to write.

10. Dimension 10: Extension level.
    This level describes how easy (if possible) it is to extend the dialogue and

interface system. Values range from easily extendable (9) to not extendable (1).

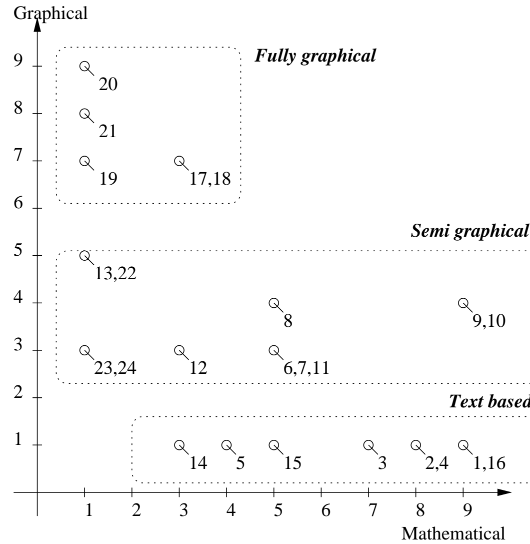


Fig. 2.3: Assessment projection: graphical and mathematical level

With the values of assessment it is possible, for example, to construct a method oriented cube (with the dimensions 1 to 3) and a user oriented cube (with the dimensions 4 to 6). Other combinations would, of course, be possible, as 10 dimensions can lead to 45 different projections into the two-dimensional space and 120 projections into the three-dimensional one.

The complete assessment and six projections (where the grouping of the different approaches can be seen very well) to the two dimensional space can be found in appendix (A): the separation-abstraction plane, the graphical-mathematical plane, the description-portability plane, the development-extension plane and the usability-readability plane.

The purpose of the next section is to classify the 24 approaches (see Tab. 2.1) in user interface design. As all approaches are different, only four characteristic elements are used as a basis for classification. This classification is based on the plane consisting of dimensions two and three (mathematic-description level which gives a good overview of the formalism of a method or approach) and the plane consisting of dimensions two and eight (mathematical-graphical level, see Fig. 2.3) giving an overview of the type of working environment.

It is important to know that other classifications would also have been possible (for example categories describing how user-friendly an approach it is). The cat-

egories chosen for this classification seem to be the two most important ones for average users who have to decide which kind of approach to use.

2.3 Specification Methods and Approaches

A look at publications in connection with user interfaces and dialogues shows that there are many approaches in the way of describing the interface, all using slightly different methods, and that most of the work was done during the late 1980s. Between 1982 and 1988 formal methods were especially in common, whereas graphical methods became popular after 1989. It is not easy to put all approaches into categories (see also previous section), because there are many different and incomplete approaches of classification in the literature.

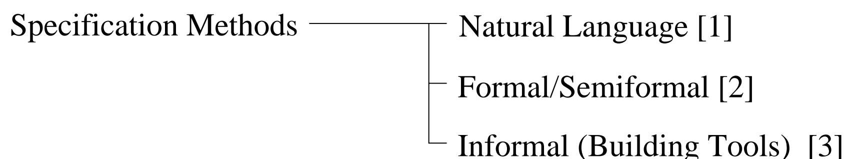


Fig. 2.4: Specification methods, general view

When looking at publications, it seems as if the border between the terms "method" and "approach" is vanishing. It is of course important not to mix the two of them up. A notation (like UAN) is not a method itself, and for every approach there are methods AND notations.

Nevertheless, in this work the term "specification method" is used. From my point of view the approach to use the notation "UAN" is also one methodology for specifying the dialogue system.

Following the classification of Ben Shneiderman [Shn97] (and the results of the analysis of the mathematical-description level), there are three specification methods for user-interfaces (see Fig. 2.4):

1. the natural-language specifications,
2. formal and semiformal languages and
3. interface building tools.

The default language for specification in any field is the designer's natural language. The problem with natural-language specifications is that they tend to be lengthy, vague and ambiguous. Using natural languages it is difficult to prove if the

specification is correct, consistent and complete. Only the formal or semiformal and graphical specification approaches will be considered, though a natural language, a sketchpad or a blackboard are still important parts in the design (process).

2.3.1 Formal and Semiformal [2]

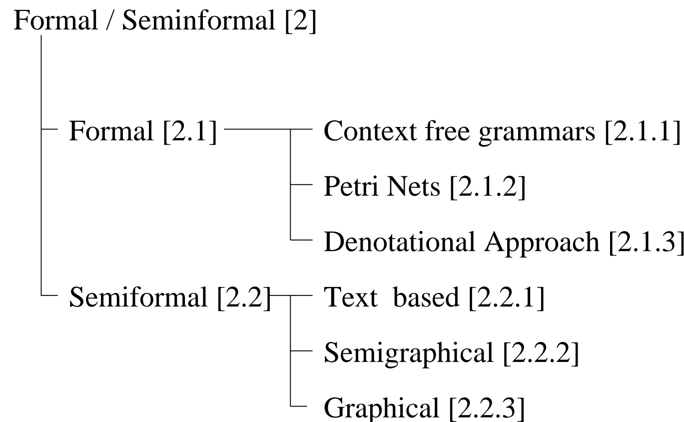


Fig. 2.5: Formal and semiformal methods, general classification

Formal and semiformal languages have been used in many areas with success. Fig. 2.5 gives a general overview of existing approaches in user interface specification (following classifications of [WB90], [Shn97], [Nym95] and [Mye97]). Formal techniques have many advantages:

- A system which is formally specified can be analysed and its properties can be deduced.
- Furthermore a system which is implemented from specification can easily and reliably be updated and managed.

The first (and very extensive) comparison of formal methods was done by Green [Gre86] in 1986. He was among the first to identify three more or less formal categories of UIMS: those that use a context-free grammar for specifying the user interface, those that use a transition network and those that are event driven.

Green also presented algorithms for converting formal specifications of the forms mentioned before into an executable code. He also stated that the event-driven approach had the largest descriptive power and presented an algorithm for converting the other two methods into the event driven form. However, each approach has its own advantages, so the following description does not imply any ranking.

Some tools require the programmer to key in a special purpose language, others provide an application framework to guide the programmer, while others automatically generate the interface from a high-level specification, and still others allow the interface to be specified graphically. Each of these types will be discussed later. Some tools use different techniques to specify different parts of the user interface, so these are classified by their predominant features.

Formal Specification Methods [2.1]

The first class of specification methods uses formal techniques to specify the user interface. Fig. 2.6 shows all methods using strictly formal approaches.

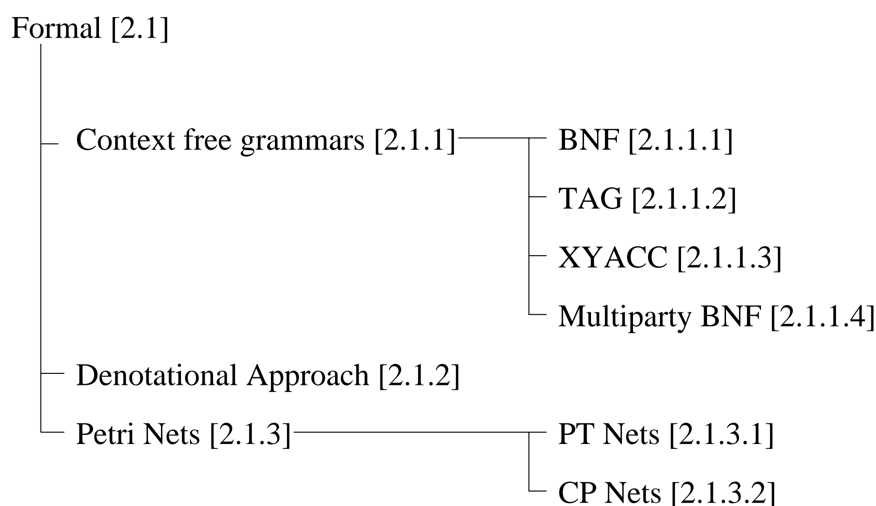


Fig. 2.6: Formal methods, detailed view

Context free grammars [2.1.1]

BNF [2.1.1.1]

In computer programming, the Backus-Naur form (BNF) is often used to describe the syntax of programming languages (see also [ASU86]). In the meanwhile, various forms of BNF have been created to accommodate the specific situation.

TAG [2.1.1.2]

An important goal for designers is a consistent user interface. Command languages easily tend to be inconsistent. For reasons of consistency Reisner (1981) [Rei81] expanded the idea of BNF to sequences of actions defining graphical interface systems. These sequences were "pushing a button", "selecting a colour", or

mv-cursor-1-character-fwd	[direction = forward, Unit = char]
mv-cursor-1-character-bwd	[direction = backward, Unit = char]
mv-cursor-1-word-fwd	[direction = forward, Unit = word]
mv-cursor-1-word-bwd	[direction = backward, Unit = word]

Tab. 2.2: TAG definition of cursor control: directory of tasks

"drawing a shape". Payne and Green (1986) expanded her work through a notational structure they called *task action grammars* (TAGs) by addressing the multiple levels of consistency (lexical, syntactic, and semantic). They also tried to solve the problem of completeness by defining complete sets of tasks. (For examples, the actions right, up and down constitute an incomplete set of cursor movements, because left is missing from the set of actions.)

As an example, a TAG definition of a cursor control has a dictionary of tasks as seen in Tab. 2.2 (see also [Shn97], p.58-60). The high-level rule schemas that describe the syntax of the commands are described in Tab. 2.3.

1.	task [Direction, Unit] → symbol [Direction] + letter [Unit]
2.	symbol [Direction = forward] → "CTRL"
3.	symbol [Direction = backward] → "ESC"
4.	letter [Unit = word] → "W"
4.	letter [Unit = word] → "C"

Tab. 2.3: TAG definition of cursor control: syntax of the commands

With the schemas described before it is possible to generate a consistent grammar as in Tab. 2.4. To demonstrate the completeness of the grammar of the command language, it can be tested against the complete set of task-action mappings.

mv cursor 1 character fwd	CTRL-C
mv cursor 1 character bwd	ESC-C
mv cursor 1 word fwd	CTRL-W
mv cursor 1 word bwd	ESC-W

Tab. 2.4: TAG definition of cursor control: consistent grammar

XYACC [2.1.1.3]

Albert Nymeyer [Nym95] showed that it was possible to use a context free grammar to describe the user interface by using the off-the-shelf compiler generator YACC to generate a dialogue controller and to integrate the dialogue controller into the popular X-windows system.

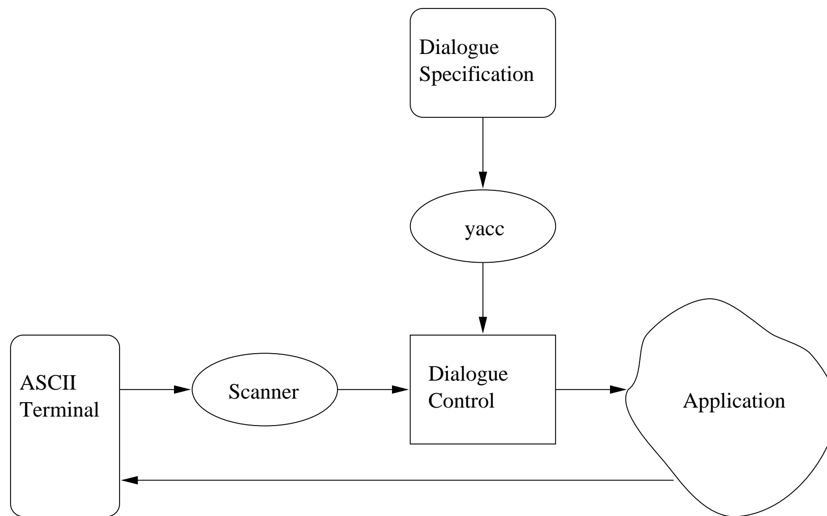


Fig. 2.7: YACC: YACC dialogue control

The YACC tool (see [ASU86] and [SF85]) (an LALR(1) parser generator) generates a parser for some language if a context-free grammar (specification) is given. The parser itself requires a scanner for reading the input as well as for generating a series of tokens.

The parser itself contains a push-down automaton, the semantic actions of the grammar and a set of tables (which are created from the specifications). Depending on the input tokens, the automaton uses the tables to change to a new state. The important part is that during the process of changing the state a semantic action can be carried out. The specification read by YACC is a context free grammar representing the syntax of the application, the ancillary C-code represents the semantic actions or functionality. See Fig. 2.7.

Fig. 2.8 shows the syntax for the game hangman and gives an idea of what YACC specification for dialogue control looks like.

Following the standard YACC notation, nonterminals begin with an upper case letter and tokens are underlined. The empty rule is indicated by an epsilon (e). The example shows that the dialogue consists of zero or more letter or word guesses (letter and word) and can be terminated via the optional token giveup. To complete the specification in Fig. 2.8, the semantics actions have to be added.

```
Hang    :   List
            Fini;

Fini    :   e
        |   giveup;

List    :   e
        |   List LorW;

LorW    :   letter
        |   word;
```

Fig. 2.8: YACC: dialogue specification for hangman

Albert Nymeyer's approach ([Nym95]) uses the X-windows characteristics, saying that an X-window application is event-driven and that the screen is constructed out of widgets which can be manipulated (also at run-time) via resource variables. He extends YACC to build XYACC applying the following modifications:

- The automaton must be a coroutine of the user-interface.
  In this case, the role of the scanner is fulfilled by the user interface itself. The coroutine could be achieved by using a token which represents the user's action as a formal parameter. The automaton shifts this token making any reductions it can. If the automaton cannot do more, it remains at a waiting state until it is called again. Terminating the application leads to the acceptance of the token.

- An error in the automaton must not be fatal.
  As the existing error-handling scheme of YACC is inappropriate to handle errors in dialogues, the error handler had to be removed.

- The user should be able to cancel a part of the dialogue.
  In the case of dialogue, cancellation is to delete the parsing of a rule. It means that it must be able to reinstate the automaton. Store rules had to be inserted.

- The application must also be able to direct the automaton.
  To receive tokens not only by the user-interface, the front end was slightly modified and a variable "nexttoken" was inserted which can be set by the application.

- The user-interface should indicate to the user those widgets that are "active".
  As the dialogue and the user interface are decoupled, neither the user interface

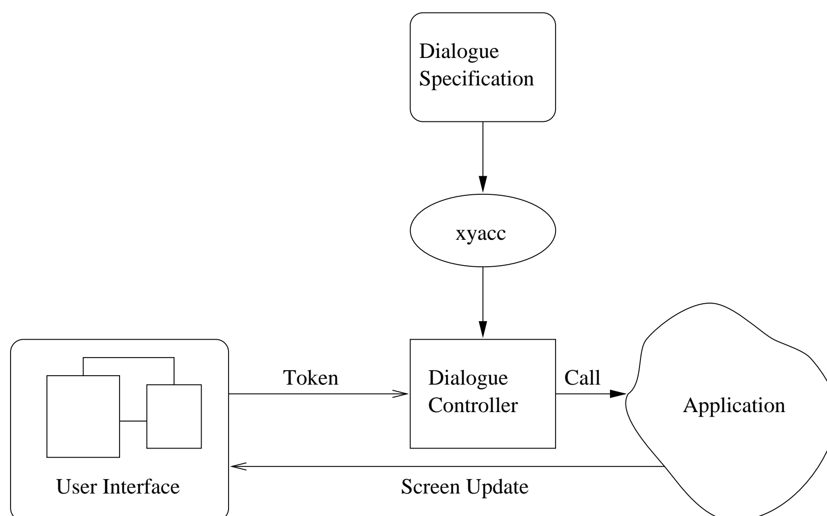


Fig. 2.9: XYACC: combining XYACC with an X-Windows application

nor the user know which widgets are active and which are not. The solution is to use a look-ahead set (which is extracted from the automaton) and an array coupling token names to the widgets.

The new parser generator XYACC has generated an automaton which could be called a dialogue controller for an X-Windows specification. Fig. 2.9 shows the relationship between the user interface (the scanner), the dialogue controller (which processes input tokens and calls application routines) and the application.

Parts of the corresponding specification of hangman for X-Windows can be seen in Fig. 2.10. In this specification, widget names are represented by an asterisk ("*message" for example). The buttons on the interface representing the alphabet are called toggle widgets. It is also remarkable that the specification uses explicit cancellation. The coroutine "initialize" (which means fetching a random word) sets the next token to cannot, if it is not possible to generate a word.

Multiparty Grammars [2.1.1.4]

To stress the interaction of software and users, Shneiderman (1982) invented *multiparty grammars* [Shn82], where nonterminals are labelled by the party that produces the string. These labels are typically the user U and the system S. Square brackets are used to access values of nonterminals. The grammar presented in Tab. 2.5 describes a typical login dialogue using multiparty BNF.

A major advantage of multiparty grammars is the effectiveness in text-oriented command sequences. A disadvantage is the lack of describing two-dimensional styles.

```
Hang      !                { initialize( cannot );
                             write2widget ( *message, "Your word has
                                             %d letters", strlen(temp));
                             write2widget ( *count, "%d", count);
                             write2widget (*word, "%s". temp);
                           }
                 List
                 Cancel
                 Fini
                 Cancel;

Fini      :      cannot    { write2widget (*message, "Unable to init"); }
          |      giveup    { write2widget ($1, "The word is%s", word); };

List      :      e
          |      LorW;

LorW      :      Letter    { if (letcmp($1, word)) {
                                  setcmp($1, word, temp);
                                  if (strcmp (word, temp))
                                      write2widget (*message, "Good guess");
                           ...}}
          |      word      { if (!strcmp( getword ($1 ), word)) {
                             ...}}

Letter    :      leta      { $$ = 'a'; toggle ( $1 ); }
          |      ...
```

Fig. 2.10: XYACC: part of xhangman specification

Though menu selection can be described using this kind of grammar, the central aspect of tree structures and traversals is not directly shown in this approach.

Denotational Approach [2.1.2]

As mentioned before, formal methods have many advantages. The disadvantage of some formal methods described before is that they cover only parts of the whole problem in a formal manner.

A complete formal specification has to be based on a fundamental model of interaction. It also has to provide the basis for interface specification, prototyping, and developing or evaluating environments. As formal methods became popular in the 1990s, some approaches use semantic functions to describe the meaning of dialogues ([BC94], [AWM95] and [BS93]). Some of them use a combination of other approaches: Lynn Marshal [Mar90], for example, uses a combination of State Charts and VDM. Gavin Lowe [Low93] shows that it is possible to describe communication processes in combination of a denotational semantics and NPA graphs (nondeterministic, probabilistic action graphs). The approach of Matsubayashi [MTT95] uses

<Session> ::= <U: Startup> <S: Response>
<U: Startup> ::= Login <U: User name>
<U: User name> ::= <U: String>
<S: Response> ::= Hello [<U: User name>]

Tab. 2.5: Multiparty BNF: login dialogue

a BNF-like style to describe the syntax (structure) of the dialogue and a notion derived from Denotational Semantics [Sch86] for the description of the semantic (meaning) of the dialogue.

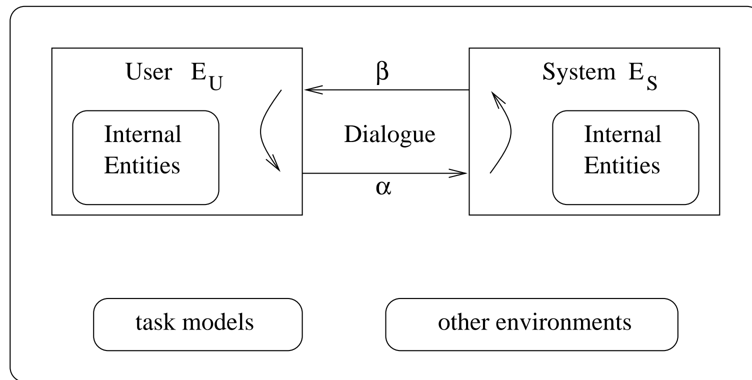


Fig. 2.11: Denotational approach: interaction between user and system

Fig. 2.11 shows the interaction between the user and the system. The dialogue itself can be specified by:

$$\mathcal{D} = \{\beta_0\alpha_1\beta_1\alpha_2\beta_2\alpha_0...\alpha_n\beta_n|\alpha_i \in \Sigma^*, \beta_i \in \Gamma^*, \Sigma \cap \Gamma = \emptyset\}$$

Thus, the dialogue is represented by a sequence of symbols, where Σ represents the set of user's input symbols and Γ represents the set of system's output symbols. Both the user and the system have internal entities.

The meaning of the user's input symbol α is represented by the mapping from E_S to E_S and β. The specification of the whole interaction can be described via the terms E_U, E_S, Σ, Γ and the mapping which defines the meaning of the user's input and system's output symbol.

Matsubayashi presents a specification of Macintosh Finder (a Macintosh file manipulation program) to show that it is possible to define the syntax and semantic of a user interface.

Desktop	:	Size × *Disk** WinList × WinDB × Selection × Menubar
Size	:	(N × N) /* display size */
Disk	:	Name × Content × Content × Content
		/* disk name, disk content, desktop and trash content */
Content	:	(Folder + File + Alias)* /* List of content
		of disk or folder */
...		

Tab. 2.6: Denotational Approach: domain specification for E_S

The domains for the specification of the meaning are E_S, Σ and Γ. E_S represents the system internal entities and can be specified as a product domain:

E_S = **Desktop** × **Mouse** × **Clipboard** × **FindStr**

An extract of the specification can be seen in Tab. 2.6. The other domains can be defined in a similar way.

Dialogue$_{com}$	::= Init **D**$_{com}$
D$_{com}$	::= ϵ | (NewFolder $<$ NewName $>$ Response |
	...
	| ResizeWindows Response) **D**$_{com}$
...	
Dialogue$_{mk}$	::= Init **D**$_{mk}$
D$_{mk}$	::= ϵ | (**MKDrag** | Click Response | DblClick Response
	...
	Right Response) **D**$_{mk}$
MDrag	::= **MC Drag1**
Drag1	::= PressMB Response **Drag2**
...	

Tab. 2.7: Denotational Approach: syntax specification of the dialogue

The syntax is specified on the command level ($Dialogue_{com}$) and on the mouse key ($Dialogue_{mk}$) level. Matsubayashi uses a BNF-like notation to describe the syntax: non-terminals are bold-faced, an ϵ represents no input and the system's output symbols are underlined (see Tab. 2.7).

The last thing to do is to define the semantic functions (execute, display, action and echo). As an example, the function `action` is given (see Tab. 2.8) which defines the meaning for mouse key level input symbols from the user.

```
action[ MC MDrag ] dt m c f =
    let
        (dt', m', c', f') = action [ MC ] dt m c f
    in
    action [ MDrag ] dt' m' c' f'

action[ PressMB Response MDrag1 ] dt m c f =
    case whereMC dt m of
        onMenuBar :
            let (dt', m', com) = selectMenu [ MDrag1 ] dt m in
            execute com dt' m' c f
        onWindow :
            ...
    end case
```

Tab. 2.8: Denotational Approach: semantic specification of the dialogue

The major advantage of this approach is that

- it is applicable to both character based and graphical based interfaces,
- it can deal with several levels of abstraction,
- the meaning of the dialogue sequences can be specified and
- the model can easily be extended by new environments.

Nevertheless, a specification that uses this method is not easy to read and the designer has to be an expert in denotational semantics. Matsubayashi shows that it is possible to develop working environments, but at the moment, the system is still at a research level.

Petri Nets [2.1.3]

Petri Nets are semi-graphic specification methods with a strong mathematical background. There exist many of its variants (Place Transition Nets or Coloured Petri Nets), all will be described in a more detailed form in chapter three.

Semiformal Specification Methods [2.2]

Semiformal specification methods can be divided into three different classes: textual based, semi graphical and graphical (see Fig. 2.12). The text based approaches (User Action Notation, Application Frameworks and Automatic Generation) will be presented below:

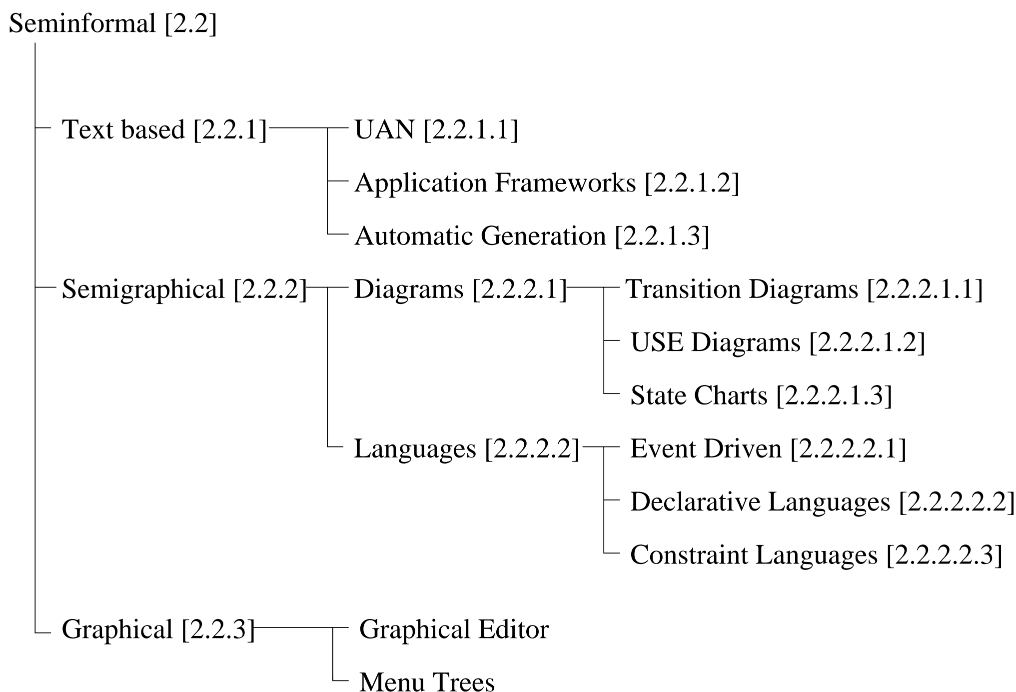


Fig. 2.12: Semiformal methods, detailed view

Text based [2.2.1]

User Action Notation [2.2.1.1]

In 1990 Hartson presented the UAN (*User Action Notation*) as a high level form of notation which focuses on the users' tasks. The major advantage is that this kind of notation solves the problems that grammars cause in connection with direct-manipulation interfaces (great variety of permissible actions and visual feedback). Furthermore, a higher level form of notation helps in situations where the context is important to determine the meaning of an input.

Hartson gives an example of a user who selects an icon with a mouse. Using the UAN notation, the selection of an icon is presented by ∼[icon], whereas pressing the mouse button is represented by Mv, followed by a mouse button release M ˆ . The highlighting of the icon is represented by icon! (see Tab. 2.9).

The deletion of a file is a more complex example. Tab. 2.10 shows how to define the dragging of a file icon around the display to a trash icon while holding down the mouse button. The interface feedback to this action is to highlight the file that is selected and to de-highlight ("file-!") other files. The outline of the icon is then dragged to the trash icon (indicated by outline(file > ˆ). The user drops the file icon, the file icon is erased and the trash icon blinks.

User Action	Interface Feedback
~ [icon] Mv	icon!
M ˆ	

Tab. 2.9: UAN: Selection of an icon

User Action	Interface Feedback	Interface State
~ [file] Mv	file!, forall(file!): file-!	selected = file
~ [x,y]*	outline(file) > ~	
~ [trash]	outline(file) > ~, trash!	
M ˆ	erase(file), trash!!	selected = null

Tab. 2.10: UAN: Deleting a file

UAN is using symbols that mimic the actions (such as "v" for pressing a button, a "ˆ" for a button release, and a "~" for a cursor movement). UAN has interface specific symbols for actions and for concurrency, interrupts and feedback. Chase et al., 1994, stated that UAN is a compact and powerful approach in specifying system behaviour and describing user actions. Nevertheless,

- UAN lacks specification in drawing programs, animations, relationships across tasks and interrupted behaviour, and
- the UAN is not easy to learn.

Application Frameworks [2.2.1.2]

Application Frameworks came up in the late 1980s. They all provide standard classes as part of the framework to guide programmers. The designer has to write code in inheriting subclasses from the application-specific superclasses.

Going back in history, frameworks became common as companies like Apple (MacApp) and Unidraw (InterViews) found out that programmers had difficulties in using their toolkits for two reasons:

1. Programmers had problems in calling toolkit functions correctly.
2. It was a problem to insure that the resulting application meets the user interface guidelines.

All application frameworks have one feature in common: the designer has to write a code. Layout and control are handled by the framework. (Apple supported Object Pascal as a programming language. Unidraw, on the other hand, used the C++ object oriented language.)

Automatic Generation [2.2.1.3]

The disadvantage of language based tools is that the designer always has to deal with the style (positioning, size, format and design) of the user interface. Automatic generation is an approach to solve this problem. Using a higher level specification (many of the systems are rule based), the tool is able to create a user interface and hides the problems of style mentioned before. Automatic generation has proved its value in designing menus or dialogue boxes. Some of the tools available are also able to generate an interface from a list and description of the application procedures.

The main advantage of automatic generation is that the interface stays consistent. The standard description of procedures (used by the application) also enables the tool to generate some kinds of on-line help automatically.

To generate the visitors information system for the EXPO'92 in Seville a new approach was developed: the ITS [WB90] approach.
The remarkable thing of ITS is that it makes use of a four-layered architecture. It provides separate tools for back-end computation, dialogue control, interface style definition and the implementation of graphic primitives.

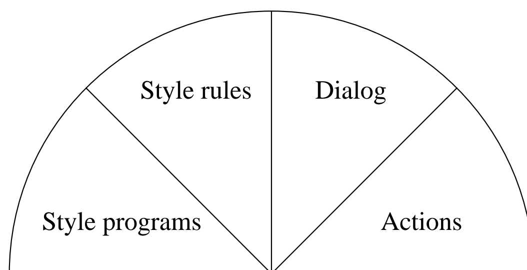


Fig. 2.13: ITS: architecture of applications

Fig. 2.13 presents the four layers of the ITS system.

1. Actions write and read data values without concerning the dialogue control. It is possible to connect more than one view with the data values they compute. There is (at this point) no connection to any style.

2. The dialogue layer contains logic frames with the control flow among them.

3. The style rule layer composes various dialogue objects in the frames with style programs. They are executed at compile time.

4. Style programs manage the run-time changes of dialogues. They include routines for text formatting, menus and images. Routines can be added on request.

```
:frame id=main
  :choice purpose=overview, message="Select an item to"
    :ci message="View flights for today", activate=check_today
    :ci message="View future flights", activate=check_by_date
    :ci message="Make a reservation", activate=reserve
  :echoice
:eframe
```

Fig. 2.14: ITS: typical dialogue frame using a simple menu

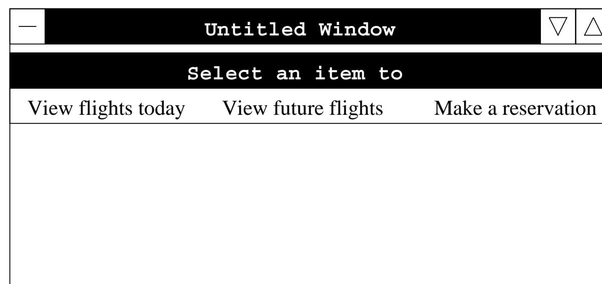


Fig. 2.15: ITS: frame of an airline reservation system

Fig. 2.14 presents a typical frame definition for a simple dialogue. It specifies an airline reservation system opening frame (see Fig. 2.15). In this example, `Choice` and `choice items` statements are the keywords of the dialogue language. Descriptive text for titles or prompts are provided by `Messages` and branches to other frames can be activated using the `activate` statement.

When arguing about the style layer, it should be clear that **style** is a coordinated set of decisions on the appearance and behaviour of the interaction techniques used in a family of applications. Three aspects of this definition are important [WB90]:

1. The definition refers to both the input and output characteristics of the interaction

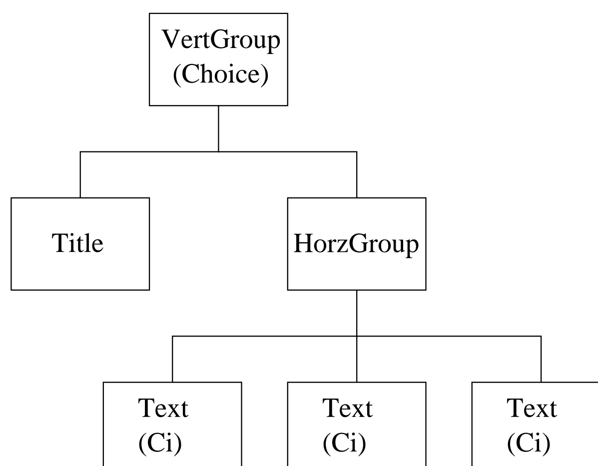


Fig. 2.16: ITS: tree of units describing a simple menu

2. It considers style in small (individual techniques) or large decisions.

3. The definition applies to more than one definition.

In ITS this range from general to special cases is represented as a hierarchy of rules which are separated into a "condition" and a "results" part (using the ITS terminology). Rules are extendable by the user when implementing an open architecture. They are responsible for refining general structures (for example a users choice) into structures as shown in Fig. 2.16 by giving templates to control how each dialogue node should be represented.

Fig. 2.17 shows a rule for the creation of a simple choice menu. All choices are matched against the condition statement. The set of units describes the nested structure of the choice for this style (equivalent to the Fig. 2.16). As it can be seen, the condition statement matches only the type of dialogue objects, in this case, the choice menu. The child unit is then duplicated for all `Ci` statements in Fig. 2.16.

Automatic generation seems to be very appealing, but in fact this approach is still at a research level. Disadvantages are:

1. Interfaces generated by automatic generation are generally not good enough.

2. The designer has to learn a special purpose specification language.

Taking a look at applications which use automatic generation, this method seems to be the best one for the generation of dialogue boxes and menu based interfaces.

Semigraphical [2.2.2]

```
:conditions source=choice
    fire for all choices
  :unit type=VertGroup
      the block arranges the title vertically
      above the set of items
    :unit type=Title
    :eunit
    :unit type=HorzGroup
        the item is replicated for all children of
        the choice, i.e. all ci statements
      :unit type=Message, replicate=all
      :eunit
    :eunit
  :eunit
:econditions
```

Fig. 2.17: ITS: style definition of a simple menu

The next class of semiformal specification methods is both textually and graphically well based, because it uses features of both. They can be divided into two further classes. The first type mainly uses diagrams for specification. The other type uses graphical objects only for illustration and focuses on the language (see also Fig. 2.12).

Transition Diagrams [2.2.2.1.1]

Since many parts of user interfaces deal with sequences of input events, it is natural to think of using state transition networks to describe the interface. Typically, transition networks have a set of nodes representing system states and a set of directed arcs (links) representing possible transitions. Arcs are labelled with input tokens which can cause the transition from one state to another.

Fig. 2.18 shows a simple example of a transition diagram taken from Dan Olsen's interactive pushdown automata (see [DRO84]) for user interface management. Transitions are labelled with a selector and the next state or action. At any time, the states S (Start), R (Reenter) or C (Cancel) are active if the nonterminal "<GetPoint>" is processed, which then leads to executing the matching transitions (drawing or removing the grid on the screen or returning to state 2).

One of the first approaches using transition networks was in 1968, when Newman implemented a simple tool using finite state machines to handle text input. In the meantime, many forms of transition diagrams have been created (with special notations to match the requirements of various application areas). Many of them

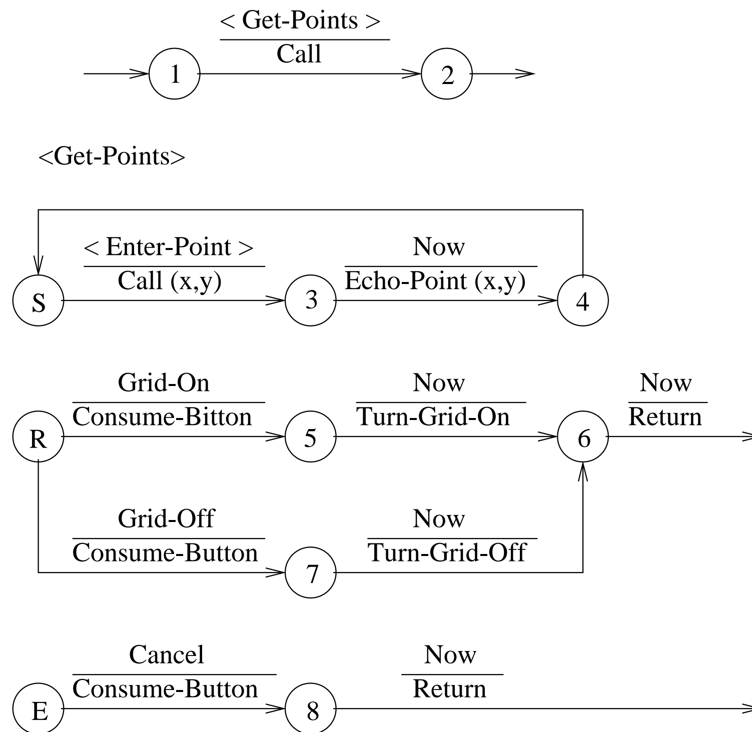


Fig. 2.18: Transition diagram

just extend the notational form of presentation (for example links are labelled by indicators to show how frequently each transition is made, see Fig. 2.19) and use additional attributes to focus on special models (for example the relationship between device handling and dialogue control, as Dan R. Olsen described in 1984 [DRO84]).

USE [2.2.2.1.2]

The year 1975 was the beginning of the User Software Engineering (USE) project with the idea of combining concerns about user involvement in the design of interactive information systems with those of software engineering. The outcome was the creation of a methodology and a set of tools supporting this methodology. The requirements for the user interfaces were:

1. Formalism.

2. Completeness, which means that the notation had to be self-contained (including user input, system output and system operations).

3. Comprehensibility. The notation had to be comprehensible to both the system developer and users.

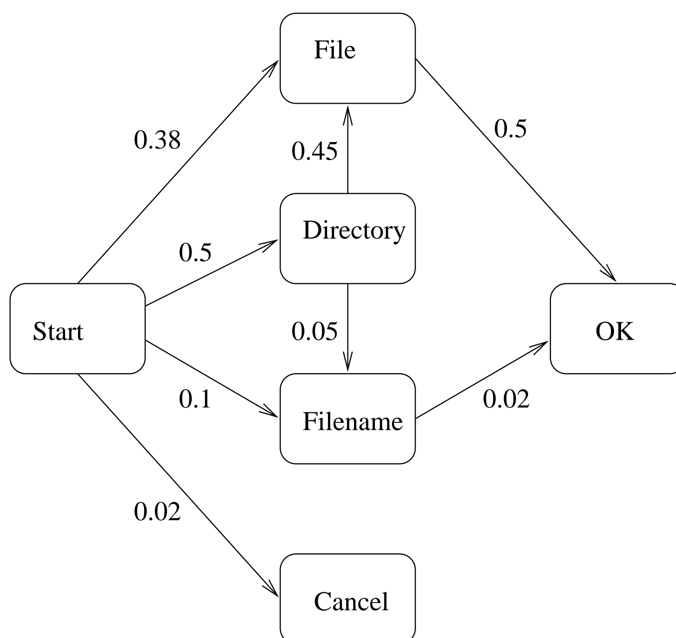


Fig. 2.19: Transition diagram: extended form indicating frequency of transitions (from [Shn97], p.163)

4. Flexibility, to support a broad variety of dialogue styles.

5. Executability.

While searching for an appropriate notation to describe user interfaces, Wasserman ([Was85]) decided to adapt transition diagrams for the USE methodology. The model itself contained just three different symbols:

1. Nodes. They are shown as circles and represent a stable state awaiting some user input.

2. Arcs. They are shown as arrows connecting nodes and represent a state transition based on some inputs. The input itself is designated either by a string literal (for example "quit") or by the name of another diagram, enclosed with angle brackets (such as <diag2>). A blank transition is the default transition.

3. Operations. They are shown by a small square with an associated integer. An action may be associated with a transition to represent an operation. The same action may be associated with more than one arc.

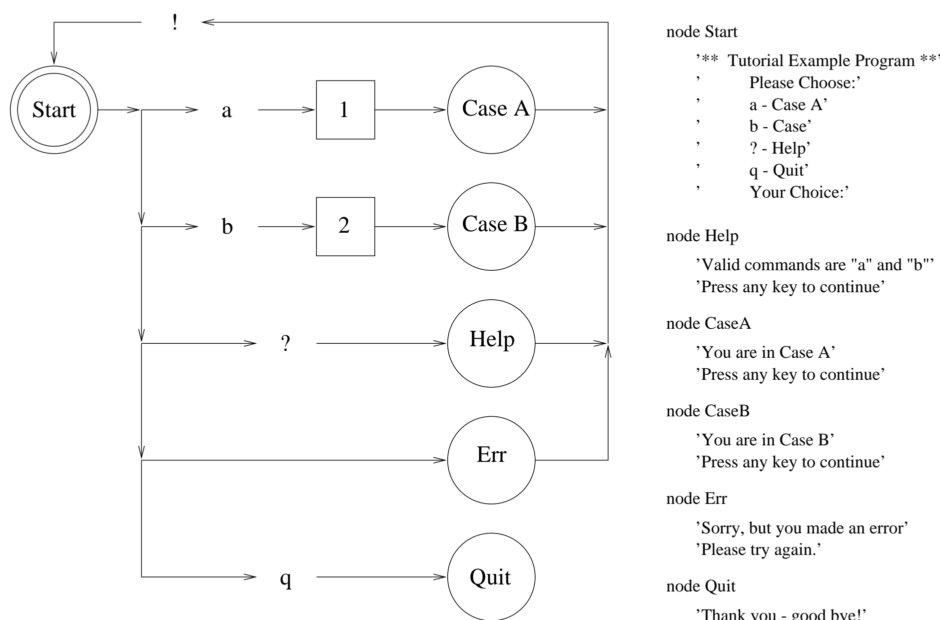


Fig. 2.20: Simple USE transition diagram

A simple state transition diagram is show in Fig. 2.20. The diagram begins at "Start" where it waits for input. There are transitions to one of five nodes based on the input. Input "a", for example, causes action 1 to be performed. Input "q" causes a transition to "Quit".

Nevertheless, transition diagrams have two main disadvantages:

1. Large numbers of arcs. Interactive systems attempt to be mode free, which means that at nearly each point the user has a wide variety of choices. Diagrams are becoming unwieldy.

2. There are no concurrent operations on multiple objects.

State Charts [2.2.2.1.3]

There are some variations that help this situation. The next extension is the use of *Petri Nets*, which will be described in detail in chapter three. The second one is the use of *State Charts*, which offer a grouping feature and extensions as concurrency, external interrupt events and user actions. Fig. 2.21 [Shn97] shows a simple example of a bank transition system where the grouping feature (represented by round tangles) is applied. Repeated actions can be factored out this way.

In the year 1985, Jacob ([Jac85]) invented a new formalism which is a combination of state diagrams and event languages. There can be multiple diagrams active

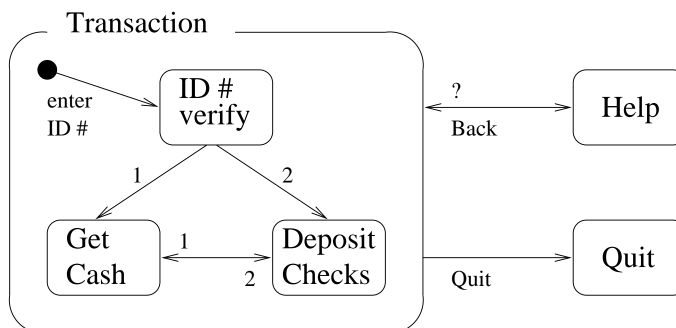


Fig. 2.21: State chart: simple bank transaction system (from [Shn97], p.164)

at the same time, and flow of control transfers from one to another in a coroutine fashion. This leads us to the second class of semigraphical specification methods: methods that are merely language driven.

Event Languages [2.2.2.2.1]

Input tokens in event languages are considered to be events that are sent to individual event handlers. Each event handler has some condition clauses, that define what types of events it can handle when it is active. The body of an event handler can itself cause events, changing the internal state of the system. The event automata itself can be simple or parallel, whereas a parallel automata is composed of simple ones.

The major advantage is that it is often much easier to program multiple interactions which are available at the same time using multiple processes. It is up to the user which interaction to make use of. The disadvantage is:

1. It is difficult to write a correct code. The flow of control is not easily localised and small changes can affect the behaviour of the whole system.

2. It is difficult to understand the code once it reaches nontrivial size.

In 1994 Viehstaedt [VM94] showed that it was possible to use event automata to describe complex graphical user interfaces in connection with editing operations and diagrams. The simple example in Fig. 2.22 shows an event automata for the selection of items. The little arrows on the borderline of states (described with circles) indicate a priority of outgoing arcs. Nassi-Shneiderman diagrams (NSDs), like in Fig. 2.23, are used to describe the behaviour of the system. Viehstaedt uses state charts whose transitions are only allowed to be labelled as “event(condition)”.

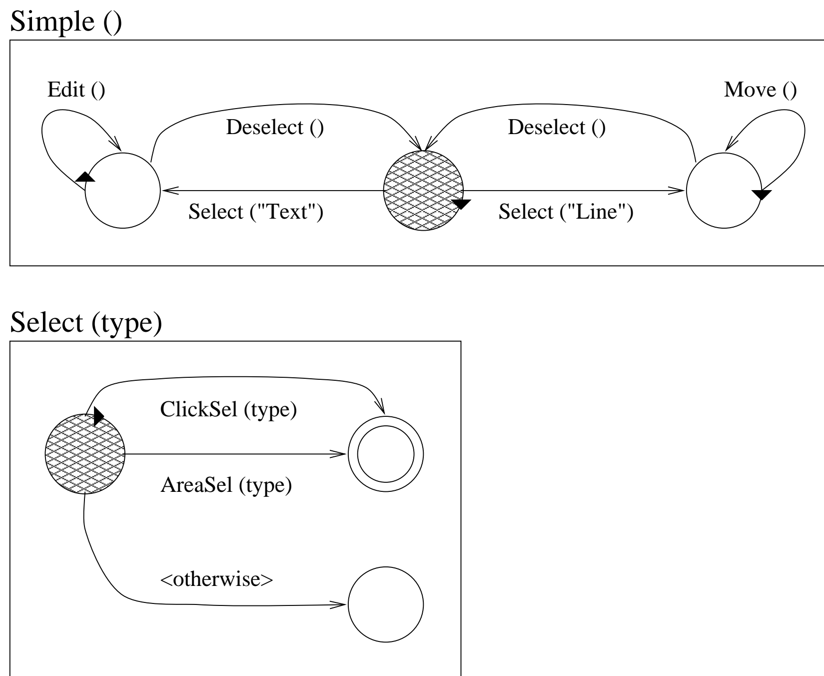


Fig. 2.22: Event automata: simple selection of items

In this example, initial states are indicated by filled circles, final states are represented by double circles. There can be one, more than one or no final state in the system. Edges are either labelled with conditions (including primitive events) or are labelled with subautomata. When `Simple` is started, it tries to reach another state from the initial one. In this case, all subautomata on edges connected to the initial state are activated. Low-level events can also be processed (via all subautomata that are activated at that time).

In the example, the initial state enters a new state if one of the subautomata reaches a final state. As the automata are evaluated in parallel, reaching the final

y — t simple transition ?	n
action(t)	Let S be transition t's subautomaton If S is not running, initialize S substatus := S.go (current_event)
Set status, new_state	if substatus != <NoStepDone>, Set staus, new_state depending on substatus

Fig. 2.23: Event automata: NSD sub-diagram for an event automata

state might happen at the same time. In this case, a small arrow indicates the priority. This rule gives `Select(''Line'')` the priority over subsequent arcs. If subautomata get stuck, they can signal unsuccessful termination via a transition to a give-up state. In this case an <`otherwise`> transition can be made.

Another approach in 1994 was the introduction of "Interactors" which are objects encapsulating states and events. More information can be found in [DFHP94].

Declarative Languages [2.2.2.2.2]

When using declarative languages, dialogues are defined in the "what should happen" manner, rather than using the procedural "how to make it happen" manner. The HP/Apollo's Open Dialogue, 1985 allows the designer to specify the user interface by means of this method. The user interface has already supported basic forms (fields, menus, buttons). The application program itself is connected to the interface via variables which can be accessed by the interface and the application. But typical layout description languages coming with toolkits also use the declarative style. In Motif (Motif UIL), for example, widgets are defined this way.

Declarative languages have two main advantages:

1. The user interface designer only has to deal with the information that should be passed between the user and the system.

2. The designer need not worry about the sequence of events.

A major disadvantage is, however, that only certain types of interfaces are supported. The kinds of interactions that are available for the designer are preprogrammed and fixed. In the systems mentioned before, there is no support for interaction objects like dragging of graphical objects or rubber-band lines.

Constraint Languages [2.2.2.2.3]

There are a number of interface tools which allow for the use of constraints to define the user interface. Examples of these are Sketchpad (which was a pioneer work from Sutherland, 1963 defining a drawing editor) or Thinglab (which uses constraints for the definition of graphical simulation). In most cases, constraints are used in research toolkits (to be described later). All objects can be connected with constraints (values) which describe relationships. These constraints are specified once and are automatically maintained by the system. As an example, the designer can specify that the position in a window is connected to (a value of) a slider. In this case the system will update the window if the slider is moved.

One of the major advantages of constraint languages is that they are a very natural way of describing relationships of objects in user interfaces. There are, however, two reasons why constraint languages are not commonly used:

1. For solving constraints, an efficient run-time system is necessary.

2. It is hard to find and to debug errors, and it is difficult to keep track of the causes and consequences of values which keep changing.

The last subclass of semiformal specification approaches is the class where graphical methods are used.

Graphical [2.2.3]

Many of the tools nowadays support a graphical way of specification. They have one thing in common: they allow the user, at least partially, to place objects on the screen using a pointing device. There are two reasons why this approach is that popular:

1. This kind of technique is much easier to use for both, the programmer and the novice.

2. A graphical tool seems to be most appropriate to specify the graphical appearance of applications.

The graphical specification approach includes three different types of tools which will be presented in this section. Once again, it is not always easy to put a tool into one of the categories, since they often use a combination of two or more approaches.

Graphical Editors [2.2.3.1]

Graphical Editors for application-specific graphics allow the designer to draw pictures of what graphics should look like. Using graphical editors, it is possible for the designer and non-programmer to use predefined primitive objects (like boxes, texts or lines) and to create parameterised objects which can then be generalised. The designer creates an example of the desired display which is then modified at run-time. It is important to understand that not the complete layout is specified at design time.

All the graphical editors (like Peridot, Lapidary or DEMO) are still at a research level. The basic idea is to provide the basis for generalisation, so that example objects can be parameterised to prototypes. In most cases those parameterised objects are then converted automatically into object-oriented procedures like those found in toolkits.

Menu Trees [2.2.3.2]

The main advantage of menu trees (also menu selection trees) is their simplicity in structure. There are several tools for creating trees (and for browsing the entire

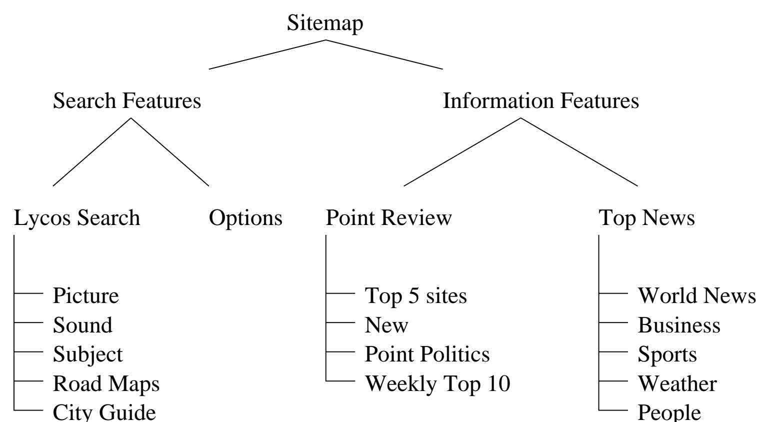


Fig. 2.24: Menu tree: Lycos search service called Sitemap (from [Shn97], p.251)

tree). Like a map, menu trees show both high and low level details; they are a good means of showing relationships and therefor are useful in checking consistency and completeness.

Fig. 2.24 shows the menu structure of Lycos (which is a popular search engine on the WWW) called Sitemap. Like a map, the system's structure is displayed, which leads to the one major disadvantage of menu trees: the problem in describing larger systems - the tree is too large to be displayed on a simple sheet of paper.

2.3.2 *Informal Methods - Building Tools [3]*

This section gives an overview of interface building tools. These building tools have one thing in common: they are all graphical and there is no limit in the types of interfaces which can be created.

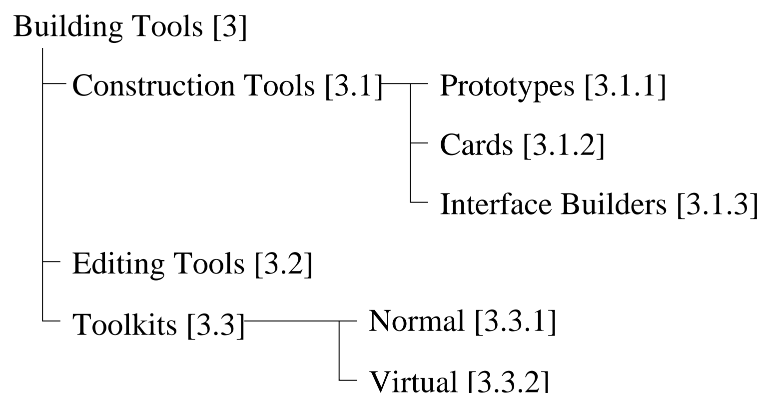


Fig. 2.25: Interface building tools

Fig. 2.25 shows the tool classes that help the designer in creating quick pictures of the look of the system and others that are powerful programming languages, including extensive toolkits.

The terminology varies from vendor to vendor. One characteristic is the support the tool provides for the *design* of the interface of the users' application. If the tool can only be used for composing an interface, then it belongs to the class of construction tools. If the tool helps the designer in developing an interface for his application, it then belongs to the subclass of software engineering tools (editing tools) or to the subclass of toolkits.

Construction Tools [3.1]

Construction tools (design tools) are the first class of building tools. There are three main types of design tools. The first and most trivial one is called Prototypes. The second one is called Cards, whose main duty is to create sketches of the interface as early as possible. The third class are interface builders which provide much more flexibility in user interface design.

Prototypes [3.1.1]

Prototype tools are only used to generate a quick example of what will be presented on the screen. These tools, in general, are not able to generate a user interface for the program. They just show some aspects of the user interface, which means that all or at least some parts are not working.

The major disadvantage of prototype tools is that they cannot generate actual code for the application's user interface. After prototyping, the user interface must be coded again. Another problem is that programmers implementing the real user interface often ignore all or some aspects of the prototype which leads to inconsistency.

Cards [3.1.2]

```
on mouseUp
    play "boing"
    wait for 3 seconds
    visual effect wipe left very fast to black
    click at 150, 100
    type "goodbye"
end mouseUp
```

Fig. 2.26: Hyper-Talk: event specification

The feature of this class of building tools is that the user interface is limited to a series of, mostly, static pages, sometimes called frames or cards. Each page is designed out of a set of widgets which are provided by the design tool and have been coded by hand. One of the most famous examples for a card based system is Apples Hyper-Card. Hyper-Card itself provides a scripting language called Hyper-Talk, which is a complete event language for writing short pieces of code that is executed when an input event occurs (see Fig. 2.26).

Other card-like applications are Microsoft's Powerpoint, a tool for creating slide-shows, or Asymetrix's Toolbook, a multimedia construction tool.

Interface Builders [3.1.3]

This third class of building tools (which is indeed very closely related to Cards) allows the designer to create interfaces as a part of larger user interfaces, and is called interface builder. The designer is able to select widgets out of predefined libraries and to place them on the screen via some pointing device. Properties can be defined afterwards and even some kinds of sequences can possibly be defined. Examples for this class of tools are parts of Microsoft's Windowing Toolkit, Apples MacApp or UIMX for X Windows or Motif.

Although Interface builders are becoming more and more popular, not all user interfaces can be designed in that way. These kinds of tools will not work with applications which use graphical areas (CAD, drawing programs or visual language editors).

Editing Tools [3.2]

There exists a large class of design tools (popular programs like Microsoft Visual Basic or Symatec Cafe) which all have easy to use interfaces for dragging buttons, labels or fields onto a workspace. They all enable the programmer to write a code in a more or less simple language (a scripting language, Basic or Java for example).

The visual editors reduce design time to a minimum, but extending the set of widgets takes a lot of programming skills. These kinds of tools are very popular, and there are many libraries of widgets for sale - another typical characterisation for this class of tools.

Toolkits [3.3]

Toolkits are libraries of interaction techniques that can be called by application programs. An interaction technique is a way of using an input device to input a certain type of value [Mye97]. Typical elements are menus, buttons or dialogue boxes. Toolkits can only be used by programmers, as only they have procedural interfaces.

Normal Toolkits [3.3.1]

A major advantage of ordinary toolkits is that the final user interface looks and acts in a very similar way in different applications (that use the same toolkit). But there are at least three main disadvantages of toolkits:

1. The style of interactions is limited to those that are provided. New styles need heavy programming.

2. Toolkits are often difficult to use, as there are too many procedures or ways of using them.

3. Toolkits are very poor in supporting consistency, as they do not provide rules for the correct use of their interaction techniques.

Depending on the level where toolkits are implemented in the whole system, toolkits can be exchanged easily. Looking at Macintosh, the toolkit is at a low level, which means that the window manager itself can use the toolkit for its user interface. Looking at X, for example, toolkits are at the top level, which means that here it is easier to switch the toolkit (Xtk or Interviews, to mention two of them).

A toolkit itself may be divided into two main layers:

1. The widget set as a collection of widgets and

2. the intrinsic layer where the implementation of the widgets is specified.

The Motif look and feel, for example, has been implemented on at least three different intrinsics (Xtk, Interviews and Garnet).

There are only small differences between the toolkits, but users still have troubles in converting their software from Motif to Microsoft Windows or OpenLook. In this case virtual toolkits can help.

Virtual Toolkits [3.3.2]

The key idea is to hide the differences among the various toolkits by providing virtual widgets which (then) can be mapped into the widgets of each toolkit. The most important fact is that the code itself can be linked to different toolkits and will run without change. The interface of a virtual toolkit is still procedural, and provides only functions that appear in common toolkits. XVT, 1991, is an example of a virtual toolkit, hiding the differences of Motif, OpenLook, Macintosh, MS-Windows and OS 2-PM toolkits.

2.4 Summary

Summing up all knowledge about the different ways of specifying user interfaces, one will find, that every method has its own advantage and disadvantage.

Context free grammars are good in describing the syntax of systems (BNF, Multiparty BNF, XYACC). They are very compact (TAG), but they lack specification in graphic interfaces and highly interactive applications (including interrupts).

The **denotational approach** is applicable to both character based and graphical based interfaces. Every level of abstraction is supported. The problem is that the specification is not easy to read, the overall structure cannot be seen and that the user has to be an expert in denotational semantics.

The **User Action Notation** is a notation focusing on the users' tasks. Structure and context are combined in a sound form of notation. Nevertheless, the UAN is not easy to learn and lacks specification in the relationship across tasks and in the interrupt behaviour.

Automatic generation (ITS) leads to a consistent interface and works best in designing menus or dialogue boxes. Its disadvantage is that the interfaces are generally not good enough and that the designer has to learn a special purpose language.

Event Languages focus on events that are sent to event handlers. It is possible to define interactions which are available concurrently. Its disadvantage is, though, that a correct code is to be written, as the flow of control is not easy to be localized.

Declarative Languages deal only with the information which is passed between the user and the system. The user needs not worry about the sequence of events. Nevertheless, only a small number of interfaces is supported and there exists no possibility of extending those systems.

Constraint Languages provide a good method in describing relationships between objects in user interfaces. They lack the debugging of the system and need an efficient run-time system.

Transition diagrams are widely used as a basis for representing the system's structure. They are excellent in representing the sequence of events (or inputs). Their major problem is that they become unwidely (large numbers of arcs) when the system grows.

Petri Nets are able to cover all aspects in describing user interfaces and dialogues and have a rich mathematical background. They are a formal but also graphically appealing language. It is also possible to model concurrent systems. Furthermore there are several tools (shareware and freeware) for building, analysing and simulating Petri Net systems existing.

Informal tools are most suitable for a fast interface design, but they are not really useful in the users' support for consistent interfaces. Construction tools and

toolkits provide only fixed types of interaction elements. Editing tools have easy to use interfaces and, depending on the tool, are extendable in style and interaction types. Nevertheless, defining the interface between the graphical objects and the application itself is hard work.

3. PETRI NETS

Look at every path closely and deliberately.
Try it as many times as you think necessary.
Then ask yourself, and yourself alone, one question ...
Does this path have a heart?
If it does, the path is good; if it doesn't, it is of no use.

Carlos Castaneda, The Teaching of Don Juan

In the year 1962 Carl Adam Petri was the first to formally define the Petri Nets language [Pet62], which was the beginning of the development of various extensions of State Transition and Predicate Transition networks. This chapter gives a short overview of the various types of Petri Nets and presents one kind of Petri Nets, Coloured Petri Nets, as a basis for analysing dialogue systems.

3.1 Motivation

As mentioned before, the history of Petri Nets started with the PhD thesis of Carl Adam Petri. Since then thousands of publications have been written.

However not only papers about Petri Nets exist [1] - there are many conferences, workshops and other events about this topic. The most important ones are:

1. The yearly "PN", the International Conference on Application and Theory of Petri Nets.
2. The "PNPM", the International Workshop on Petri Nets and Performance Models, which is held every two years.

Other Petri Net events include:

1. "AWPN" - Algorithms and Tools for Petri Nets,
2. "HPC" - High Performance Computing,

[1] Probably one of the best overviews can be found in the World Wide Web under "http://www.daimi.aau.dk/ petrinet/". (May, 1999)

3. "EKA" - Symposium on the Development and Operation of Complex Automation Systems,
4. "FM" - the World Congress on Formal Methods in the Development of Computing Systems and
5. "CPN" - the Workshop and Tutorial on Practical Use of Coloured Petri Nets and Design/CPN.

Since Jan. 3, 1996, there have been activities to create a standard for high level Petri Nets within the International Organisation for Standardisation (ISO). The project is called 7.19.3 (Petri Nets), the standardisation phase will last until February 2001.

There are a number of tools available, helping in the analysis and design of Petri Nets. At the moment there are more than 40 tools available, all containing different components and different support for various dialects and platforms. One freeware tool, Design/CPN from the CPN group at the Aarhus University in Denmark, will be presented later.

3.2 The World of Petri Nets

This section gives a short overview of existing Petri Nets variants (in order to argue about choosing Coloured Petri Nets as a basis for analysing dialogues). The classification uses a graphical kind of representation where sub-nodes denote a restriction to the more general parent nodes. Each node represents one class of Petri Nets. (See Fig. 3.1)[2].

Petri Nets are usually divided into three basic classes:

1. Net level 1.
   The characteristic feature is that they use boolean tokens, and the places are marked by not more than one token.
2. Net level 2.
   Nets of level two contain integer tokens, which are not structured and represent counters.
3. Net level 3.
   The tokens are now high-level tokens, which indicated that places are marked by structured tokens, which represent information as well as data types.

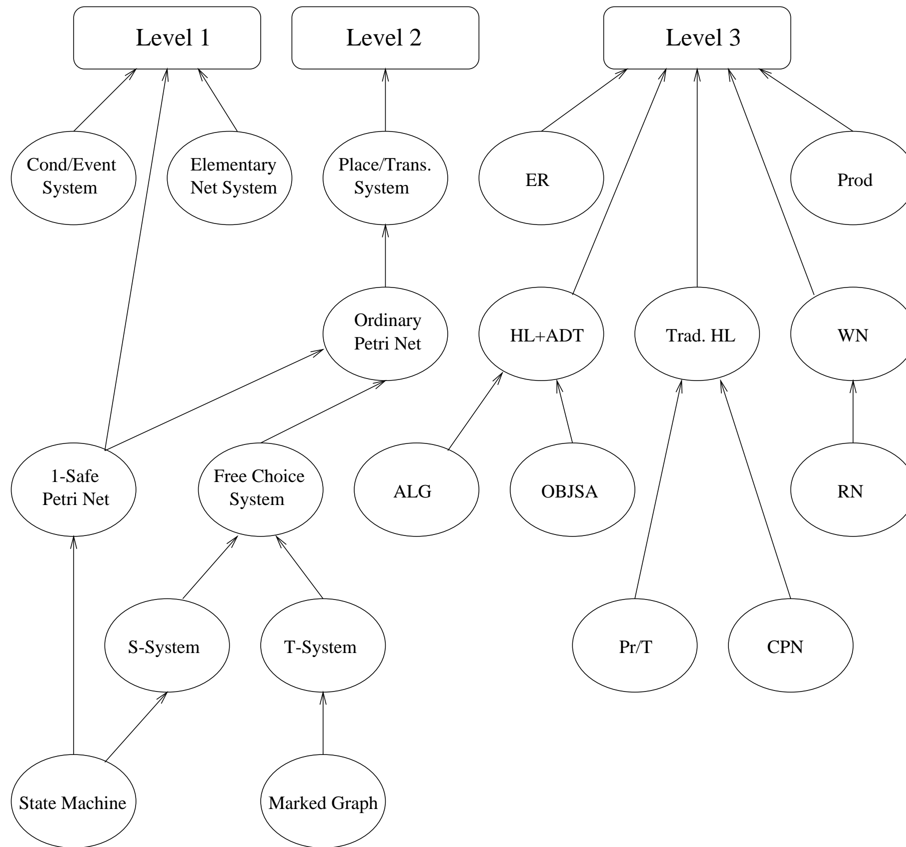


Fig. 3.1: Petri Nets: a general classification

Fig. 3.1 gives an overview of the most important classes of Petri Nets. Appendix C will present a short description of all mentioned classes. Important extensions to these general Petri Net classes have been developed in the last years. The most interesting ones are:

- Timed Petri Nets where time can be associated with tokens, transitions and places.
- "Axiomatic Architecture Description Language" (AADL) Nets where the transitions are labeled with the transformations on state space variables of the "Very High Speed Integrated Circuit (VHSIC) Hardware Description Language" (VHDL) implementation. The places can be annotated with the predicates on state space.

[2] This figure was designed following the classification of Monika Trompedeller at (http://www.dsi.unimi.it/User/Tesi/trompede/petri/home.html. May, 1999).

- Alternate Mark Inversion (AMI) Nets, a high level net model with temporal extensions. Extensions are used for queued places, timed transitions, priorities associated with transitions and inhibitor arcs.
- Communication Time Petri Nets for the specification of complex time-critical systems as a set of message passing Time Petri Nets (consisting of elementary channels and transmitting conditions and receiving events).
- Simulation Nets. They are ordinary Petri nets extended with time, inhibitor arcs, probabilities associated with the branches of input arcs, coloured tokens, arc weights and labels, stochastic functions, queues and interrupts. The firing rule is that an enabled transition must fire.

There are more than 20 recently developed extensions left, each with its own strength or weakness. The description of all extensions is not part of this work, but the publication of Bernadinello and Cindio [BC92] will be a good point to start with.

As it has turned out, Petri Nets of level one and two have not been chosen, as they use boolean tokens which do not have any structures. As the main goal of this work is to simplify the dialogue system, abstraction can be done by using coloured tokens. Probably all nets of level three could have been used, but Coloured Petri Nets seemed to be most appropriate. Section 3.5 will argue about the benefits of Coloured Petri Nets.

3.3 Informal Introduction

Since the beginning in 1962, Petri Net variants and their graphical representation have rapidly changed. Generally a Petri Net is a model for the description and analysis of non-deterministic, concurrent processes (see [Eng93]). Among the various kinds of Petri Nets, Place Transition Nets (PT Nets) are mainly known. Formally, this kind of Petri Net is a graph with the following properties:

- The net consists of a directed graph.
- The net has two kinds of nodes which are called places and transitions.
- Every arc connects a place with a transition, two arcs between one state and one transition are forbidden.
- the net does not contain isolated arcs or repeated nodes.

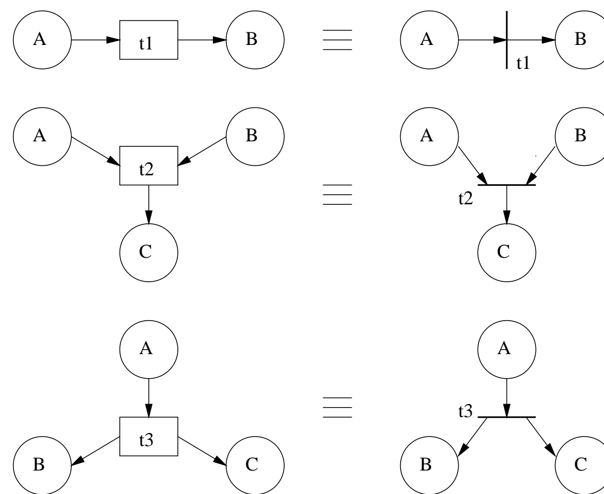


Fig. 3.2: Examples of simple PT nets

Fig. 3.2 shows that there are two kinds of graphical representation for PT Nets. At the beginning, the graphical notation at the right side (with a kind of "line" for a transition) was in common use, whereas nowadays the notation at the left side is common ([Zus80]). To improve the readability, net inscriptions (such as names of places) are sometimes written into, and sometimes written outside places. As design tools are getting more and more common, inscriptions are mostly written inside places and transitions.

Petri Nets are state and action oriented at the same time. States are indicated by means of ellipses (or circles) which are called places. The names of the places have no formal meaning (A,B,C in the example before). Actions are indicated by means of rectangles which are called transitions. In this case, the names (t1, t2 and t3) do not have a formal meaning, either.

As a more complex example let us consider a resource allocation system, since this example is very popular in literature. Let us assume that the system consists of two processes (p and q) which share a common pool of resources (R, S and T). Each process is cyclic and during a cycle, the process needs to have an exclusive access to the resources.

Each place in a PT Net may contain a varying number of small black dots, which are called tokens. A distribution of tokens on places is called a marking, the initial distribution of tokens is called initial marking. Fig. 3.3 shows a representation of the resource allocation system with an initial marking M_0.

As you can see, four of the places (Bp, Cp, Dp and Ep) represent the four possible states of the process p, and the five remaining places (Aq, Bq, Cq, Dq and

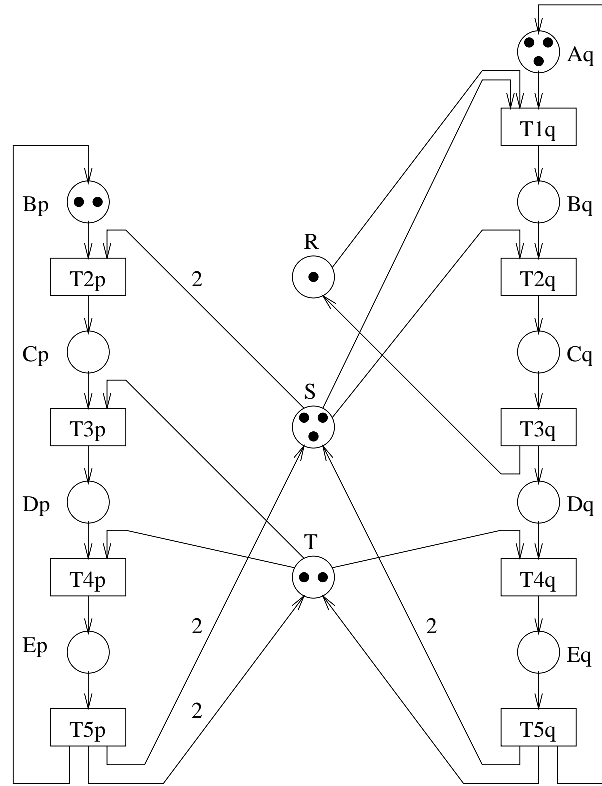


Fig. 3.3: PT net describing a simple resource allocation system (with the initial Marking M_0) (from [Jen97], p.4)

Eq) represent the states of the process q. The two tokens on Bp tell us that there are two p processes and they both start in state Bp. As you can see, there are also three q processes waiting for their start at state Aq. The tokens on the resources R, S and T show us that there are one R resource, three S resources and two T resources available. The net also consists of nine actions, called transitions (T2p to T5p and T1q to T5q) which represent the possible actions of the p and q processes.

As mentioned before, a PT Net also contains a set of directed arrows, which are called arcs. Each arch connects a place with a transition or a transition with a place. It never connects two nodes of the same kind. The integer that is connected to an arc is called "arc expression". By convention, arc expressions which are equal to 1 are omitted. A node A is called the "input node" of another node B, iff (where "iff" stands for "if and only if") there is a directed arc from A to B. An output node is defined in the same way.

The behaviour of a PT Net can be explained via a game board where the tokens are markers. Each transition represents a move in the "Petri net game" (a nice com-

parison from [Jen97]). A move is only possible, iff the number of tokens available at each input place is equal to or higher than the number defined by the arc expression. When revisiting Fig. 3.3, we find that transitions T2p and T1q are enabled. All other transitions are disabled, because there are too few tokens on some (or all) of their input places.

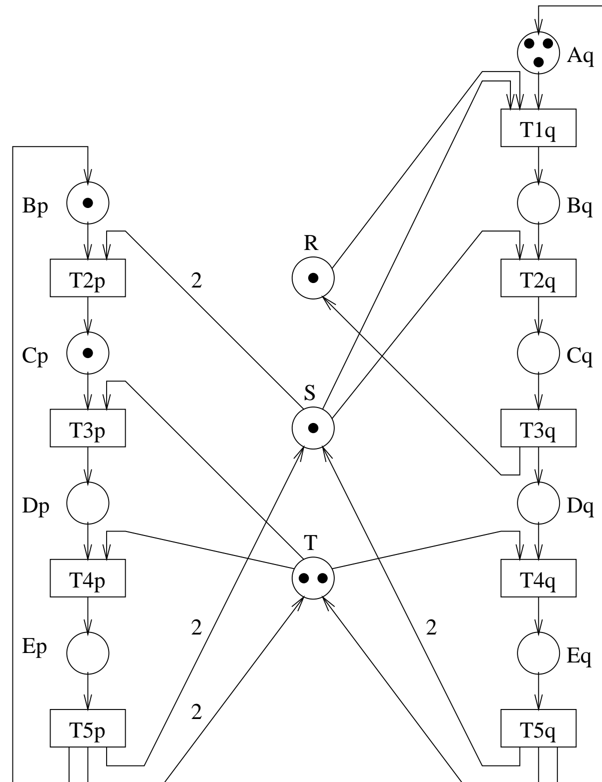


Fig. 3.4: PT net describing a simple resource allocation system with the marking M_1, reached from M_0 by T2p (from [Jen97], p.6)

The most important fact is that a transition MAY take place. If a transition is enabled, it is not always necessary that the transition should occur, which means that tokens are removed from the input places and added to the output places. Fig. 3.4 shows one possible marking that can be reached from the initial marking M_0 if the transition T2p fires.

3.4 Mathematical Model of PT Nets

This section presents the formal definitions of expression or phrases that will be used throughout the whole work. Using mathematics, the definition and the semantics

of Petri nets (and later Coloured Petri Nets) will be sound and unambiguous.

Let us start with the definition of the, up till now more or less often used term net ([Bau96]):

Definition 3.1: A Net. A net is a triple $N = (S, T, F)$ satisfying the following requirements:

$$S \cap T = \emptyset \text{ and}$$
$$F \subseteq (S * T) \cup (T * S).$$

S is a non-empty set of places, T is a non-empty set of transitions. Both the places and the transitions are called nodes. The elements of F (a relation) are called arcs.

As you can see in definition 3.1, a net is nothing else but a bipartite graph. A PT Net can be defined as follows (using the classical definition method - it is important to know that there is also a definition of PT Nets which does not include a capacity function. This is due to the fact that every PT Net with capacities can be transformed into an equivalent PT Net without capacities [Jen97]).

Definition 3.2: Place Transition Net. A Place Transition system is a six-tuple $\Sigma = (S, T, F, K, W, M_0)$ where

1. (S, T, F) is a net where S is the set of places and T the set of transitions. F represents the set of arcs.
2. $K : S \rightarrow \mathbb{N}^* \cup \{\infty\}$ is a capacity function which expresses the maximal number of tokens each place can contain.
3. $W : F \rightarrow \mathbb{N}^*$ is a weight function (arc expression)which expresses how many tokens flow through them at each occurrence of the involved transition.
4. $M_0 : S \rightarrow \mathbb{N}$ is an initial marking function which satisfies: $\forall s \in S : M_0(s) \leq K(s)$

As an example let us look at a simple producer/consumer system as shown in Fig. 3.5. The definition of the system can be seen in table 3.1.

Definition 3.3: Transition in PT Nets. A transition t is enabled at a marking M iff

$$\forall s \in S : W(s, t) \leq M(s) - W(t, s)$$

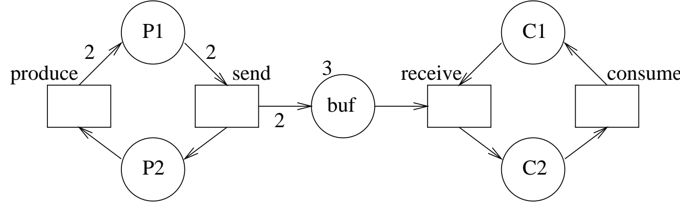


Fig. 3.5: PT net: producer consumer system

$S = \{p_1, p_2, buf, c_1, c_2\}$
$T = \{produce, send, receive, consume\}$
$F = \{(produce, p_1), (send, buf), (send, p_2), (recive, c_1), (consume, c_2),$
$(p_1, send), (p_2, produce), (buf, receive), (c_1, consume), (c2, receive)\}$
$K(buf) = 3$, the capacity for all other places is assumed to be infinite
$W(produce, p_1) = W(p_1, send) = W(send, buf) = 2$,
the other arc weights are assumed to be equal to 1
$M_0(p2) = 1, M_0(buf) = 2, M_0(c2) = 2$

Tab. 3.1: PT net: definition of a simple producer/consumer system

If t occurs, it yields a new marking

$$M‘(s) = W(s,t) + W(t,s) \forall s \in S$$

denoted by

$$M[t\rangle M‘.$$

The Forward Reachability Class, denoted by $[M_0\rangle$ is the smallest set of markings of Σ so that

1. $M_0 \in [M_0\rangle$
2. if $M_1 \in [M_0\rangle$ and $M_1[t\rangle M_2$ for some t $\in T$, then $M_2 \in [M_0\rangle$

The above definition leads to the result that a marking M'' can be reached from a marking M' if there exists a finite occurrence setting (a sequence of markings and steps $M_0[t1\rangle M_1[t2\rangle M_3[t3\rangle$ ...) with M' as a start marking and M'' as end marking and there exists a sequence of steps $t_1, t_2, .., t_n$ so that $M'[t_1 t_2 ... t_n\rangle M''$.

The step semantics can be defined as follows:

Definition 3.4: Concurrent behaviour of PT Nets. A subset $T' \subseteq T$ is enabled at a marking M iff:

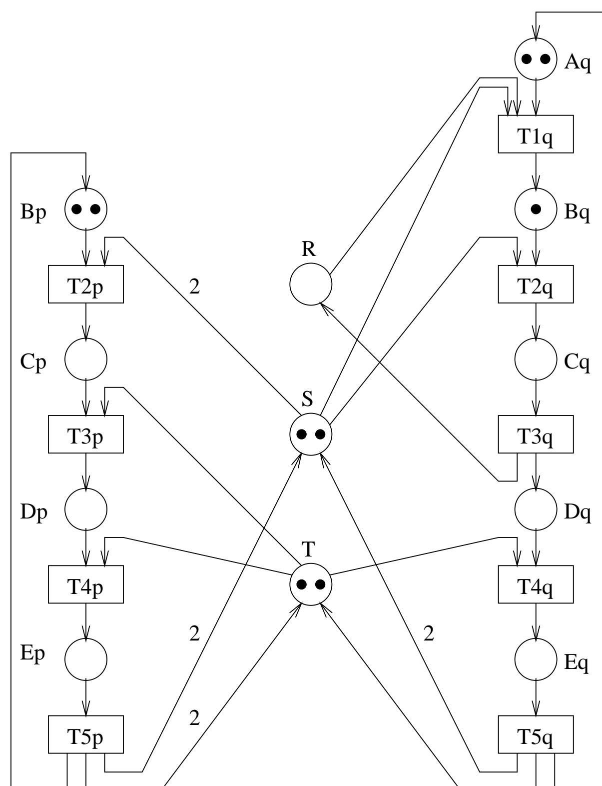


Fig. 3.6: PT net describing a simple resource allocation system with the marking M_2, reached from M_0 by transition T1q (from [Jen97], p.7)

$$\forall s \in S : \sum_{t \in T} W(s,t) \leq M(s)$$

and

$$M(s) + \sum_{t \in T} W(t,s) \leq K(s)$$

As mentioned before, we can see that two transitions T2p and T1q in Fig. 3.3 are enabled at the same time (concurrently enabled). This indicates that both of the transitions may occur at the same time. Step $S_1 = \{T2p, T1q\}$ is enabled in M_0. Fig. 3.4 and Fig. 3.6 show the two different markings.

In general, each possible step corresponds to a multi-set. A multi-set is analogous to a set, but it may contain multiple occurrences of the same element. If we add two further tokens to place S, the multi-set of step one (with the initial marking M_0) will be:

$$S_2 = \{T2p, T2p, T1q\} \tag{3.1}$$

The multi-set in 3.1 tells us that there are two occurrences of T2p and one occurrence of T1q and that there are no other transitions enabled. By convention, elements with zero coefficients are omitted and between a coefficient and an element an ' is inserted. Thus 3.1 can be written as:

$$S_2 = 2`T2p + 1`T1q \tag{3.2}$$

We can now move on to the definition of a multi-set (following [Jen97]):

Definition 3.5: Multi-set. A multi-set m, over a non-empty set S, is a function $m \in [S \to \mathbb{N}]$. The non-negative integer $m(s) \in \mathbb{N}$ is the number of occurrences of element s in multi-set m. We usually represent multi-set m by the formal sum:

$$\sum_{s \in S} m(s)`s$$

By S_{MS} we denote the set of all multi-sets over S. The non-negative integers $\{m(s)|s \in S\}$ are called the coefficients of multi-set m, and $m(s)$ is called the coefficient of s. An element $s \in S$ is said to belong to multi-set m iff $m(s) \neq 0$ and we then write $s \in m$.

3.5 Coloured Petri Nets

With the short presentation of Petri Nets it will be clear that they are a sound form of description and are excellent in analysis as well as the description of communication and resource sharing. The problem with ordinary Petri Nets (level one and two) and complex, interactive systems is that there is no data concept behind the model and the model becomes large and practically unreadable. The disadvantage of high level Petri Nets of the 1970s was that they had no hierarchical concept or interfaces. In the 1980s hierarchical Petri Nets came up and especially the Coloured Petri Nets became popular. This section gives reasons for choosing Coloured Petri Nets and explains the mathematical background.

3.5.1 Motivation

Coloured Petri Nets (CP Nets) are Petri Nets of level three. There are six main reasons why CP Nets seem to be best for describing and analysing dialogue structures:

1. CP Nets offer a sound graphical representation and a hierarchical way of description. The notation of states, actions and flow are easy to understand and very often intuitively used in system design. The hierarchy option allows the user to break down a large, almost unreadable system into small and easy to understand parts.

2. They have a well-defined semantics which unambiguously defines the behaviour. Therefore it is possible to use simulators and formal (and also automatic) analysis tools.

3. They offer an explicit description of states and actions so that the point of focus can easily be changed during the work.

4. CP Nets have a semantics that is built upon true concurrency instead of an interleaving approach. This makes it possible to have two actions in the same step, which is a more natural way for a human being's thinking about concurrency.

5. CP Nets are extendable to time concepts.

6. There is a good support for drawing, simulation and formal analysis of the net.

The definition of CP Nets is very short and is based only on a few primitives. This makes CP Nets easier to use and to read. Furthermore, there is an excellent description of data manipulation, control and synchronisation.

The resource allocation system in Fig. 3.2 shows two processes which are very similar and use the same resources. The disadvantage of specifying two separate subnets (for each process) is not important for small systems, but as the system grows (image more than 10 processes), the description becomes unwidely and unreadable.

The solution is to equip every token with the kind of data values which are then called **token colour**. These values can be of any complex type (text strings, integers, records) and their use is similar to that in any programming language. The net in Fig. 3.7 describes the same resource allocation system as in Fig. 3.3, but is a more compact form and is extended by a counter representing the number of cycles each process has completed.

As you can see in Fig. 3.7, every place has its own type, called **colour set**. This indicates that all tokens at this place must have the token colour appropriate to this place. By convention, token sets are written in italics. In Fig. 3.7 you can see that the places A to E belong to the colour set of type P and that the places R to T have the colour set E assigned to them.

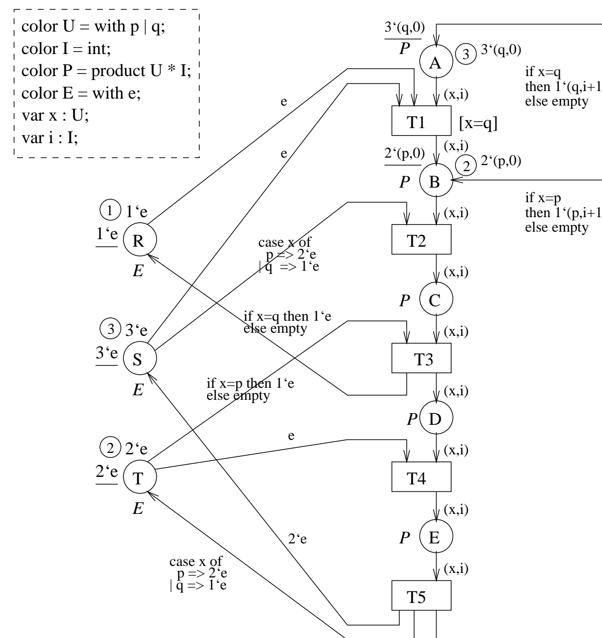


Fig. 3.7: CP net: net describing the simple resource allocation system of Fig. 3.3 (from [Jen97], p.10.)

To understand the types of colours and variables used in the net, a **CPN declaration** section is added to the net. The section is put into a dashed box in the upper corner of the figure. Many tools use a dialect of ML for the declaration (**CPN ML**, for example. The syntax of CPN ML can be found in appendix F). In the example you can see that the token colour of the places A to E (this is colour P) is a Cartesian product of two other colours, namely U and I. The first element U is either p or q (enumeration type in CPN ML) representing the process, the second element is an integer (the colour set I is declared to be the standard type int), representing the cycle counter.

As with PT Nets, the distribution of tokens is called a marking. Evaluating the **initialisation expression** leads to the initial marking of the net. This expression is put next to the responding place and is underlined. As we can see in the example above, place A gets three (q,0) tokens at the beginning. Multi-sets are used for marking the colour set attached to the place.

To indicate the **current marking** of a CP Net, a small circle with an integer inside it and a text string attached to it is used. The integer represents the number of tokens available at this place and the text string is the multi set representing the individual token colour. As you can see it at the initial marking in Fig. 3.7 there are two (p,0) tokens at place B.

As with PT nets, the dynamic of the net is described by the transitions. To describe the influence of input tokens to output tokens (in connection with a transition), **arc expressions** are introduced. Arc expressions evaluate to multi-sets. Transition T2 in the example has three arc expressions around it. (x, i) appears twice and "$case\ x\ of\ p \Rightarrow 2`e \mid q \Rightarrow 1`e$" appears once. These three expressions have two variables (x and i) and three elements of colour sets (e of colour set E and p and q from the colour set U) representing some kind of constants. As you can see transition T2 moves a token from place B to place C without changing the colour (the arc expressions are identical). The third expression indicates that a multi-set of tokens is removed from place S by evaluating the arc expression.

The calculation of the enabling and occurrence of the transition is quite easy. As transition T2 has two variables (x and i), colours of the corresponding type (elements of the colour sets U and I) have to be bound to them. This **binding** can be done in many ways, one possibility is to use binding $b_1 = \langle x = p, i = 0 \rangle$, or binding $b_2 = \langle x = q, i = 2 \rangle$. For each binding (possible or not possible) it must be checked, if the transition is enabled or not. For binding b_1 the two input arc expressions at T2 will lead to $(p, 0)$ *and* $2`e$ which indicates that $b_1$ is enabled, as each of the input places (B and S) contain the necessary amount of tokens. For binding $b_2$, the arc expressions are evaluated to $(q, 2)$ *and* $1`e$. We can see that b_2 is not enabled, as there is no (q,2) token at place B. When a transition is enabled, it MAY **occur**, which indicates, that the tokens are removed from the input places, and the tokens evaluated at the outgoing arc expression are added to the output places. The pair (t,b), where b is a transition and b is a binding for t, is called a **binding element**. The binding element $(T2, b_1)$ is enabled in initial marking, or in the same way binding element $(T1, \langle x = q, i = 0 \rangle)$ is enabled in marking M_0.

When looking at transition T1 we can see that in addition to the arc expressions, there is a boolean expression $[x = q]$, a **guard** connected to it. Its purpose is to define additional constraints which must be fulfilled in order to enable the transition. In the example, the guard guarantees that only q processes can move from A to B. The guard could have been omitted, as the arc expression of the incoming transitions filter the p and q processes. In large systems a guard is very useful, and improves readability.

Using different colour sets, declarations and arc expressions, it is possible to change the level of the abstraction of a system. Fig. 3.8 shows the same Coloured Petri Net as in Fig. 3.7, and is much more compact, in the sense that most of the information is put into the declarations and the net inscriptions. Jensen shows in his books [Jen97] and [Jen95], that it is possible to convert every PT Net into a corresponding CP Net. These transformations can be specified by formal transformation rules, and modellers and checkers can perform those transformations. In Jensen's

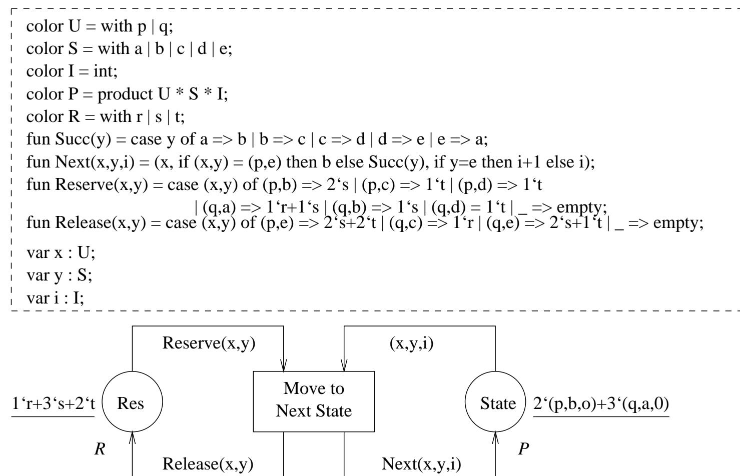


Fig. 3.8: CP net: very compact net describing the simple resource allocation system of Fig. 3.3 (from [Jen97], p.20)

opinion it is a typical approach to start with a large and broad structure and make it denser later on. It is up to the designer to choose the correct level of abstraction - one more benefit of CP Nets.

3.5.2 Mathematical Background

This section gives a mathematical definition of non-hierarchical Coloured Petri Nets. It is not necessary to fix the syntax in which the modeller writes net expressions, it is only necessary to assume that such syntax together with some well defined semantics exist. With the following abbreviations (Jensen uses in his book [Jen97]), it is possible to describe CP Nets:

- Type(v) denotes the type of a variable, v.
- Type(expr) denotes the type of an expression, expr.
- Var(expr) denotes the set of variables in an expression, expr.
- A binding b of a set of variables, V - associating with each variable $v \in V$ an element $b(v) \in Type(v)$.
- $expr\langle b\rangle$ denotes the value obtained by evaluating an expression, expr in a binding b. Var(expr) is required to be a subset of the variables of b, and the evaluation is performed by substituting for each variable $v \in Var(expr)$ the value $b(v)$ determined by the binding.

1.	$\Sigma = \{U, S, I, P, R\}$.
2.	$P = \{Res,\ State\}$.
3.	$T = \{``Move''\}$.
	(* Where "Move" stands for the transition "Move to Next State" *)
4.	$A = \{ResToMove,\ MoveToState,\ MoveToRes,\ StateToMove\}$.
5.	$N(a) = (Source, Dest)$ if a is in the form SourceToDest.
6.	$C(p) = \begin{cases} P & : \ p = State \\ R & : \ otherwise \end{cases}$
7.	$G(t) = true$
8.	$E(a) = \begin{cases} Reserve(x,y) & : \ a = ResToMove \\ Release(x,y) & : \ a = MoveToRes \\ Next(x,y,i) & : \ a = MoveToState \\ (x,y,i) & : \ otherwise \end{cases}$
9.	$I(p) = \begin{cases} 1`r + 3`s + 2`t & : \ p = Res \\ 2`(p,b,0) + 3`(q,a,0) & : \ p = State \end{cases}$

Tab. 3.2: CPN: tuple Σ representing the Coloured Petri Net specification of the resource allocation system in Fig. 3.8.

With this, it is now possible to define CP Nets as a tuple. To make it easier to understand the following definition, Table 3.2 presents the resource allocation system in Fig. 3.8 as a tuple.

Definition 3.6: (Non-hierarchical) Coloured Petri Net. A CP Net (CPN) is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ where

1. Σ is a set of types called Colour Sets which are finite and non-empty.

2. P is a finite set of places.

3. T is a finite set of transitions.

4. A is a finite set of arcs, such that $P \cap T = P \cap A = T \cap A = \emptyset$. For pragmatic reasons $P \cup T \cup A = \emptyset$ is allowed, too.

5. $N : A \to P \times T \cup T \times P$ is a node function.

6. $C : P \to \Sigma$ is a colour function.

7. G is a guard function $G : T \to Bool$. It is defined from T into expressions such that: $\forall t \in T : (Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \Sigma)$.

8. E is an arc expression function. It is defined from A into expressions such that $\forall a \in A : (Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma)$ where $p(a)$ is the place on $N(a)$ and C_{MS} denotes the set of all multi-sets over C. This indicates that the result of an arc expression has the same colour as the place which the arc belongs to.

9. I is an initialisation function. It is defined from P into expressions such that $\forall p \in P : (Type(I(p)) = C(p)_{MS} \wedge Var(I(p)) = \emptyset)$

It is assumed that each colour set has at least one element. The places, transitions and arcs are described by three sets which are finite and pairwise disjoint. The net structure can be empty, which is in contrast to classical Petri nets. The guards can be a list of boolean expressions in the form $[Bexpr_1, Bexpr_2, .., Bexpr_n]$ as this is short for $Bexpr_1 \wedge Bexpr_2 \wedge \ ... \ \wedge Bexpr_n$. This indicates that the binding must fulfil each of the boolean expressions. Arc expressions can have expressions $expr$ of type $C(p(a))$, as this is a shorthand for 1‘$(expr)$ - a multi-set with only one element.

Earlier we were told that a binding is used to bind colours to variables. A binding of a transition t is nothing else but a substitution that replaces each variable of t with its corresponding colour (each colour has to be of the correct type). Furthermore, it is necessary that the guard's evaluation should be true.

Definition 3.7: Binding in CP Nets. A function b is called binding of a transition, if $(\forall v \in Var(t) : \ b(v) \in Type(v))$ andalso $(G(t)\langle b \rangle)$.

With this it is possible to define transitions in CPN Nets.

Definition 3.8: Transition in CP Nets. A step Y is enabled at a marking M iff

$$\forall p \in P : \sum_{(t,b) \in Y} E(p,t)\langle b \rangle \ \leq \ M(p)$$

where a step Y is a non-empty binding distribution: $\forall t \in T \ : \ Y(t) \ \in \ B(t)_{MS}$. $B(t) = \{(c_1...c_n) \in BT(t) | G(t)(v_1 = c_1...v_n = c_n)\}$, where BT(t) is a binding type: $BT(t) = Type(v_1) \times Type(v_2) \times ... \times Type(v_n)$ for a transition $t \in T$ with variables $v_1, v_2, ...v_n$.

If a step is enabled, it may occur. This indicates that tokens at input places are removed and added to the output tokens of the occurring transitions. Arc expressions determine the colours and number of tokens.

Definition 3.9: Occurrence in CP Nets. If step Y is enabled at marking M_1 it may occur, which results in a change from marking M_1 to marking M_2, which is defined as follows:

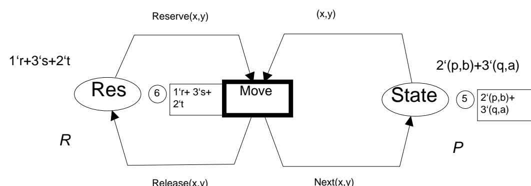
$$\forall p \in P : M_2(p) = (M_1(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle) \; + \; \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$$

The first sum represents the removed tokens, the second sum is called the "added" tokens. The occurrence of step Y tells us that M_2 is "directly reachable" from M_1, denoted by $M_1[Y\rangle M_2$.

Jensen shows that every non-hierarchical CP Net has exactly the same set of markings, steps and occurrence sequences as the equivalent PT Net or hierarchical CP Net and that they are in their behaviour equivalent.

3.5.3 Analysis

```
color U = with p | q;
color S = with a | b | c | d | e;
color P = product U * S;
color R = with r | s | t;
fun Succ(y) = case y of
  a => b | b => c | c => d |
  d => e | e => a;
fun Next(x,y) = (x,
  if (x,y) = (p,e) then b else Succ(y));
fun Reserve (x,y) = case (x,y) of
  (p,b) => 2`s | (p,c) => 1`t |
  (p,d) => 1`t | (q,a) => 1`r + 1`s |
  (q,b) => 1`s | (q,d) => 1`t |
  _     => empty;
fun Release (x,y) = case (x,y) of
  (p,e) => 2`s + 2`t | (q,c) => 1`r |
  (q,e) => 2`s + 1`t | _     => empty;
var x : U;
var y : S;
```



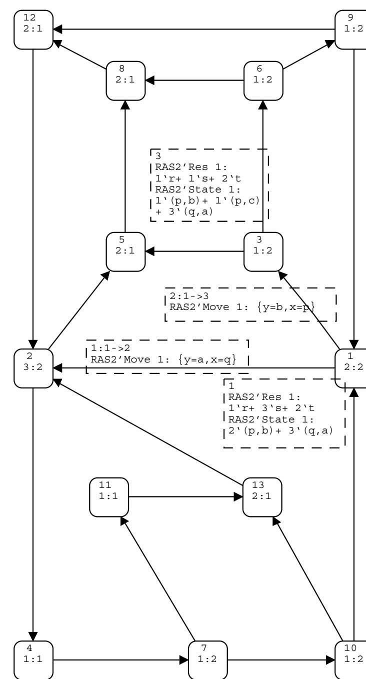


Fig. 3.9: CP net: occurrence set for the produce/consumer system of Fig. 3.8. Node and arc descriptors are toggled on for nodes 1 and 3

A major advantage of the precise definition is that CP Nets can be formally analysed. There are four ways to do the analysis:

1. Simulation.
   This method is very similar to debugging or program execution. CP Net models can be executed to filter out statistical information. This can be achieved by setting breakpoints and by summing up simulation results. Furthermore CP Nets can be extended with timed tokens or a global clock and time dependent behaviour can be analysed, too. For some kind of delays it is also possible to translate CP Nets into Markov chains (a type of stochastic model) which define an equation system leading to analytic solutions for the different performance values.

2. Occurrence Graphs.
   This second method is also called state spaces or reachability graphs. The basic idea is to construct a directed graph which has a node for each system state that is reachable. The graph has an arc for each possible state change. This graph can be very large, even for small systems, but its advantage is that it can be constructed and analysed automatically, and using equivalence classes, it is also possible to condense those graphs without loosing analytic information. Fig. 3.9 shows that the occurrence graph for the producer/consumer example exists of 13 different states, leading to 13 places (nodes) on the occurrence graph. For a better understanding, the node descriptors (for nodes 1 and 3) and arc descriptors (for the outgoing arcs for node 1) are added.

3. Place Invariants.
   The third method is called place invariants. This method can be compared with invariants in ordinary program verification. The user constructs a set of equations, which are then proved to be satisfied for all reachable system states. These equations are used to prove special properties of the modelled systems, for example the absence of deadlocks. As an example one can expect that places A to E always have five tokens, two which belong to the p processes, and three which belong to the q processes. This can be expressed by the following equation (PR is a projection function mapping multi-sets to multi-sets, M represents the set of all reachable markings and Q_c represents a function counting all occurrences of q in the marking):

   $$PI_P: \; PR_1(M(A)+M(B)+M(C)+M(D)+M(E)) = 2`p + 3`q \tag{3.3}$$

   $$PI_R: \; M(R)+Q_c(M(B)+M(C)) = 1`e \tag{3.4}$$

   With this (and other analogous defined equations) it is possible to argue about liveness or deadlocks.

4. Reduction.
   The last method is called reduction. The basic idea is to apply reduction rules

to the system, which do not influence the properties one wants to investigate (boundedness or liveness). If the reduction rules are sound (the correctness proof is often hard), the reduced CP Nets have the same properties as the original CP Net and can be investigated.

For those who are not familiar with the definition of liveness or boundedness, the following section will give you a short overview.

3.5.4 Dynamic Properties

The behaviour of individual CP Nets is characterised by their dynamic properties. For example, these properties answer the question if it is possible to reach a marking in which no step is enabled. It is rather difficult to verify dynamic properties, nevertheless, a small number of formal methods exist (see above).

The upper and lower bounds tell us the maximum and minimum number of tokens (of particular colours) which can arise in the net on a particular place.

Definition 3.10: Boundedness of CP Nets. X is a set of token elements $X \subseteq TE$ and n is a non-negative integer $n \in \mathbb{N}$.

1. n is an upper bound for X iff: $\forall M \in [M_0\rangle : |(M|X)| \leq n$.
2. n is a lower bound for X iff: $\forall M \in [M_0\rangle : |(M|X)| \geq n$.

X is bounded iff it has an upper bound.

The definition of the boundedness for token elements corresponding to particular place instances is defined in the same way.

A home marking is a marking to which it is always possible to return. A set of home markings is called home space.

Definition 3.11: Home properties of CP Nets. A marking $M \in \mathcal{M}$ (the set of all markings) and a set of markings $X \subseteq \mathcal{M}$ is given.

1. M is a home marking iff: $\forall M' \in [M_0\rangle : \; M \in [M'\rangle$.
2. X is a home space iff: $\forall M' \in [M_0\rangle : X \cap [M'\rangle \neq \emptyset$.

The liveness property tells us if a set of binding elements X remains active, which indicates that it is possible (for each reachable marking M') to find an occurrence sequence starting in M' and containing an element from X. For liveness it is only necessary for elements of X to become enabled, in fact, they must not be enabled at a special occurrence sequence.

Definition 3.12: Liveness of CP Nets. A marking $M \in \mathcal{M}$ and a set of binding elements $X \subseteq BE$ is given.

1. M is dead iff no binding element is enabled, which means that $\forall x \in BE : \neg M[x\rangle$.
2. X is dead in M iff no elements of X can become enabled, which means that $\forall M' \in [M\rangle\ \forall x \in X : \neg M'[x\rangle$.
3. X is life iff there is no reachable marking in which X is dead, which means that $\forall M' \in [M_0\rangle\ \exists M'' \in [M'\rangle\ \exists x \in X : M''[x\rangle$.

X is dead, iff it is dead in M_0. Nevertheless, live is not the negation of dead, each live set of binding elements is non-dead, but the opposite is not true.

The fairness property tells us how often the different binding elements occur.

Definition 3.13: Fairness of CP Nets. X is a set of binding elements $X \subseteq BE$ and $\sigma \in OSI$ an infinite occurrence sequence of the form $\sigma = M_1[Y_1\rangle M_2[Y_2\rangle M_3$. $EN_X(\sigma)$ denotes the total number of enablings and $OC_X(\sigma)$ denotes the total number of occurrences in σ.

1. X is impartial for σ iff it has infinitely many occurrences: $OC_X(\sigma) = \infty$
2. X is fair for σ iff an infinite number of enablings implies an infinite number of occurrences: $EN_X(\sigma) = \infty \;\Rightarrow\; OC_X(\sigma) = \infty$.
3. X is just for σ iff a persistent enabling implies an occurrence: $\forall i \in \mathbb{N}^* : (EN_{X,i}(\sigma) \neq 0 \;\Rightarrow\; \exists k \geq i : \; (EN_{X,k}(\sigma) = 0 \;\vee\; OC_{X,k}(\sigma) \neq 0))$.

The properties listed before are sufficient for many CP Nets. Nevertheless, some nets have binding elements, which are indefinitely repeated. In this case, the modeller has to remove certain elements to investigate the net for a special property.

An automatically generated statistics of the producer/consumer system can be found in the appendix D.2. As you can see, the occurrence graph consists of 13 nodes and 20 arcs. The statistics also contains information about the "Scc Graph" . Scc represents the set of all strongly connected components of an occurrence graph.

Definition 3.14: Scc graph of CP Nets. The nodes N of an occurrence graph should be given. A "strongly connected component" of an occurrence graph is a maximal set of nodes $c \subseteq N$, so that all elements of c are reachable from each other (in the occurrence graph). The set of all strongly connected components is denoted by Scc. The directed graph, which has nodes for each component $c \in Scc$ is called Scc Graph. There is an arc from component $c_1 \in Scc$ to component $c_2 \in Scc$ iff there are nodes $n_1 \in c_1$ and $n_2 \in c_2$ so that n_1 on the occurrence graph has an arc to n_2.

The major advantage of Scc graphs is that they are often very small and that important properties (home marking or liveness) can be deduced, too.

3.6 Design/CPN

There are many tools available, each with its own advantage or disadvantage. An excellent comparison of 40 of those tools can be found in [Stö98]. Design/CPN was chosen for the following reasons:

- Design/CPN supports CP Nets with and without time.
- The editor allows the construction, modification and syntax check of CP Net models.
- The tool package supports interactive and automatic simulation of CP Net models.
- An occurrence graph tool supports construction and analysis of occurrence graphs.
- There is support for hierarchical Petri Nets.
- Design/CPN supports CP Net models with complex data types and manipulations. Both can be specified using the functional programming language Standard ML.

The short "quiz" example will give an idea about how Design/CPN works. The quiz program guesses a natural number inside a given interval by asking questions from the operator. Tab. 3.3 shows an example of a typical dialogue.

```
- quiz (1,10);
Think of a number between 1 and 10
Is the number <= 5 ?  no
Is the number <= 8 ?  yes
Is the number <= 7 ?  no
The number is 8
val it = () :  unit
```

Tab. 3.3: Simple Quiz: typical human computer dialogue

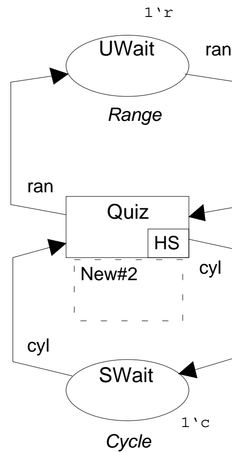


Fig. 3.10: Design/CPN: quiz model, general structure

The structure of the system is easy to define. Fig. 3.10 shows the principal structure of the communication between the system and the user: the user defines the range of the quiz and the system waits for the users input.

The first thing to do is to open a new project using Design/CPN and to draw the principal structure of the system. A picture of the working environment can be found in appendix F. As it is possible to draw hierarchical CP Nets with Design/CPN, this first structure can easily be refined. Using the editor, the transition Quiz is set to be a hierarchical substitution (and gets the HS tag assigned to it). The editor automatically generates a new page with all input and output arcs belonging to this transition. This page is automatically called "New#2" (but can be renamed if wanted). On this newly created page, the transition is specified in more detail.

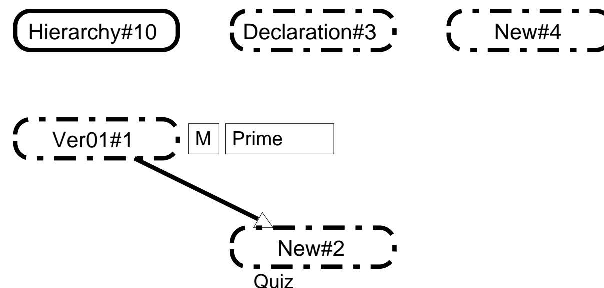


Fig. 3.11: Design/CPN: quiz model, hierarchy overview

Fig. 3.11 shows the principal hierarchical structure of the system. Every page is represented by its page name which is surrounded by a dashed and dotted line.

As you can see in Fig. 3.11, there are four pages defined. “Ver01#1” is the page with the general structure of the system (as you can see in Fig. 3.10). “New#2” page (page two) contains the refinement of the system. “Declaration#3” (page three) contains the region definition and page “New#4” (page four) contains the occurrence graph for that system. Pages three and four were created to improve the readability of the project.

As you can see in the hierarchical view, page two is a hierarchical substitution of page one. The text “Prime” denotes the page to be a prime page, which indicates, that this page is selected in the simulation phase. (Every page which is to be simulated must have the tag “Prime” page assigned to it.)

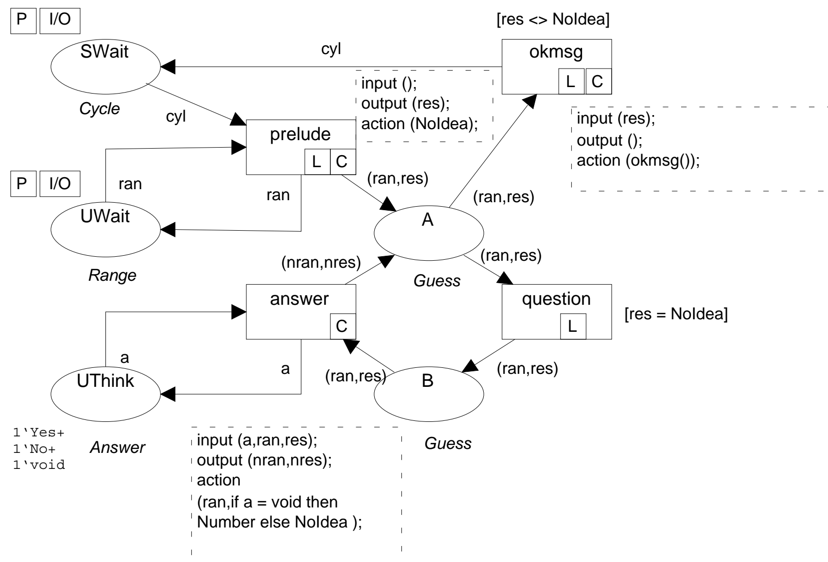


Fig. 3.12: Design/CPN: quiz model, detailed view

First of all the principal structure is defined. This can be done by placing States and Transitions onto the page. The next thing to do is to connect states and transitions. If this is done, net inscriptions are defined. Fig. 3.12 shows the redefined transition Quiz including all net inscriptions.

With the definition of the declarations (which you can see in Fig. 3.13, showing the content of page four) the system is fully defined and can be simulated and analysed. The definition is written using a dialect of Standard ML. First of all the colours and variables are defined. Furthermore, functions can be defined and these

```
color Cycle = with c;
color Answer = with Yes | No | void;
color Result = with Number | NoIdea;
color Range = with r;
color Guess =
  product Range * Result;
var res,nres : Result;
var a : Answer;
var ran,nran: Range;
var cyl : Cycle;

fun answer () = (
  case input_line (std_in) of
   "yes\n" => Yes
  | "no\n"  => No
  | _       => void
);
fun prelude () = (
  output (log, "Think of a number between ");
  output (log, "MIN");
  output (log, " and ");
  output (log, "MAX");
  output (log, "\n")
);
fun question () = (
  output (log, "Is the number <= ");
  output (log, "N");
  output (log, "?")
);
fun okmsg () = (
  output (log, "The number is");
  output (log, "N");
  output (log, "\n")
);
```

Fig. 3.13: Design/CPN: quiz model, colour set definition

functions can be assigned to transitions. The “L” in a transition indicates that a “Log Region” is assigned to it. This region produced an output which is appended to a predefined log stream. A “C” in a transition denotes a region with a code which can contain user declared and pre-defined constants, operations and functions. The code for both of them is defined in regions directly attached to the transitions.

The statistics in appendix D.3 shows us that the occurrence graph has four nodes (as you can also see in D.4), which indicates that the system has four different states. The Scc graph contains only one node (the initial marking). The statistic also tells us that the home property is satisfied for all markings, indicating that every marking can be reached again. The system has no dead marking, indicating that there is no state in this model, where the quiz example will not work. Finally all transitions are life and none are dead. When looking at the fairness property, one can see that

the answer and question transitions (sets of binding elements) are impartial, which indicates that there can be infinitely many occurrences. All other transitions are fair.

Working with Design/CPN has also some disadvantages:

1. It is not easy to port a model from one platform to another. I have used the Linux version of Design/CPN to model the quiz example. It was not possible to fully load this model using the Solaris version. Design/CPN provides an interchange format, but with this you can only export one page and not a complete hierarchy of pages. All connections get lost.

2. It is also tricky to move a model from one directory to another. Design/CPN assumes the ML heap image to be on the same place and if it does not find it, the system terminates with a core dump.

3. The user interface sometimes reacts in a way not expected. Some hot keys do not work, neither under Linux nor under Solaris. Sometimes a command (for example, "information about a node") works, sometimes not.

4. Verification and the building of the occurrence sets need a lot of process power (and time).

Nevertheless the tool was of great help in designing different interface models and to argue that models are from the point of view of their behaviours equal. The next chapter presents the different models of interaction and concludes with a classification of dialogue forms.

4. DIALOGUE CLASSIFICATION

> There never comes a point where a theory can be said to be true. The most that anyone can claim for any theory is that it has shared the success of all its rivals and that it has passed at least one test which they have failed.
>
> A.J. Ayer, Philosophy [1982]

The design of dialogues leads sooner or later to the following question: which types of dialogues exist and how can they be described? The basic idea is to break down the whole interaction into smaller pieces (called dialogue elements or interaction units) which are the basis for a general dialogue system.

4.1 The Problem of Dialogues

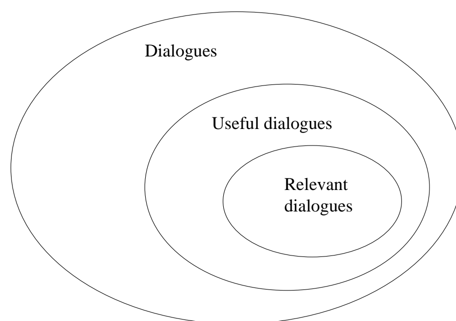


Fig. 4.1: Natural language dialogues forming subsets

A conversation can be constructed out of a set of dialogue fragments. When taking a glance at natural language conversations, the whole set consists of a large number of possible dialogues and contains (depending on the topic of discussion) useful and useless dialogue elements. As you can see in Fig. 4.1 the set of "useful dialogues" is a subset of all possible dialogues (including nonsense). Depending on the topic of conversation it also contains a rather small subset of "relevant dialogues" (dialogues necessary for the distinct topic and information exchange).

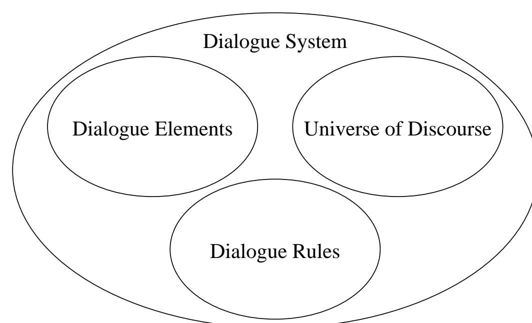


Fig. 4.2: Basis of dialogue systems

In order to understand the term “dialogue system” in the correct context, one must recognise that the whole system is based upon three basic sets. Fig. 4.2 shows that the system consists of the **universe of discourse** (the things dialogues are about) AND a **basic set of dialogues** (dialogue fragments forming the conversation) AND **dialogue rules** (the syntax the conversation must follow in order to be understandable). The set of dialogue elements represents the basic unit of dialogues (like questions or answers), and on a higher level, is used to form more complex dialogues (like sequences and series of dialogues).

Building dialogue systems raises three mayor kinds of problems:

1. The first problem is a semantic one (see universe of discourse). The system has to filter out the set of relevant dialogues (has to understand the context).
2. Nearly all user interface tools provide a basic set of dialogue elements for the interface designer (see dialogue elements). The advantages are:
   - The set of possible dialogues is finite.
   - The set can be described.
   - All constructed dialogues can be verified.
   - Sets can easily be extended.

   The risk and the problem is the size of these basic sets. If they are too small, the systems are easy to implement but the conversation might lack quality. If the sets are too large, the conversation will be of higher quality, but the systems will be too complex to be described.
3. The third problem is a syntactic one (see dialogue rules). As you saw in chapter two, there exists no commonly used method in specifying the syntax of dialogues.

The solution to the first problem will not be part of this work. As knowledge bases, re-writers and AI systems are becoming better and better, the problem will, hopefully disappear in the future.

Concerning the second problem, providing sets to the user seems to be a good idea, but two further disadvantages among common specification methods exist: 1. there is no clear separation between dialogue type and presentation (too often there is a mixture of both, style and type) and 2. there is no distinct set of basic elements.

Three approaches are possible for the classification (which enables the specification) of dialogues:

1. **Classification I:** based on the field of application.
2. **Classification II:** based on dialogue types.
3. **Classification III:** based on the structure.

4.2 Dialogue System

Chapter two presented many approaches, each covering another part in the puzzle of the whole dialogue system. The first conclusion from all approaches is that there are many parties (computer processes, the user) involved until a "real" conversation becomes possible.

Fig. 4.3 shows (in detail) the set of parties (applications, subprocesses and users) which are involved (a more general view is presented in [DRO84], p.179). For a better understanding you can imagine the whole system of being a company, where every department deals with a special part of the dialogue generation or execution. Every department of the virtual company is represented by a box in Fig. 4.3. The whole system covers the real world, the application, the dialogue management, the operating system and the semantic area. Except the application itself, the hardware and the user following processes (managers) are necessary for a working system:

- **Application Controller**.
  This process is specified by the application programmer. It controls the usage of the whole application (context), and therefore it strongly interacts with the dialogue manager.
- **Dialogue Manager**.
  Using a pool of dialogue (interaction) elements, it controls the structure (syntax) of the whole dialogue.

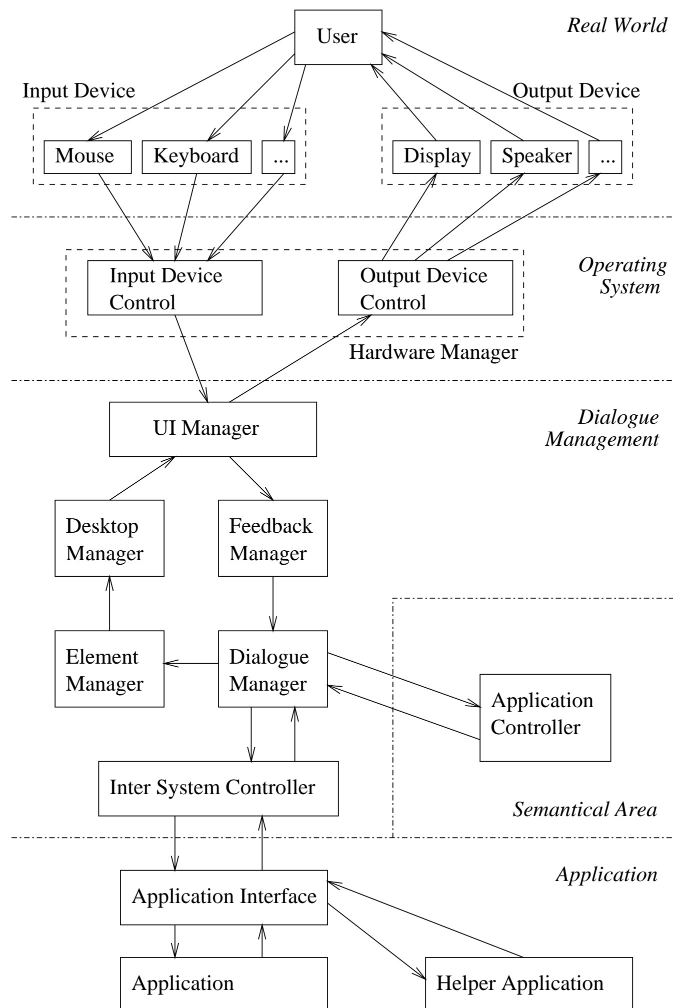


Fig. 4.3: Dialogue system architecture, detailed view

- **Element Manager**.
  Concerning different dialogue elements, this process chooses suitable interaction styles (out of a style sheet database).

- **Desktop Manager**.
  This process is responsible for the correct interaction with the window manager (User Interface Manager). Window managers (Motif or Open Look, just to mention two of them) are usually fixed on a system and differ in their behaviour of interaction elements.

- **Feedback Manager**.
  This process handles the input actions of the window manager.

- **Inter System Controller**.
  The Inter System Controller is the part of the system which is responsible for the communication with other applications (helper applications).

- **User Interface Manager**.
  This process (usually represented by the window manager of the used system) interacts with the input and output devices (Hardware Manager).

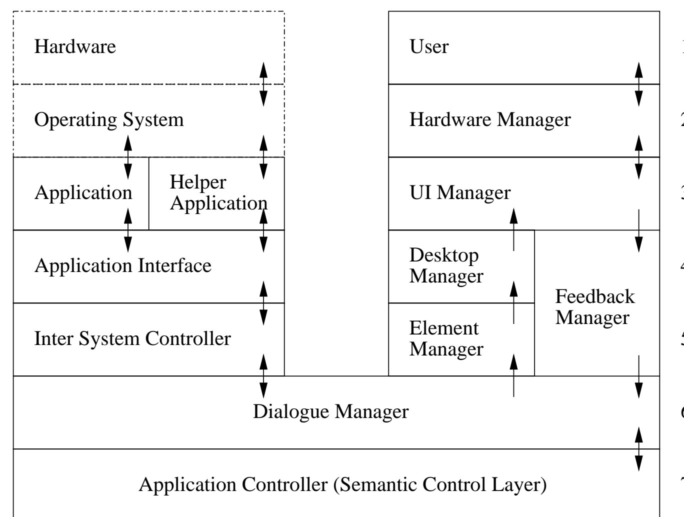


Fig. 4.4: Dialogue system, ADI model

When revisiting the flow of information and dialogue control, it is also possible to draw a layered model of the system. Fig. 4.4 shows the ADI (Application Dialogue Interface) model, a seven-layered model of the dialogue system of Fig. 4.3.

4.3 General Classification Method

The classification of dialogues is more complex than one might think, as the same dialogue element can belong to different dialogue classes. (As an - early - example a dialogue element for synchronisation of two dialogue parties can be internal or external.) Fig. 4.5 presents the first classification approach, a classification based on the field of application (a very pragmatic view!).

For reasons of completeness, all classes of dialogues are presented in Fig. 4.5 (including the human ones), but only the computer related dialogue types will be concerned.

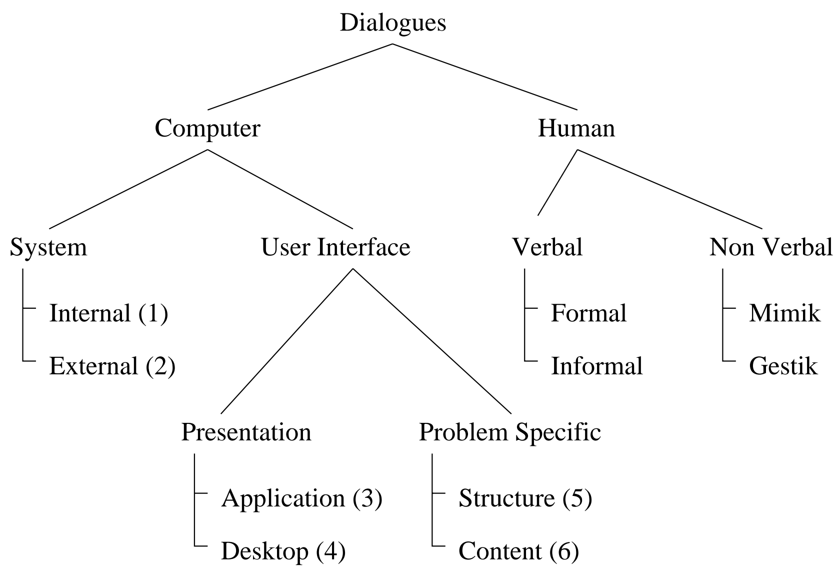


Fig. 4.5: Dialogue classification I, based on fields of application

The classification is based on the ADI model, as communication happens not only between the user and the application, but also inbetween the involved processes.

The class of computer-related dialogues can be divided into the **system subclass** and the **user interface subclass**. The system subclass contains dialogues which are **internal (1)** or **external (2)**. Internal system dialogues are mainly used between two different processes (IPC, Inter Process Communication) of the same application. External system dialogues are related to the external interface of programs or directly to the interface of UIs. Examples are remote procedure calls (RPC) or the UNIX remote shell (rsh). As the communication leaves the process' boundaries, the communication between processes and the mouse (keyboard, interface card, etc.) also belongs to this dialogue class.

The second subclass of computer related dialogues is called user interface subclass. It contains dialogues that are related to a special interface to the user (the user interface manager). This class can be divided into those dialogues which are **presentation oriented** (application or desktop based) and **problem oriented** (structure or context based).

The **application subclass (3)** represents dialogues of application based services (and explains which areas or widgets are represented on the screen). The **desktop subclass (4)** represents desktop related services. This includes the mouse and the window representation.

The problem specific subclass can be divided into the **structure-related (5)** and the **context related (6) subclasses**. Dialogues belonging to these two classes

are relevant to the control of dialogue elements and their meaning. Structure dialogues are responsible for the formal relationship of dialogue elements (syntax), whereas context dialogues, as the name might suggest, are responsible for the informal dialogue (covering the semantic area).

THE MOST IMPORTANT FINDINGS OF FIG. 4.4 AND FIG. 4.5 IS: IN ORDER TO DESCRIBE THE INTERACTIVE SYSTEM, DIALOGUES OF EVERY SUBCLASS (AND EVERY MANAGER) HAVE TO BE SPECIFIED.

The classification of dialogue types and structures is a little bit more complicated. Here CP Nets (see chapter three) help, as at the beginning of the analysis phase the correct level of abstraction needs not be fixed and can later be changed without any problems. Using CP Nets for the specification of the applications, it is easy to filter out all interaction elements. Furthermore, it is possible to separate the dialogue structure from its representation (style), which leads to the foundation of dialogue specification.

The following analysis phase is based on a model very similar to the Seeheim model and a model, Peter Lucas used in the late 1970s to describe interactive systems (as stated in [HH82], p. iii). Fig. 4.6 shows the mapping of ADI managers to the Seeheim model. For reasons of terminology, the process responsible for the user interface is called **Format Manager** (FM, including the Element Manager and Desktop Manager of the ADI model). The dialogue itself is controlled by the **Dialogue Manager** (DM, including the Dialogue Manager, the Feedback Manager and the Inter System Controller of the ADI model). The Application Interface of the Seeheim model is represented by the **Application** (APP, including the Application Interface, the Application and Helper Applications of the ADI model) itself.

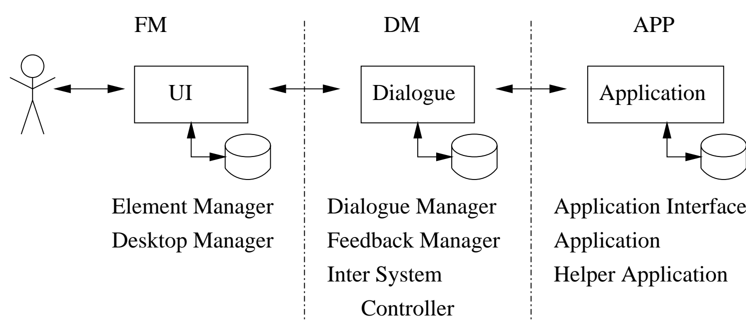


Fig. 4.6: Combination of the Seeheim and the ADI model

With this simplification it is possible to analyse the structure of different applications without loosing the characteristics needed for filtering out the dialogue elements.

4.4 Examples of Dialogues

For a general description of dialogues, it is important to know what kind of dialogues arise in more or less common software products. To be as general as possible, it is also important to choose applications from several fields.

The analysis of two applications forms the basis for the second and third approach of dialogue classification. The analysis phase will always follow these steps:

i) Summing up all dialogues arising in the application (collecting all objects, actions, processes and states).

ii) Description of all interaction styles.

iii) CPN analysis phase.

iv) Summing up structure and style.

4.4.1 Standard ML Quiz Example

The quiz example was already presented in the previous chapter. In chapter three the system was simplified so that the bound of the quiz was an abstract colour and, for that reason, the result of the quiz (the number the operator was thinking of) was not calculated. The executable system, of course, is initialised by calling the function "quiz (min,max);" and will "guess" the number following two rules:

1. If $min \geq max$, then the number is min.
2. If $min \leq max$, the middle ($m = (min + max)\ div\ 2$) is calculated and the system asks the user, if the number is $\leq$ m. If the answer is a "yes", than the number must be inbetween min and m, otherwise the number must be in the interval of $m + 1$ and max.

The quiz example is constructed from the following basic components:

Table 4.1 shows the most important components of the system. With these components, it is possible to draw the first CPN model of the quiz example. This general model (including the region definitions) can be found in the appendix (E.1 to E.2). The occurrence graph (E.4) and CPNs statistic function show us that the system has no dead marking and that the home property is satisfied.

The occurrence graph and the flow of coloured tokens (in the simulation phase) show us the structure of the system. Figure 4.7 uses a notation, very similar to the block diagrams in hardware design. Here a block represents a dialogue unit and

Objects:	user U, system S, DM, FM, range, upper bound, lower bound, guess, result, answer
Actions:	U.StartQuiz, U.Answer, U.EnterRange, S.Prelude, S.Question, S.OKMsg, S.ErrorMsg
Processes:	S.Calculate, S.GetAnswer
States:	U.Wait, U.Think, U.Read, S.WaitAnswer, S.Wait, S.StartQuiz, Answer.correct, Answer.incorrect, Result.found, Result.noIdea, S.gettingAnswer

Tab. 4.1: Quiz Example, basic components

the lines symbolise the flow of control. Each block has input and/or output nodes which can be activated. It is possible for a line to be split into two (or more) lines, symbolised by a small black circle (with arrows indicating the flow of control). As the block chart symbolises only the structure, the semantics is hidden in the small black circles. This is the place where decision is made which dialogue elements to activate (or not). (The calculation of the number between the two integer boundaries also takes place in the small circles).

The quiz example is constructed from the following dialogue elements and includes the following interaction styles:

- **I-Element**. The information element. The style is trivial, as the output is text based. The user has to read the message in order to step forward in the program.
- **BI-Question**. The binary question element with the possibility of interrupting the system. As with the I-Element, the style is trivial, it is text based.
- **Init-Element**. The initiative element represents special dialogue input elements from the user (the start of the program or an interrupt is a very special type of interaction).

As you can see in Fig. 4.7, the structure of the dialogue is not complex. The "duty" of the I-Element dialogue is simple, as the user only has to acknowledge (read) the information. The question-answer dialogue is fair and balanced, but permits an interrupt. Later it turns out how this influences the classification.

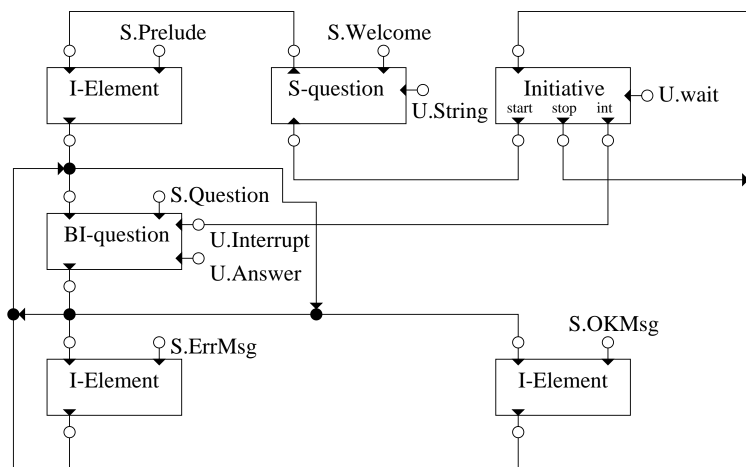


Fig. 4.7: Block structure of the quiz example

4.4.2 Mathematica Notebook

Educational software has become an important area of research. Probably the most popular educational (and commercial) software is Mathematica. It provides both a system for solving mathematical problems and a system for collecting and presenting mathematical knowledge (more or less structured into cells), better known as Notebooks (see the Mathematica 3.0 online documentation). The interface of Mathematica is of interest too, as there are many approaches (Bruno Buchberger [Buc96b] and [Buc96a], Walther Neuper [Neu97]) using Mathematica as an educational software.

Looking at typical Notebook sessions, we will again find basic dialogue components (see table 4.2). Dialogues are based on cells which can be opened or closed. They include text fields and functions. Functions can be executed, they take values, which can be defined by the user. Functions can provide graphics or animated feedback. As Mathematica is a very complex system, it is a good idea to start with the grouping of the most important interaction types (see 4.3).

Focusing on dialogue elements, Mathematica provides the user, one will find out, with many possibilities of interaction (for both the system and the user). One reason is that every party is able to control parts of the behaviour of the system. For example, it is possible for both to initiate the evaluation of a function (the user by pressing “shift and enter” and the system, when a cell is opened). For the reason of complexity only the most important interaction elements were chosen and analysed using Design CPN. (The verification of the abstract subsystem in the appendix lasts 270 minutes on a Pentium 90 and 140 minutes on a Sun Ultra Sparc 1C.)

Objects:	user U, system S, DM, FM, solver, cursor, result, notebook, cell, menu, text field, graphic field, mouse, menu options, entry field, function, expression, graphic, animation, counter
Actions:	U.StartMath, U.OpenNotebook, U.OpenCell, U.ImplodeCell, U.WriteText, U.WriteFunction, U.CreateCell, U.Quit, U.Interrupt, U.GetHelp, U.EvaluateFunction, U.SetValues, U.GetValues, U.CloseNotebook, U.SaveNotebook, U.EditEntry, U.DeleteCell, S.ShowMenu, S.CloseMenu, S.ShowCell, S.ExplodeCell, S.ShowText, S.SetUserAction, S.ShowResult, S.GetValues, S.IncrementCounter, S.EvaluateFunction, S.SetValues, S.DisplayGraphic, S.DisplayAnimation S.CreateCell, S.DisplayHelp, S.CloseHelp
Processes:	S.CalculateResult, S.InterpretAnswer, S.InterpretInput, U.EnterInformation
States:	S.Calculating, S.CellOpen, S.CellClosed, S.AnimationRunning, S.WaitforInput, U.Thinking, U.Waiting, U.Reading

Tab. 4.2: Mathematica Notebook: basic components

The flow of coloured tokens (during the simulation phase) leads to the structure of the application. As you can see in Fig. 4.8, the block structure differs from the quiz example. The structure is enriched by the choice of choosing more than one direction in the global flow of dialogue elements (work with the Notebook, with the menu, the help facility or just starting to use the solver). The specification (see E.3) demonstrates that the system can be constructed from the following dialogue elements:

- **Init-Element**. This initiative element enables the user to change the context, start, interrupt and stop the system. In Mathematica the context change can be initialized via mouse or keyboard. The interrupt is only possible by using the keyboard.
- **I-Element**. This element is used to present information to the user. The style depends on the information (text based for help text or error messages, but also graphical for some results of the solver).
- **M-Question**. The multiple question element asks the user to choose one

Cells:	open, close, explode, implode, generate, delete, help, mark
Notebooks:	open, close, start, quit, help
Solver:	enter text, enter function, help evaluate function, interrupt, edit text, mark function, present result, mouse enter
Feedback:	error message, help text, graphical result, text result animation result
Menu:	submenu, state change, options

Tab. 4.3: Mathematica Notebook: interaction elements grouped by common application fields

element out of m. In this case, the question is represented by a menu structure.

- **NI-Question**. The navigation question element (including interrupts) enables the user to take the result of a navigation element as an answer. The style is nevertheless simple: a string element (sometimes including the reference to a cell).
- **Nav-Element**. This element (navigation element) enables the user to navigate in (and work with) the information (in this case the style is very complex. Scrollbars, cells, graphic fields, entry fields and hierarchic structures are provided for the user).

At this point it is important to note that the set of elements is only **one out of many** possible sets describing the system interaction elements. One can sit a long time thinking about the best representation of highly interactive user interfaces. In fact, there exists no satisfying answer: every kind of description has its own advantages or disadvantages. As an example, another approach might use navigation elements and not question elements for the menu. It may lead to a different set of dialogue elements, but it will result in a similar structure of the system.

4.5 Dialogue Types

From the two examples in the previous section, it is possible to filter out four types of interaction elements (families of dialogue elements):

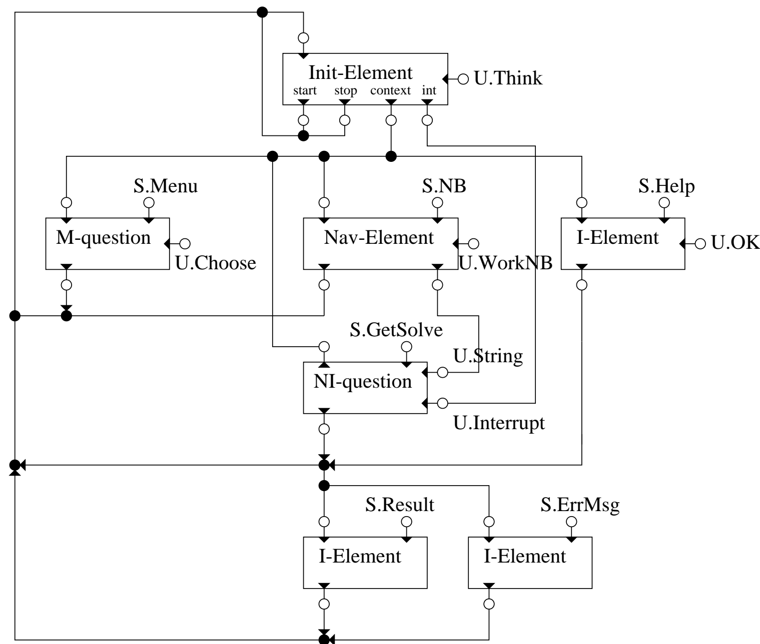


Fig. 4.8: Notebook dialogue structure block diagram

1. **Navigation Elements**. These elements are used for navigating in the *context* (for example switching between the menu and other interface elements), in the *dialogue* (for referring to previous elements) and in the *information* (especially if the information is too large).

2. **Information Element**. This element is responsible for the information exchange. The information can, but doesn't have to be acknowledged.

3. **Initiative Element**. This family includes special interacting styles: the *time-out*, the *begin* and *end* of an dialogue, an *initiative change*, a *context change*, a *dialogue type change* and the *general interrupt*.

4. **Question Element**. This element can arise in various forms: *unary* (only an acknowledge, U-question), *binary* (decision between two answers, B-question), *choice* (1 answer out of m, C-Question), *multiple choice* (n answers out of m, M-Question), *general* (a special datatype will be the answer, S-Question) and *navigation* (where the answer is the result of an navigation element). All of them can allow interrupts, too.

Fig. 4.9 summarises all dialogue elements of this approach. These dialogue elements are now used to form the basis for the second approach of classification: the classification bases upon dialogue types (see Tab. 4.4).

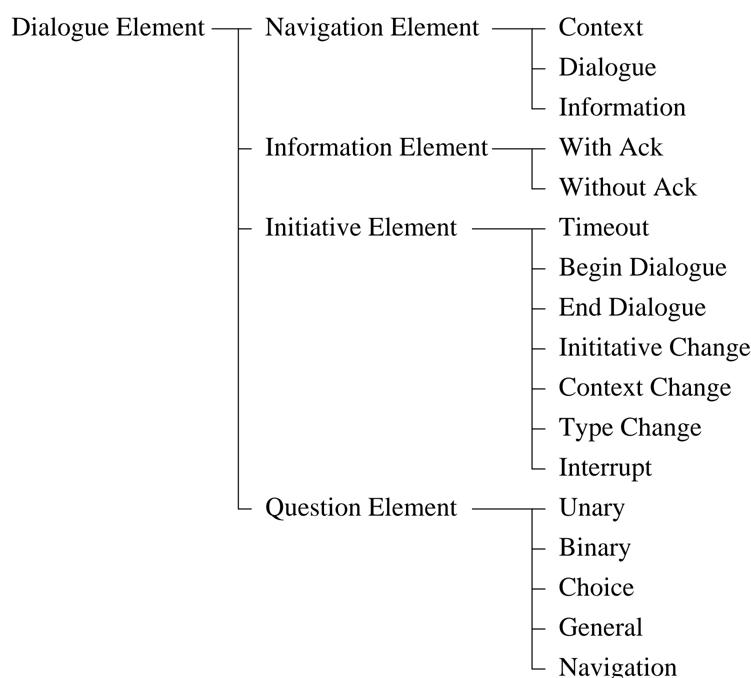


Fig. 4.9: Basic dialogue elements

4.6 Dialogue Structures

Dialogues can be written as a list of dialogue elements (with a special structure, depending on the system and the user). Using a multiparty notation, the dialogue might look like:

```
[S.Question, U.Answer]
[S.Question, U.Answer, S.Response]
[S.Question, ..., S.Interrupt, S.Answer]
```

As the context plays a crucial part in the user interface, the definition has to be refined a little bit:

A DIALOGUE IS A SET OF DIALOGUE UNITS. A DIALOGUE UNIT IS A LIST OF DIALOGUE ELEMENTS (WITH THE SAME CONTEXT). DIALOGUE UNITS ARE ALLOWED TO HAVE A DIFFERENT CONTEXT.

When looking at the quiz example, the general behaviour of the dialogue elements can be summed up as follows: the stream of elements ends by finding the solution to the riddle, the communication is balanced (question-answer game) and an interrupt is possible only for the user.

Class 1	Dialogues which only use simple question (U-Question) and simple information elements (I-Elements). Typical exponents of this class are early Web pages and the first HM-Card version. The user can only (passively) view the information and has no chance to interact or choose the direction where to go.
Class 2	Dialogues which use nearly all kinds of question elements (except the navigation question element). This class restricts the use of initiative elements and the navigation element. Timeouts, initiative changes, context changes or type changes are not possible. The quiz system is a typical example for this class, but Web pages with hyper-links belong to this class as well.
Class 3	Systems which use all kinds of interaction elements belong to this third class. An example is Mathematica.

Tab. 4.4: Dialogue classification II: based on dialogue types

On the other hand, Mathematica does not terminate itself. Interrupts are possible only for the user. The list of dialogue elements is weakly alternating (the user can ask questions from the system, but the system can also ask the user to work with menus).

There exist three basic properties for dialogue units (see Fig. 4.10):

1. **Fairness**. The structure of the list can be balanced (both user and system are represented in the stream of dialogue elements). The unit is single sided

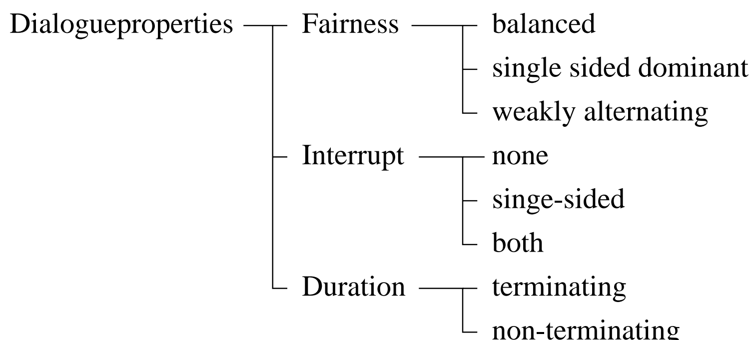


Fig. 4.10: Dialogue unit properties

dominant, if only one communication partner is mainly represented in the stream. If there is no general pattern, the unit is weakly alternating.

2. **Interrupt**. This property describes the appearance of interrupts in the unit. There can be no interrupts either. The unit is single-sided if only one communication partner can make use of interrupts.

3. **Duration**. This property describes how large the dialogue unit can be. The unit can either be terminated or not.

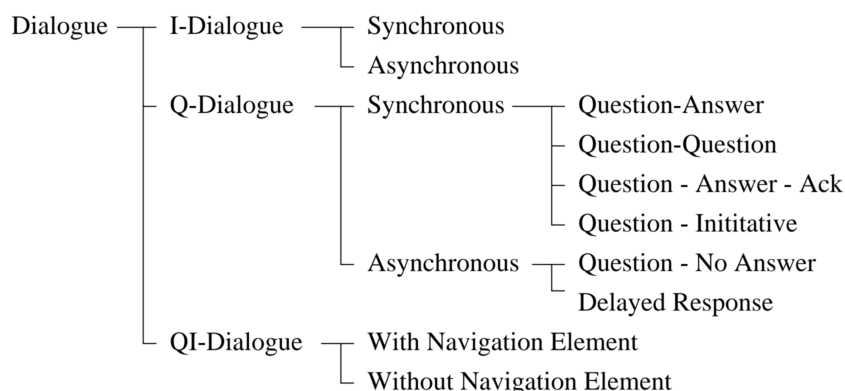


Fig. 4.11: Dialogue classification III: based on different structures

Fig. 4.11 presents the last approach of classification of dialogue systems. When revisiting dialogue properties and dialogue elements, three types of dialogue units (structures) are possible:

1. **I-Dialogues**. The dialogue unit only uses primarily I-Elements. The dialogue can be synchronous (simple Web pages with guided tours) or asynchronous (Web pages with auto-reload, slide shows).

2. **Q-Dialogues**. The dialogue unit can be synchronous (question-answer system, question-question system or a question-answer system followed or replaced by an initiative element) or asynchronous (a question followed by no answer or a question with a delayed response).

3. **QI-Dialogues**. A combination of the Q and I classes. One characteristic feature is the appearance of navigation elements (changing the context, the dialogue, the information) in the dialogue unit.

The quiz example can then be classified as follows:

```
Properties:  Fairness.balanced
             Interrupt.single sided (User)
             Duration.terminating

Structure:   Q-Dialogue.synchronous.
                       Question-Answer-Init-Element
                       (Interrupt)
```

A Mathematica Notebook can then be classified as follows:

```
Properties:  Fairness.weakly alternating
             Interrupt.single sided (User)
             Duration.non terminating

Structure:   QI-Dialogue.With Navigation Element
```

4.7 Summary

Using CP Nets makes it possible to analyse common applications. CP Nets provide a suitable level of abstraction and make is possible to trace the flow of information in the system leading the general structure of the application.

It is also possible to extract the dialogue elements and the dialogue manager from the overall structure. This was done for the simple quiz example. The dialogue elements can be collected in a basic set of interaction elements and can be used by other applications. Using this approach, only the state change of the application has to be specified, parts of the previously specified dialogue manager can be re-used, as well.

```
1.   fun initDM (ist) = case ist of
2.     is (GetRange, env) => 1‘(InitElem, dmenv env)
3.   | is (GetYesNo, env) => 1‘(BQIElem,  dmenv env)
4.   | is (SendTxt,  env) => 1‘(IElem,    dmenv env)
5.   | is (SendRes,  env) => 1‘(IElem,    dmenv env)
6.   | _                  => empty;
```

The dialogue manager is able to select the correct type of interaction (dialogue element). If the application needs a “yes” or a “no”, the dialogue manager can use the BQI-Element (line 3 in the function `initDM`) for the communication.

The major advantage of this approach is that the style of the dialogue elements and the type of the dialogue element is not fixed. This means that the appearance

of the BI-Element can be graphical or text based, the dialogue manager does not have to deal with the style.

It is also possible for the dialogue manager to change the type of interaction element. All that has to be done is to replace line 3 (in the function `initDM`):

```
3.   | is (GetYesNo, env) => 1`(BQIElem,  dmenv env)
```

with the following line

```
3.   | is (GetYesNo, env) => 1`(SQElem,  dmenv env)
```

This can be done automatically by the dialogue manager (if determining that the user has no problems with text based inputs) - influencing the behaviour of the whole application.

5. FUTURE WORK

sero medicina paratur,
cum mala per longas invaluere moras.

Ovid [43 a.D. - 18]

5.1 Ideas for Implementing a WWW-based Educational System

In the area of educational programs and classroom builders, many tools and environments (WebCT, Webfuse or Topclass, see [McCormack 98]) have become popular. But all the tools and methods have some disadvantages:

1. Building the user interface is hard work. In fact, too many lines have still to be written for the interface and dialogue control.
2. They lack the definition of heavy graphic human-computer interfaces or do not use all features of WWW-based systems.
3. They only provide a small set of dialogues and interaction styles. In this case, the author is limited in didactics and methodology.
4. There are no different interaction styles for the different users.

The solution is to describe dialogues at a very abstract level and let the system generate the dialogue automatically (as it is the same with markup languages like Latex or HTML). Style files guarantee that the content is separated from its layout on the screen.

Extensions already exist for authors (popular classroom builders - like WebCT - are getting more and more extensions - SATML, see [Ferrandez 98]). Many of them use meta languages for the description of dialogues and contents, but they are limited in interaction elements, as well.

Meta languages are very popular, they are highly readable and therefore it is a good idea to use a meta language for the description of WWW-dialogues. LaTeX could be used as a basis in order to avoid learning a completely new language or dialect. Every other meta language would be possible, but LaTeX is widely used and furthermore it has the following advantages:

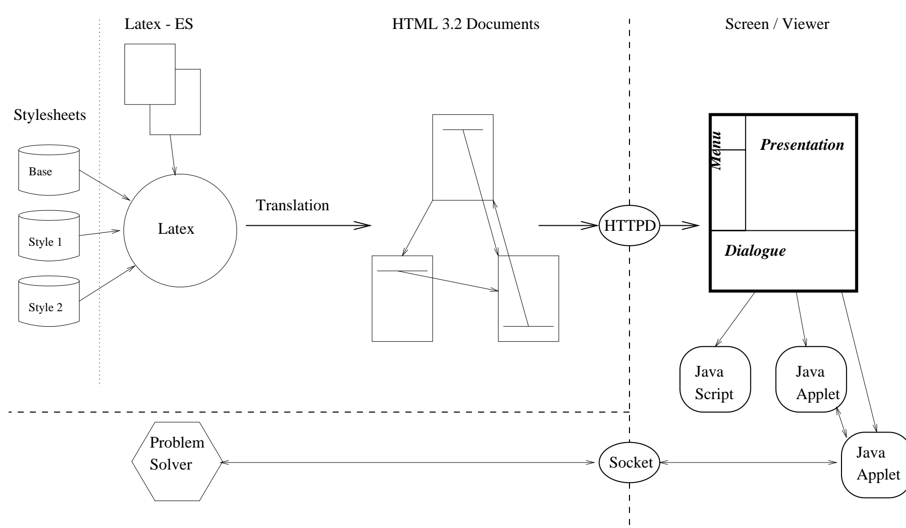


Fig. 5.1: General structure of WWW-based educational systems

1. LaTeX is easy to extend. Extensions for Lyx (an editor) are also possible.
2. LaTeX is easy to understand.
3. The document can be verified easily.
4. It is possible to print the content in a pleasant way.
5. Dialogues and content can be described as the author has them in mind. The decision about methodology or didactics is up the author and not the tool designer.
6. For different problems different style sheets can be used.
7. The style sheet is responsible for an optimal representation of the lecture.

To build a WWW-based educational system (which is capable to handle all demands listed before) the following steps are necessary:

1. Analysing common WWW-based applications in the same way as in chapter four. This leads to a set of WWW-based dialogue elements.
2. Forming a rudimentary set of dialogues and checking their completeness.
3. Definition of a meta language (LaTeX-ES, LaTeX for Educational Systems) to represent the dialogue elements.

4. Definition of style files.

5. Definition of the dialogue manager and the application interface manager, which will be the link between the user, system and other applications.

Fig. 5.1 shows the principal structure of a system which automatically generates WWW applications. The description of the content, dialogues and interfaces, together with style sheets are (depending on the interaction styles) translated to HTML (Java Scripts, XML, ...) files (and Java classes). Because of the separation of content and representation, every style is possible for the same kind of dialog (graphical input, textual input or even speech recognition - depending on the user who is working with the system).

5.2 Helper Applications

As stated in chapter four, the definition of dialogues to helper applications (knowledge base, problem solver etc.) is also necessary. The goal is the definition of dialogue elements related to the inter system controller (to describe the communication between the helper applications - applets, scripts, databases, Mathematica Notebooks) and the dialogue manager.

Using helper applications within the system means the introduction and definition of different states at both sides.
The new set of dialogues between the dialogue manager and the application is finite, and what is most important, only a small part of all possible dialogues with the helper application is really necessary to be defined.

The dialogue manager is then enriched by a well defined set of dialogues to the inter system controller and plays the part of a mediator between the two worlds (see 5.2). The set of dialogues which can be realized within the system is called State Dialogues (S-Dialogues, as they change the state of the application and the dialogue manager). Once again, for every application it is necessary only to describe the relevant dialogues. This approach has the following advantages:

- The author can decide which features of the application to use.
- The basic set is easy to extend and macros can help to deal with more complex structures.
- The interface only has to be described as detailed as necessary. Irrelevant parts can be omitted.

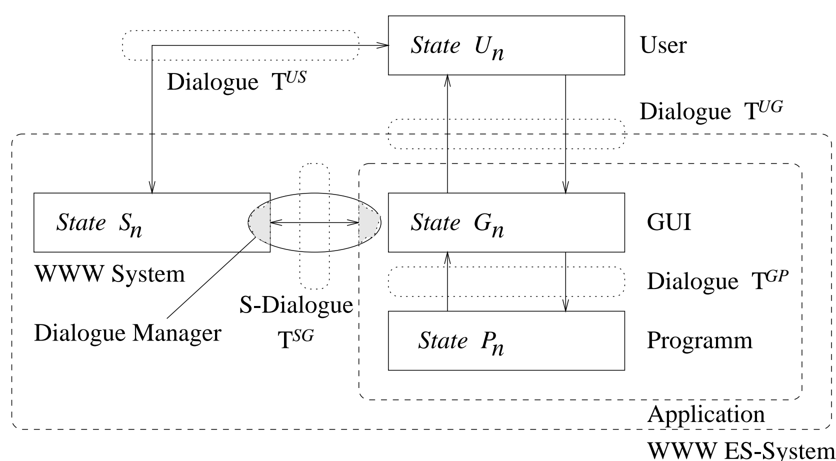


Fig. 5.2: General structure of a WWW-based educational system using S-Dialogues

When looking at Fig. 5.2, it should be clear that every part of the system is represented by different states, which can be changed by transitions (T) obeying the following properties:

1. Transitions T are right-unique transition relations.
2. Not defined transitions T^* do not change any states S.
   $T^*(S_n) = S_n$.

This indicates that only defined actions lead to a state transition and the author can forget about irrelevant features of the helper application.

5.3 Summary

The analysis phase of simple dialogues in chapter four showed that it is possible to build a system from basic dialogue elements. These elements can be described by a meta language (LaTeX-ES), and the TeX-system can be used for the automatic generation of WWW based dialogue systems.

In order to build a flexible WWW-based educational system using Mathematica, the following work has to be done:

- CP Net analysis of the Mathematica interface (on a powerful computer).
- Analysis of common WWW-based educational systems.
- A style database has to be provided (to show the benefits of the abstraction).

- Some other helper applications and guidelines for application writers also have to be provided.
- Extension of the system to be able to work with or within classroom builders and adapting it to work within other Hypermedia systems.
- Definition of a bigger set of dialogues to cope with other applications and to be able to build Edutainment systems, too.

LaTeX-ES is only an idea of what could be done. Further research is necessary to find the limits of such a system. It is still not clear, if all kinds of dialogue (interaction) elements can be described using a markup language. Also, up to now, the definition of S-Dialogues is not specified in detail.

Nevertheless, the use of markup languages seems to be a good idea (as a test prototype showed) and encourages further investigation.

6. CONCLUSION

An idea that is not dangerous
is unworthy of being called
an idea at all.

Oscar Wilde [1854 - 1900]

When building educational systems, the application programmer is concerned with both the structuring of the knowledge correctly (by means of methodology) and designing the dialogues for the human computer interaction. There exist many books about methodology and didactics, but up to now there has been no sound description combining contents, style and types of dialogues. When looking at the problem of writing down (defining) dialogues, the question of what kind of dialogues one is talking about arises.

Chapter two goes back to the roots and starts with the Seeheim model, a model useful for future analysis. As many phrases in literature are not clear, it gives an overview of commonly used words like "UIMS" or "toolkits".

The chapter also introduces 24 approaches from literature, and classifies them using a 10-dimensional feature space. All methods are assessed and the results are summed up in the appendix. Six projections to the plane are chosen to help the user in his decision, which approach to use for a distinct problem.

The result of the comparison of all approaches leads to the assumption that Coloured Petri Nets are a sound form for describing and analysing interactive applications.

Chapter three introduces the world of Petri Nets and explains their advantages in the design and analysis phase: their great mathematical background, a sound graphical representation, a well-defined semantics, concurrency, time concepts and a good support for drawing and simulation. Coloured Petri Nets provide a data structure (Petri Nets of class three) and an explicit description of both action and states.

The chapter concludes with the presentation of a simple example and the tool Design/CPN, a public domain tool for the drawing, analysis and simulation of CP Nets.

Chapter four focuses on the problems arising in the classification of dialogues and dialogue properties. The underlying idea in the analysis phase breaks down the

whole interaction into small units (called dialogue elements). Three approaches are presented: the classification based on the subject field, the classification based on the dialogue type and, the third method, the classification based on the structure of the dialogue.

The basis for the first classification method is the structure of general interactive systems. It turns out that there are many levels of dialogues in the complex system, and each of them has to be described.

The next two classification methods are based upon CPN analysis of a simple SML program and a simple model of a Mathematica notebook. The second class (based on types) divides dialogues into three subclasses, depending on the used dialogue elements. The different types describe how powerful the dialogue system of the applications really is. The third class is based upon different structures of dialogues arising in applications.

The chapter closes with the definition of dialogues in general and shows that it is possible to separate the interacting type and style from the application itself.

Chapter five presents an idea of how to describe and (automatically) implement an educational system (based on Mathematica). It argues about the use of a meta language to describe the dialogue elements and presents the idea of extending LaTeX classes and style files. It also focuses on the problem of helper applications and closes with a summary of work that has to be done in order to build a running system.

There are still things of interest left: First of all it is important to analyse more applications in order to extend the classification (as this classification is the basis for a specification). This work has shown that it is possible to separate dialogue elements from an interactive system. Dialogues take also place inbetween two applications and should be analysed, too. Having a large collection of elements and different styles available, it is possible to build a WWW-based educational system prototype, but research is necessary to find out the benefits and restrictions of such a system.

How much time did programmers need to create their interfaces? How often did they change them? Would it not be great to write them down like editors, only defining the content and not concerned with details? Style files can do the work for you, generating interfaces as wanted. Of course, it sounds unrealistic, but those problems are solved for editors - and LaTeX-ES might be a step in the right direction.

APPENDIX

A. ASSESSMENT OF METHODS

Method Name:		**BNF**
Level	Mark	Remarks
Separation	4-7	excellent for syntax
Mathematical	9	lack description of semantics
Description	1	only with extensions semantics
Usability	5	BNF is easy to understand
Readability	3	large for huge systems, many tokens
Development	2	large for huge systems, many tokens
Portability	6	standard notation
Graphical	1	only text based
Abstraction	5	compiler needed
Extension	8	in syntax nearly no limit

Tab. A.1: Assessment: BNF

Method Name:		**TAG**
Level	Mark	Remarks
Separation	7	good: UI actions to be defined
Mathematical	9	BNF as background, rule schemas
Description	4	schemas describing tasks
Usability	4	rule definition is complex
Readability	4	as BNF, but also semantic fields
Development	1	as BNF, extra rule definition
Portability	6	as BNF, rule schemas for consistency
Graphical	1	no graphical editor available
Abstraction	5	as BNF
Extension	6	as BNF, be careful with rules

Tab. A.2: Assessment: TAG

Method Name:		**Lex-yacc**
Level	Mark	Remarks
Separation	5	Generates dialogue controller
Mathematical	7	context free grammar, actions have no background (-2)
Description	2	controls syntax of dialogue
Usability	1	CFG and X-Windows knowledge
Readability	2	expert in CFG and X-Windows commands
Development	3	good for text, bad for graphics
Portability	3	CFG has advantages, need of X windows (-2)
Graphical	1	text based system
Abstraction	3	Control level is high (5), but also X code (-2)
Extension	5	Fixed X set (-2), flexible in design of new widgets (7)

Tab. A.3: Assessment: Lex-yacc

Method Name:		**Multiparty BNF**
Level	Mark	Remarks
Separation	5	shows dialogue flow
Mathematical	8	BNF as background
Description	3	syntax like BNF (1), labels show meaning (+2)
Usability	6	like BNF, labels make it easier (+1)
Readability	5	advantage of labels
Development	4	more abstract, less code
Portability	6	as BNF
Graphical	1	text editor
Abstraction	6	as BNF, more general with labels (+1)
Extension	4	only good for textual interfaces lack in two dimensional environments

Tab. A.4: Assessment: multiparty BNF

Method Name:		**UAN**
Level	Mark	Remarks
Separation	7	describes both feedback and control
Mathematical	4	rule based
Description	6	describes both, stresses semantics
Usability	3	notation hard to learn
Readability	4	notation lacks, mnemonics helps (+1)
Development	4	only notation
Portability	6	only notation
Graphical	1	text based, mnemonic symbols
Abstraction	7	as BNF, but more life (+2)
Extension	3	lack in heavy GUIs

Tab. A.5: Assessment: UAN

Method Name:		**Transition Diagrams**
Level	Mark	Remarks
Separation	6	focus on input control
Mathematical	5	TDs have background, actions not
Description	3	syntax of control, actions for semantics
Usability	7	simple notation, graphical appealing
Readability	7	graphical appealing and simple
Development	4	only notation, no code generated
Portability	6	only notation
Graphical	3	text and graphic
Abstraction	7	shows structure
Extension	6	interaction styles limited

Tab. A.6: Assessment: transition diagrams

Method Name:		**USE Diagrams**
Level	Mark	Remarks
Separation	5	control level
Mathematical	5	TDs have background, labels not
Description	3	semantics through operations
Usability	4	TDs easy, operation notation has to be learned
Readability	6	partly graphical
Development	5	only notation, but parts are code
Portability	7	notation and operation code
Graphical	3	text and graphic
Abstraction	5	shows structure and code
Extension	6	interaction styles are limited

Tab. A.7: Assessment: USE diagrams

Method Name:		**State Charts**
Level	Mark	Remarks
Separation	5	control level
Mathematical	5	TDs have background, labels not
Description	3	little semantics with grouping
Usability	5	TDs make it easy, also the grouping
Readability	7	graphical notation, also hierarchies
Development	4	only notation
Portability	7	only notation
Graphical	4	graphic and text
Abstraction	6	shows structure and hierarchy
Extension	6	interaction styles are limited

Tab. A.8: Assessment: state charts

Method Name:		**Place Transition Petri Nets**
Level	Mark	Remarks
Separation	5	shows only syntax
Mathematical	9	excellent mathematical background
Description	2	shows only syntax
Usability	4	token game is often tricky
Readability	5	graphical method
Development	5	hard to find level of abstraction
Portability	8	only notation
Graphical	4	graphical editors exist
Abstraction	5	depends on system designer
Extension	3	limited game of tokens

Tab. A.9: Assessment: place transition Petri Nets

Method Name:		**Coloured Petri Nets**
Level	Mark	Remarks
Separation	3-7	Colour benefits
Mathematical	9	excellent mathematical background
Description	5	with tokens very flexible
Usability	3	coloured token game very tricky
Readability	4	graphical method, colour types tricky
Development	6	hard to find level of abstraction
Portability	8	only notation, colour sets in ML
Graphical	4	graphical editors exist
Abstraction	4-9	depends on the system designer
Extension	9	tokens are very mighty

Tab. A.10: Assessment: Coloured Petri Nets

Method Name:		**Event Languages**
Level	Mark	Remarks
Separation	5	direct control of dialogue
Mathematical	5	TD and NSD as background
Description	5	shows syntax and semantics
Usability	6	TD easy, but NSD needs knowledge
Readability	4	graphical notation, NSD benefits (6) but the code is large (-2)
Development	4	difficult to write correct code
Portability	7	only notation
Graphical	3	graphics and text
Abstraction	4	sub-diagrams are near to system
Extension	6	interaction styles are limited structure and hierarchy definable

Tab. A.11: Assessment: event languages

Method Name:		**Declarative Languages**
Level	Mark	Remarks
Separation	4-6	deals with system information
Mathematical	3	programming language
Description	7	information oriented
Usability	5	limited set of interactions, variable based
Readability	5	simple, as limited styles
Development	5	only sets variable values
Portability	2	limited to toolkit
Graphical	3	text based, toolkit exists for UI
Abstraction	1	near to language
Extension	1	fixed set

Tab. A.12: Assessment: declarative languages

Method Name:		**Constraint Languages**
Level	Mark	Remarks
Separation	7	describes relationship between objects
Mathematical	3	graphical specification and description but programming language
Description	7	relationship between objects
Usability	1	difficult to keep track of consequences
Readability	1	difficult to debug, understand system
Development	3	hard to find errors
Portability	1	language support dependent
Graphical	5	graphical and textual specification
Abstraction	2	constraint solver does some work
Extension	6	research system, limited set

Tab. A.13: Assessment: constraint languages

Method Name:		**Application Frameworks**
Level	Mark	Remarks
Separation	4-7	programmers choice
Mathematical	3	programming language
Description	4	describes both,
Usability	3	C code, OO programming knowledge necessary
Readability	5	programming language
Development	7	pre-defined classes
Portability	3	language support dependent
Graphical	1	textual bases, UI-builder might help
Abstraction	2	classes benefits, but near to system
Extension	9	no limits

Tab. A.14: Assessment: application frameworks

Method Name:		**Automatic Generation**
Level	Mark	Remarks
Separation	4-7	user and system dependent
Mathematical	5	rule based, programming language
Description	7	dealing mainly with semantics
Usability	5	language has to be learned
Readability	6	only defining lists
Development	7	system is deciding
Portability	2	language based
Graphical	1	text based
Abstraction	9	system applies all rules and styles
Extension	4	style, rules are fixed
		not easy to extend

Tab. A.15: Assessment: automatic generation

Method Name:		**Denotational Approach**
Level	Mark	Remarks
Separation	4-7	user definable
Mathematical	9	fully formal
Description	4-9	user definable, syntax and semantics
Usability	1	complex notation
Readability	2	complex
Development	1	everything has to be defined
Portability	7	notation, translators exist
Graphical	1	text based
Abstraction	4-9	user definable
Extension	9	no limits

Tab. A.16: Assessment: denotational approach

Method Name:		**Graphical Editor**
Level	Mark	Remarks
Separation	9	purely front end
Mathematical	3	rule based,
		programming language as background
Description	7	defining parameters of objects
Usability	8	UI specification
Readability	8	graphical and parameters
Development	7	drag and drop and specifying parameters
Portability	6	language dependent, translator to C
Graphical	7	modification at run-time
Abstraction	8	only parameter specification
Extension	2	limited set

Tab. A.17: Assessment: graphical editor

Method Name:		**Menu Trees**
Level	Mark	Remarks
Separation	7	front end only, some dialogue structure
Mathematical	2	hierarchical order
Description	4	describes both, structure and meaning
Usability	9	tree structure
Readability	9	simple as tree structure
Development	3	notation, some translator exist
Portability	8	only notation
Graphical	7	graphical editor
Abstraction	8	shows structure
Extension	3	not for all interaction styles

Tab. A.18: Assessment: menu trees

Method Name:		**Prototypes**
Level	Mark	Remarks
Separation	9	only front end
Mathematical	1	no background
Description	4	only visual effects
Usability	9	novice user because of UI
Readability	9	only graphical objects
Development	7	high, but only parts of code usable
Portability	3	only visual meaning, no translator
Graphical	7	direct manipulation
Abstraction	3	looks like system, but no connection
Extension	8	no limits in behaviour

Tab. A.19: Assessment: prototypes

Method Name:		**Cards**
Level	Mark	Remarks
Separation	7-9	user definable
Mathematical	1	no background
Description	4	both, but stresses syntax
Usability	9	graphical UI
Readability	9	graphical objects
Development	8	drag and drop
Portability	1	system dependent
Graphical	9	direct manipulation
Abstraction	2	direct manipulation of system
Extension	8	no limited set

Tab. A.20: Assessment: cards

Method Name:		**Interface Builders**
Level	Mark	Remarks
Separation	4-8	User definable
Mathematical	1	no background
Description	3	both, but stresses syntax
Usability	7	language knowledge
Readability	8	graphical UI
Development	7	graphical and textual specification
Portability	3	system dependent, limited set of translators
Graphical	8	graphic and text
Abstraction	2	direct system programming, some classes
Extension	7	limited set, heavy programming

Tab. A.21: Assessment: interface builders

Method Name:		**Editing Tools**
Level	Mark	Remarks
Separation	6-9	user definable
Mathematical	1	no background
Description	3	both, but stresses syntax
Usability	5	programming language needed
Readability	4	complex code
Development	7	graphical UI and programming tools
Portability	3	language dependent, libraries exist
Graphical	5	graphical UI for programming
Abstraction	3	near to implemented system
Extension	5	possible, but heavy programming

Tab. A.22: Assessment: editing tools

Method Name:		**Ordinary Toolkits**
Level	Mark	Remarks
Separation	8	focuses on representation
Mathematical	1	no background
Description	2	focuses on presentation
Usability	4	programming knowledge needed
Readability	3	complex, large code
Development	6	large libraries
Portability	3	limited to system
Graphical	3	editor exists, textual
Abstraction	7	hides code in libraries
Extension	5	possible, but difficult

Tab. A.23: Assessment: ordinary toolkits

Method Name:		**Virtual Toolkits**
Level	Mark	Remarks
Separation	8	focuses on presentation
Mathematical	1	no background
Description	4	presentation and meaning
Usability	3	programming knowledge
Readability	3	complex
Development	6	large libraries
Portability	6	open to other systems
Graphical	3	editor exists
Abstraction	8	hides code in libraries
Extension	3	possible, much work for more than one system

Tab. A.24: Assessment: virtual toolkits

B. GRAPHICAL COMPARISON

B.1 Abbreviations

Table B.1 sums up the numbers that are used to denote the different methods and approaches in the graphical comparison (to save space on the area).

1..BNF	2..TAG	3..Context Free Grammars
4..Multiparty BNF	5..UAN	6..Transition Diagrams
7..USE Diagrams	8..State Charts	9..Ordinary PN
10..Coloured PN	11..Event Lang.	12..Declarative Lang.
13..Constr. Lang.	14..Appl. Fwks	15..Autom. Gen.
16..Denotational Appr.	17..Graph. Editors	18..Menu Trees
19..Prototypes	20..Cards	21..Interface Builders
22..Editing Tools	23..Toolkits	24..Virtual Toolkits

Tab. B.1: Abbreviations used in assessment diagrams

B.2 Graphical Comparison based on Assessment

In the following, six projections have been chosen to help the user in the decision of choosing between the different approaches and methods. (Other projections would have been possible, but are up to the reader.)

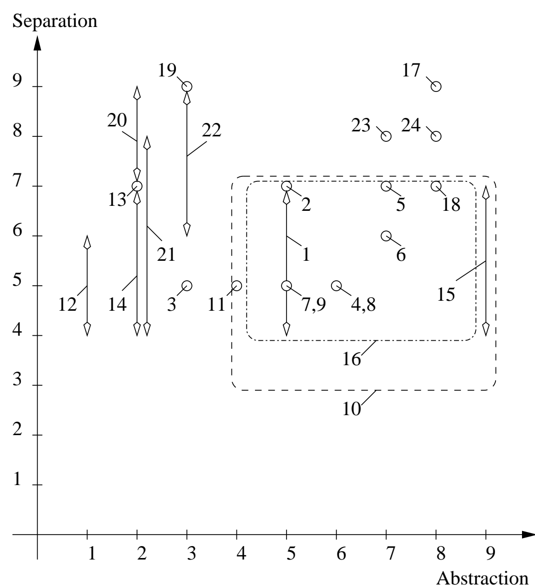


Fig. B.1: Comparison via separation and abstraction level

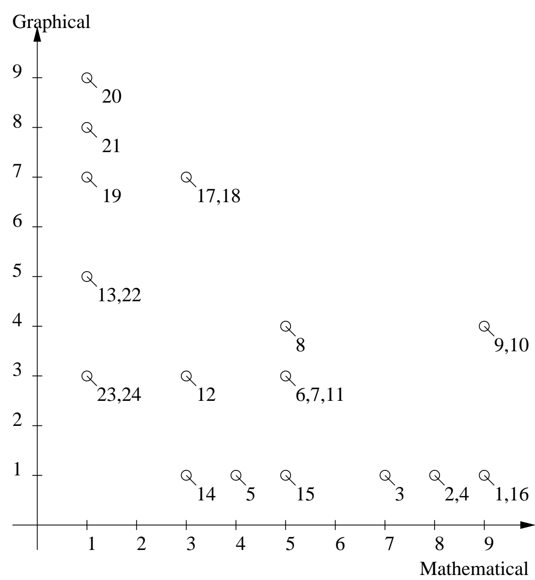


Fig. B.2: Comparison via graphical and mathematical level

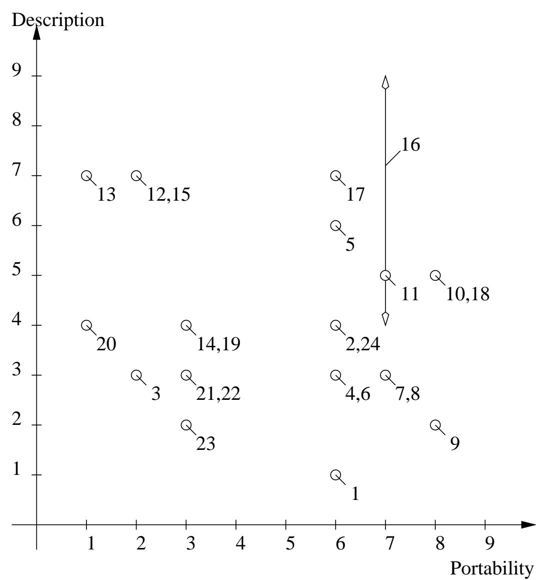


Fig. B.3: Comparison via description and portability level

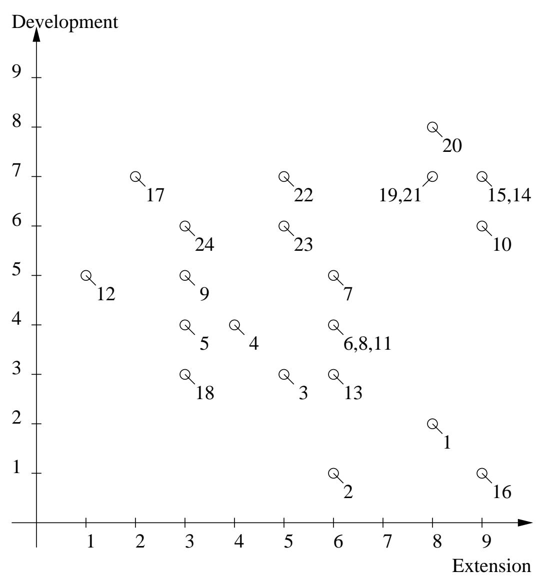


Fig. B.4: Comparison via development and extension level

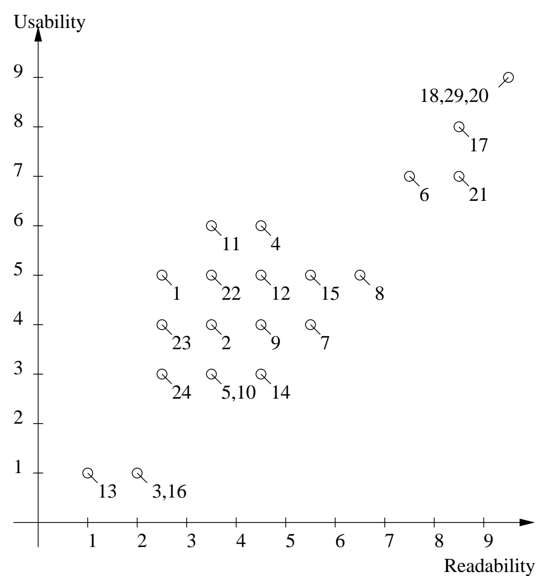


Fig. B.5: Comparison via usability and readability level

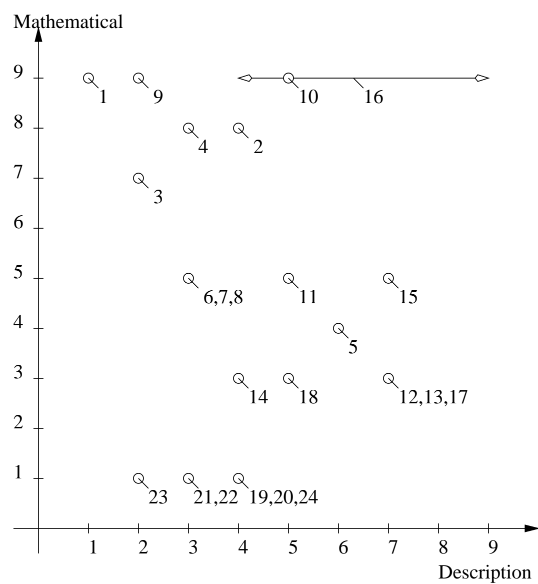


Fig. B.6: Comparison via mathematical and description level

C. PETRI NET CLASSIFICATIONS

1. **C/E Systems**. Condition Event systems belong to Petri Nets of level one. The structure is very simple and the net must fulfil the 1-liveness and forward and backward reachability. That means that transitions can occur even concurrently, either backward or forward.
   Some tools also exist, like HyperNet, a modelling and analysing tool, or WinPetri, a Petri Net tool for Windows.

2. **EN Systems**. Elementary Net systems are a simpler class than C/E systems, fulfilling only the forward reachability property.

3. **1-safe systems** are nets whose places can be marked by at most one unstructured token. They are often characterised as a subclass of ordinary Petri Nets as they have infinite place capacities, unary arc weights and boolean markings.

4. **State Machines** are a simple subclass of 1-safe systems. Every transition can have only one input place and one output place. With state machines it is not possible to model concurrency.

   Definition C.1: State Machine. A State Machine is a Petri Net with the properties $\forall t \in T : |\cdot t| = |t \cdot| = 1$

   An example of a simple state machine can be seen in Fig. C.1.

5. **PT Nets**. Place Transition Nets belong to level 2, because their places may be marked by more than one token (counter tokens). They have arc weights and place capacities.
   There are many tools for this important class of Petri Nets, CPN/AMI or WinPetri, just to mention two of them.

6. **PN**. Ordinary Petri Nets also belong to the second level and are a subclass of PT Nets. They have infinite place capacities and unary arc weights:

   Definition C.2: Ordinary Petri Net. A PN net is a P/T system $\Sigma = (S, T, F, K, W, M_0)$ with the properties $\forall s \in S : K(s) = \infty$ and $\forall f \in F : W(f) = 1$. The net $\Sigma = (S, T, F, M_0)$ is then called an (ordinary) Petri Net.

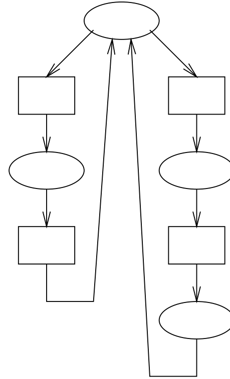


Fig. C.1: State machine: simple example

7. **FC Nets**. Free Choice are a subclass of ordinary Petri Nets. They avoid confusion, as they have the property, that if two transitions have a common input place, they have no other input places. Every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition.

   Definition C.3: Free Choice System. A FC Net is a PN net $\Sigma = (S, T, F, M_0)$ with the properties $\forall t, t' \in T, t \neq t' : \cdot t \cap \cdot t' \neq \emptyset \Rightarrow |\cdot t| = 1 = |\cdot t'|$

   There are many extensions to FC Nets which have weaker structural constraints. One of them is the extended FC Net or the behaviourally FC Net.

   Definition C.4: Extended Free Choice System. An EFC Net is a PN net $\Sigma = (S, T, F, M_0)$ with the properties $\forall t, t' \in T : \cdot t \cap \cdot t' \neq \emptyset \Rightarrow |\cdot t| = |\cdot t'|$

   Definition C.5: Behaviourally Free Choice System. An BFC Net is a PN net $\Sigma = (S, T, F, M_0)$ with the properties $\forall t, t' \in T : \cdot t \cap \cdot t' \neq \emptyset \Rightarrow \forall M \in [M_0\rangle M \textit{ enables } t \Leftrightarrow M \textit{ enables } t'$

   Fig. C.2 shows some simple examples of FC Nets.

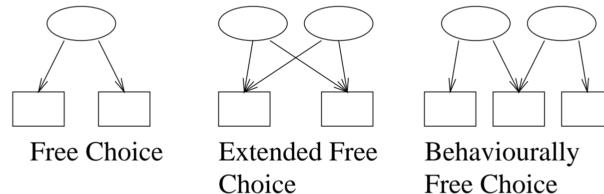


Fig. C.2: FC Nets: simple net structures

8. **S-Systems**. S-systems are a subclass of FC nets, where every transition has at most one pre- and post place. That means that $\forall t, t' \in T : |\cdot t| \leq 1 \wedge |\cdot t'| \leq 1$. Synchronisation is not possible.

9. **T-System**. A T System is a subclass of a FC Net, where every place has at most one pre- and post transition. $\forall s \in S : |\cdot s| \leq 1 \wedge |\cdot s'| \leq 1$. Since there are no (forward) branched places, there can never be any conflicts.

10. **Marked Graph**. The characteristic is that the marked graph has an underlying net where every place has only one input transition and one output transition. $\forall s \in S : |\cdot s| = 1 = |\cdot s'|$.

11. **ER Nets**. Environment/Relationship nets are Petri Nets of level three. Their tokens represent environments which are functions associating values to variables.

12. **ALG Nets**. An Algebraic Net consists of a pair, an algebraic specification and a net with sorted places. Their arcs are marked with terms over the specification and their transitions have predicates (a set of equations). The algebraic net scheme consists of a basic coloured net where colours and firing rules are represented by the interpretation of terms over a given algebraic specification. An example of an algebraic net can be seen in Fig. C.3.

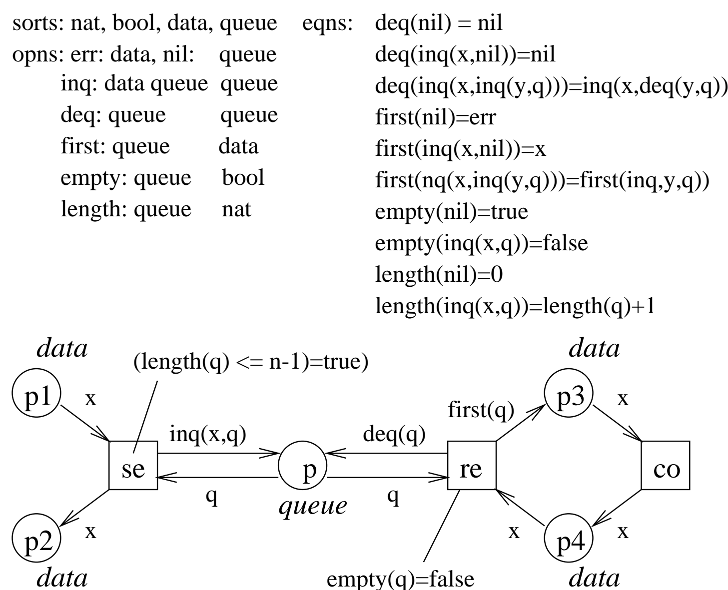


Fig. C.3: ALG Net: sender/receiver model

13. **OBJSA Nets**. Object SA (Superposed Automata) Net Systems are high level Petri Nets where the net can be decomposed in State Machine Components. The domains of the individual tokens are defined as abstract data types using the OBJ2 language. A SA Net is a automata with the property that components cannot share places.

14. **HL+ADT Nets**. High level Petri Nets with abstract data types belong to the third level and have tokens and firing rules that are specified over an algebraic specification.

15. **Product Nets**. Product Nets are Petri Nets which are extended by inhibitor and erase arcs, transition inscriptions and individual tokens. Product Nets consists of

    - a preamble where the used sets and functions are defined,
    - an unlabelled net including inhibitor and erase arcs and
    - a labelling with arc labels (n-tuples), optional transition inscriptions and domains for places.

16. **CP Nets**. Coloured Petri Nets are described in chapter three in more detail.

17. **Regular Nets**. A Regular Net is a Coloured Petri Net with the property that the domains of places and transitions are made of any Cartesian product of basic object classes. Every class appears not more than once in the product.

18. **Well formed Nets**. They are formally defined as an extension to Regular Nets and have the same modelling power as CP Nets. The difference is that the expression of the colour functions and classes is written in a more explicit (and parametric) form.

19. **Trad. HL Nets**. Traditional High Level Nets are a superclass of Coloured Petri Nets and Pr/T Nets (see below).

20. **Pr/T Nets**. Predicate/Transition Nets have been the first class of high level Petri Nets. It uses the notation and concepts of many-sorted algebras.

D. DESIGN/CPN EXAMPLES

D.1 Design CPN - Screen Shot

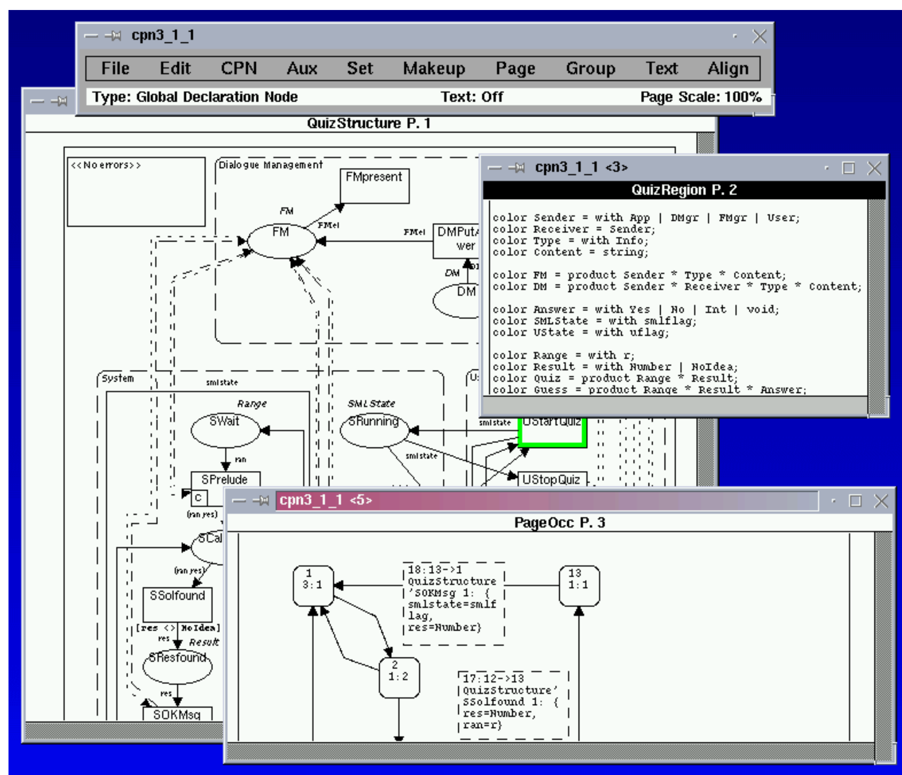


Fig. D.1: Design/CPN: screen shot

D.2 Producer/Consumer System - Statistics

```
 Statistics
------------------------------------------------------------------------
  Occurrence Graph
    Nodes:   13
    Arcs:    20
    Secs:    0
```

```
    Status: Full

  Scc Graph
    Nodes:  1
    Arcs:   0
    Secs:   0


 Boundedness Properties
------------------------------------------------------------------------
  Best Integers Bounds    Upper      Lower
  RAS2'Res 1             6          0
  RAS2'State 1           5          5

  Best Upper Multi-set Bounds
  RAS2'Res 1          1`r+ 3`s+ 2`t
  RAS2'State 1        2`(p,b)+ 1`(p,c)+ 1`(p,d)+ 1`(p,e)+ 3`(q,a)+
                      1`(q,b)+ 1`(q,c)+ 1`(q,d)+ 1`(q,e)

  Best Lower Multi-set Bounds
  RAS2'Res 1          empty
  RAS2'State 1        1`(p,b)+ 1`(q,a)


 Home Properties
------------------------------------------------------------------------
  Home Markings:  All


 Liveness Properties
------------------------------------------------------------------------
  Dead Markings:  None
  Dead Transitions Instances: None
  Live Transitions Instances: All

 Fairness Properties
------------------------------------------------------------------------
  RAS2'Move 1            Impartial
```

D.3 Quiz Model - Statistics

```
 Statistics
------------------------------------------------------------------------
  Occurrence Graph
    Nodes:  4
    Arcs:   6
    Secs:   1
    Status: Full

  Scc Graph
    Nodes:  1
    Arcs:   0
    Secs:   0


 Boundedness Properties
------------------------------------------------------------------------
  Best Integers Bounds    Upper      Lower
```

```
 New'A 1               1          0
 New'B 1               1          0
 New'SWait 1           1          0
 New'UThink 1          3          3
 New'UWait 1           1          1
 Ver01'SWait 1         1          0
 Ver01'UWait 1         1          1

 Best Upper Multi-set Bounds
 New'A 1             1`(r,Number)+ 1`(r,NoIdea)
 New'B 1             1`(r,NoIdea)
 New'SWait 1         1`c
 New'UThink 1        1`Yes+ 1`No+ 1`void
 New'UWait 1         1`r
 Ver01'SWait 1       1`c
 Ver01'UWait 1       1`r

 Best Lower Multi-set Bounds
 New'A 1             empty
 New'B 1             empty
 New'SWait 1         empty
 New'UThink 1        1`Yes+ 1`No+ 1`void
 New'UWait 1         1`r
 Ver01'SWait 1       empty
 Ver01'UWait 1       1`r


Home Properties
------------------------------------------------------------------------
 Home Markings:  All


Liveness Properties
------------------------------------------------------------------------
 Dead Markings:  None
 Dead Transitions Instances: None
 Live Transitions Instances: All

Fairness Properties
------------------------------------------------------------------------
 New'answer 1          Impartial
 New'okmsg 1           Fair
 New'prelude 1         Fair
 New'question 1        Impartial
```

D.4 Quiz Model - Occurrence Set

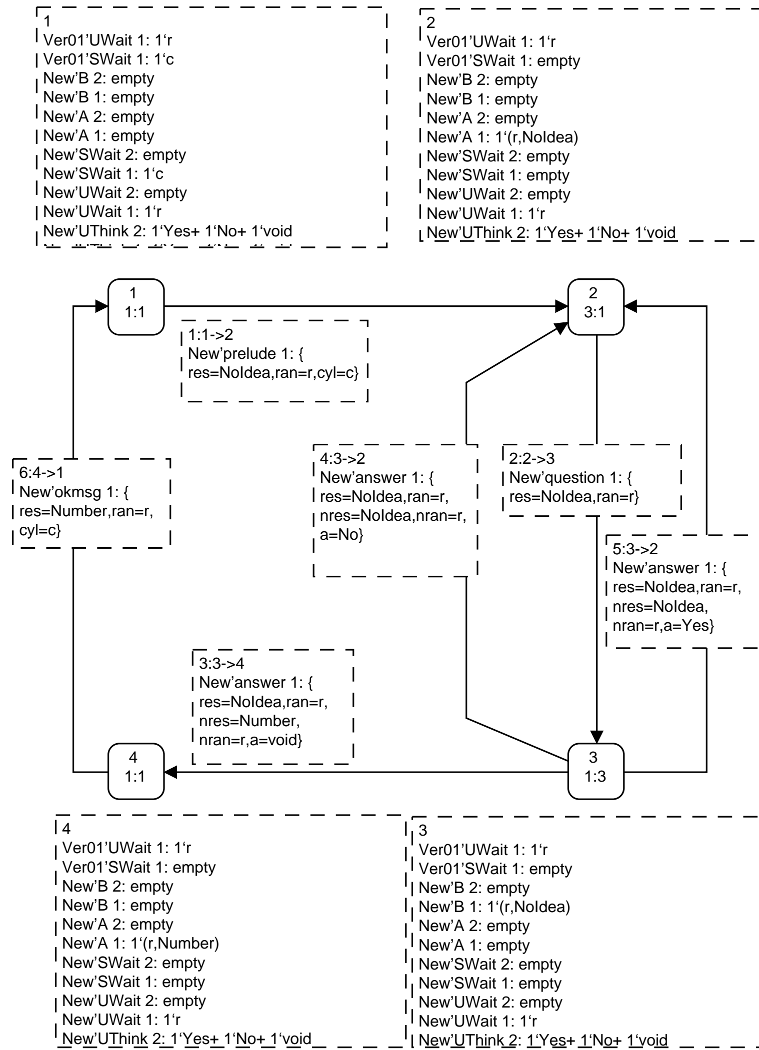


Fig. D.2: Design/CPN: quiz model, the occurrence set

E. ANALYSES RESULTS

E.1 Quiz Example - General View

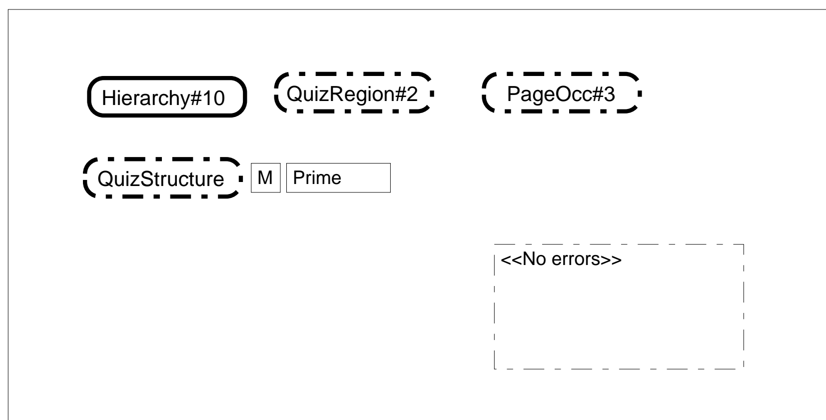


Fig. E.1: Analysis: Quiz v1.0 hierarchy

```
 Statistics
------------------------------------------------------------------------
  Occurrence Graph
    Nodes:  13
    Arcs:   18
    Secs:   0
    Status: Full

  Scc Graph
    Nodes:  1
    Arcs:   0
    Secs:   0


 Home Properties
------------------------------------------------------------------------
  Home Markings:  All


 Liveness Properties
------------------------------------------------------------------------
  Dead Markings:  None
```

<<No errors>>

```
color ElState = with elact;
color Text = string;
color UAnswer = with Yes | No | Void | Int;
color Initiative = with Start | Stop | Break;

color Range = with r | nor;
color Answer = with Found | NoIdea;
color QuizState = product Range * Answer;
color SMLState = with Srun | Sstop;
color SState = product SMLState * QuizState;

color UState = with ust;
color Element = with BIQel | SQel | Infoel | Initel | Wait;
color Argument = with arg;
color DMState = Element;

var el : Element;

var elst : ElState;
var txt,utxt, stxt : Text;
var uans, sans : UAnswer;
var inel : Initiative;
var ran : Range;
var sts : SState;
var stu : UState;
var std : DMState;


fun squestion txt =
  let
    fun getinp txt =
      DSUI_GetString {prompt=txt, def=""}
      handle _ => ""
  in
    getinp txt
  end;

fun biquestion txt =
  let
    fun getinp txt =
      DSUI_GetString {prompt=txt, def=""}
  in
    (case (getinp txt) of
  "no"  => No
 | "yes" => Yes
 | _    => Void
    ) handle _ => Int
  end;

fun NextStep (st) = case st of
      (Sstop,_) => 1`(Initel)
 | (Srun,_)  => 1`(Wait);
```

Fig. E.2: Analysis: Quiz v1.0 declarations

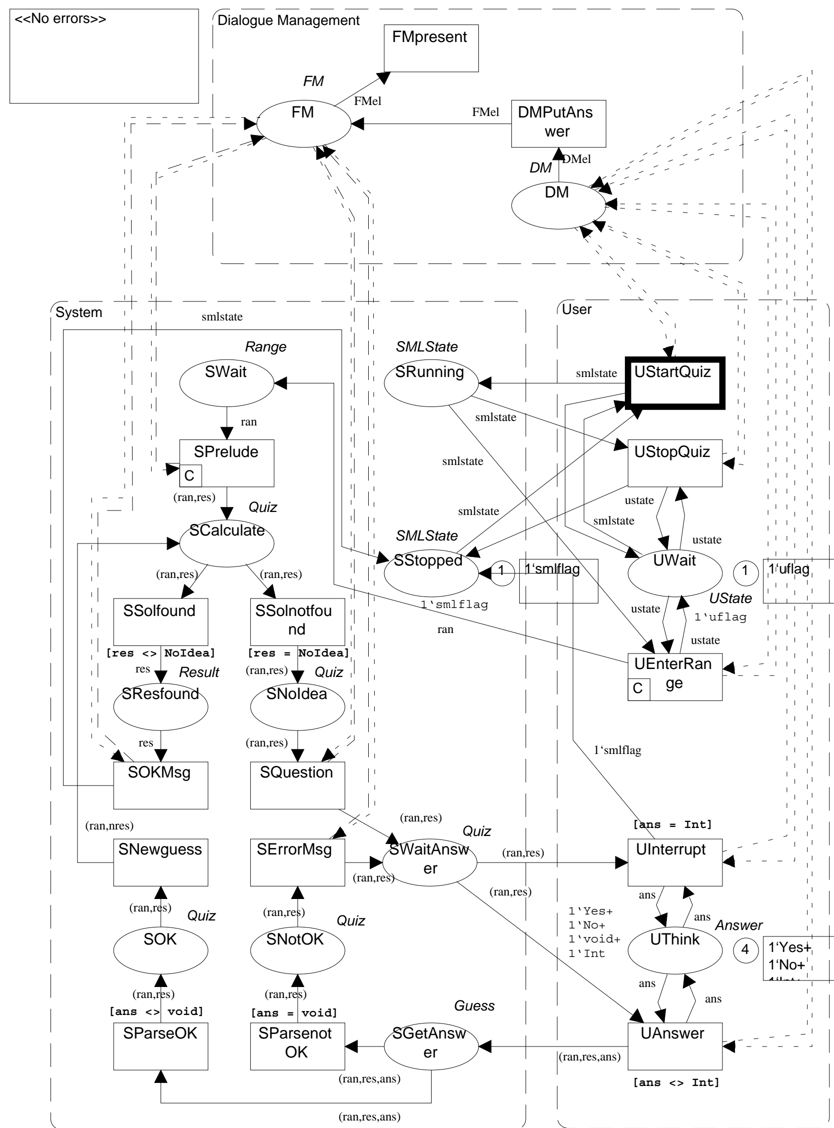


Fig. E.3: Analysis: Quiz v1.0 structure

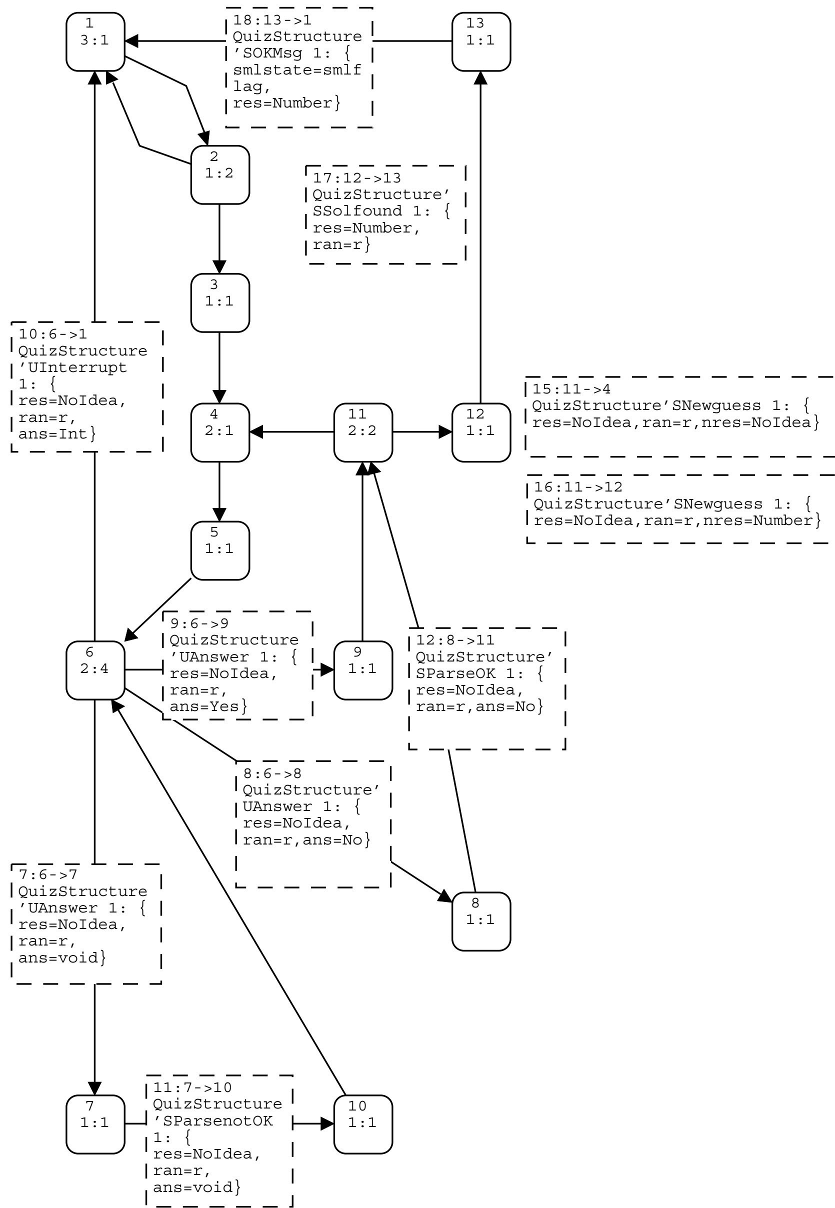


Fig. E.4: Analysis: Quiz v1.0 occurrence set

E.2 Quiz Example - Detailed View

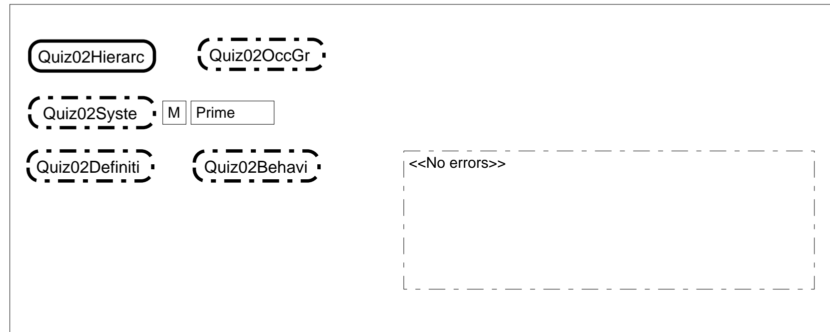


Fig. E.5: Analysis: Quiz v2.0 hierarchy

```
 Statistics
------------------------------------------------------------------------
  Occurrence Graph
    Nodes:  30
    Arcs:   36
    Secs:   20
    Status: Full

  Scc Graph
    Nodes:  2
    Arcs:   1
    Secs:   0


 Home Properties
------------------------------------------------------------------------
  Home Markings:  29 [2,3,4,6,8,...]


 Liveness Properties
------------------------------------------------------------------------
  Dead Markings:  None
```

```
   (* Quiz V 2.0 Behaviour of UI System *)

(* S-Element Functionality I *)

fun sqelement txt =
 let fun getinp txt = DSUI_GetString {prompt=txt, def=""} handle _ => ""
 in s (getinp txt) end;

(* I-Element Functionality *)

fun initelement txt =
 let fun getinp txt = DSUI_GetString {prompt=txt, def=""} handle _ => ""
 in ini (quiz (getinp txt)) end;

(* BI-Element Functionality I *)

fun bielement txt = let fun getinp txt = DSUI_GetString {prompt=txt, def=""} in
    (case (getinp txt) of "no" => bi No |  "yes" => bi Yes |  _ => bi Void
    ) handle _ => bi Int end;

(* I-Element Functionality I *)

fun ielement txt = DSUI_UserAckMessage (txt);

(* Application States Changing *)

var sst,nsst : SMLState; var qsta,nqsta: Range; var qstb,nqstb : Answer;
fun setnewstate (resp,sst,(qsta,qstb),ist) = let
  fun welcome (nl,nh) = "Think of a number between " ^ Integer.makestring(nl)
        ^ " and " ^ Integer.makestring(nh) ^ "";
  fun question (nl,nh) = "Is the number <= "
                       ^ Integer.makestring((nl+nh) div 2) ^ " ?";
  fun foundmsg n = "The number is " ^ Integer.makestring(n) ^ "";
  fun get1 lst = hd lst;
  fun get2 lst = hd (rev lst)
in
case (sst,resp,qsta,qstb) of
  (Sstop,eps,{low=nor,up=nor},NoIdea)     => (SInit,({low=nor,up=nor},NoIdea),
             is (GetRange,"Standard ML of New Jersey v110.8, August 5, 1998\nval use
= fn : string -> unit\n- "))
| (SInit,eps,{low=nor,up=nor}, qstb)      => (sst,(qsta,qstb),ist)
| (SInit,ini res,{low=nor,up=nor},qstb) => (
    if (length res) = 0
     then (Sstop,({low=nor,up=nor},NoIdea),
           is(SendRes,"stdIn:6.1-6.5 Error: unbound variable or constructor \n- "))
    else if ((get1 res) >= (get2 res))
         then (Sstop,({low=nor,up=nor},NoIdea), is (SendRes,foundmsg (get1 res)))
         else (Srun,({low=b (get1 res),up=b (get2 res)},gs(get1 res, get2 res)),
               is (SendTxt,welcome(get1 res,get2 res))))
| (Srun,eps, qsta, gs(l,u))               => (
    if (l >= u) then (Sstop,({low=nor,up=nor},NoIdea), is (SendRes,foundmsg l))
    else (Srun,(qsta,gs (l,u)), is (GetYesNo, question (l,u))))
| (Srun,bi resp, qsta, gs(l,u))           => (case resp of
  Void => (sst, (qsta,qstb),is (SendTxt,"Please answer yes or no"))
   | Int  => (Sstop,({low=nor,up=nor},NoIdea),iseps)
   | Yes  => ( if (l >= ((l+u) div 2)) then (Sstop,({low=nor,up=nor},NoIdea),
                                   is (SendRes,foundmsg ((l+u) div 2)))
      else (Srun,(qsta,gs (l,(l+u) div 2)),
                  is (GetYesNo, question (l,(l+u) div 2))))
   | No   => ( if (((l+u) div 2 + 1) >= u) then (Sstop,({low=nor,up=nor},NoIdea),
                                   is (SendRes,foundmsg ((l+u) div 2 + 1)))
      else (Srun,(qsta,gs ((l+u) div 2 + 1,u)),
                   is (GetYesNo, question ((l+u) div 2 + 1,u)))))
| _   => (sst,(qsta,qstb),ist)
end;

fun initDM (ist) = case ist of
  is (GetRange,env) => 1`(InitElem,dmenv env)
| is (GetYesNo,env) => 1`(BQIElem, dmenv env)
| is (SendTxt, env)   => 1`(IElem, dmenv env)
| is (SendRes, env) => 1`(IElem, dmenv env)
| _    => empty;
```

Fig. E.6: Analysis: Quiz v2.0 behaviour declaration

```
   (* Quiz V 2.0 - General Definitions *)

(* State of Dialogue Elements *)

color ElState = with elact; (* Element is active, passive otherwise *)
color Integer = int; color IList = list Integer; var elst : ElState;
color Text = string; var utxt,stxt : Text;

(* Possible User Response *)

color SResponse = Text; color InitResponse = IList;
color BIResponse = with Yes | No | Void | Int;
color Response = union s: SResponse + bi: BIResponse + ini: InitResponse + eps;
var resp,nresp : Response;

(* Possible States of the DM *)

color DMAct = with InitElem | SQElem | BQIElem | IElem;
color DMEnv = union dmenv : Text + dmeps; color DMState = product DMAct * DMEnv;
var dmst : DMState; var dact : DMAct; var denv : DMEnv;

(* Possible states of the App *)

color SMLState = with Srun | Sstop | SInit; color Limit = int;
color Bound = union b: Limit + nor; color Range = record low: Bound * up: Bound;
color Guess = product Limit * Limit;
color Answer = union fnd: Limit + gs: Guess + NoIdea;
color QuizState = product Range * Answer; var qst,nqst: QuizState;
color InterType = with GetRange | GetYesNo | SendTxt | SendRes;
color InterEnv = Text; color InterAction = product InterType * InterEnv;
color InterState = union is : InterAction + iseps; var ist,nist: InterState;
color AppState = product SMLState * QuizState * InterState; var appst : AppState;

(* Simulation of SML Start *)

fun quiz txt =
let
  fun nospace ([]) = [] | nospace (l::ls) = if l=" " then nospace ls
                                            else l :: (nospace ls)
  fun syntaxquiz (lst) =
    let val txt = implode lst;
      fun comma l = if l = "," then true else false
      val size = length lst; val fquiz = if substring (txt,0,4) = "quiz"
                                         then true else false
      val flb = if substring (txt,4,1) = "(" then true else false
      val frb = if substring (txt,size-2,1) = ")" then true else false
      val fsc = if substring (txt,size-1,1) = ";" then true else false
      val fco = if (size > 6) then exists comma (explode (substring(txt,5,size-7)))
                else false
    in if (fquiz andalso flb andalso frb andalso fsc andalso fco)
       then explode (substring (txt,5,size-7)) else []
    end
    fun getbound (lst) =
    let fun copos (l::ls,st) = if l = "," then st
  else copos (ls,st+1) | copos ([],st) = st;
  val txt = implode lst; val size = length lst; val pos = copos(lst,0);
       val string1 = substring(txt,0,copos(lst,0));
  val string2 = if pos > 0 then
                      substring(txt,pos+1,size-1-pos) else ""
       fun isnum st =
         let fun numtest ([]) = true
               | numtest (l::ls) = if ((ord l - 48) >= 0 andalso (ord l - 48) <= 9)
         then true andalso numtest (ls) else false
         in numtest (explode st) end;
       fun getval st = let fun calcval [] = 0
         | calcval (l::ls) = (ord l - 48) + 10 * calcval (ls);
       in calcval (explode st) end
    in if ((isnum string1) andalso (isnum string2) andalso (length lst > 0))
       then [getval string1, getval string2] else []
    end
in getbound (syntaxquiz (nospace (explode txt))) handle _ => [] : int list
end;
```

Fig. E.7: Analysis: Quiz v2.0 definitions declaration

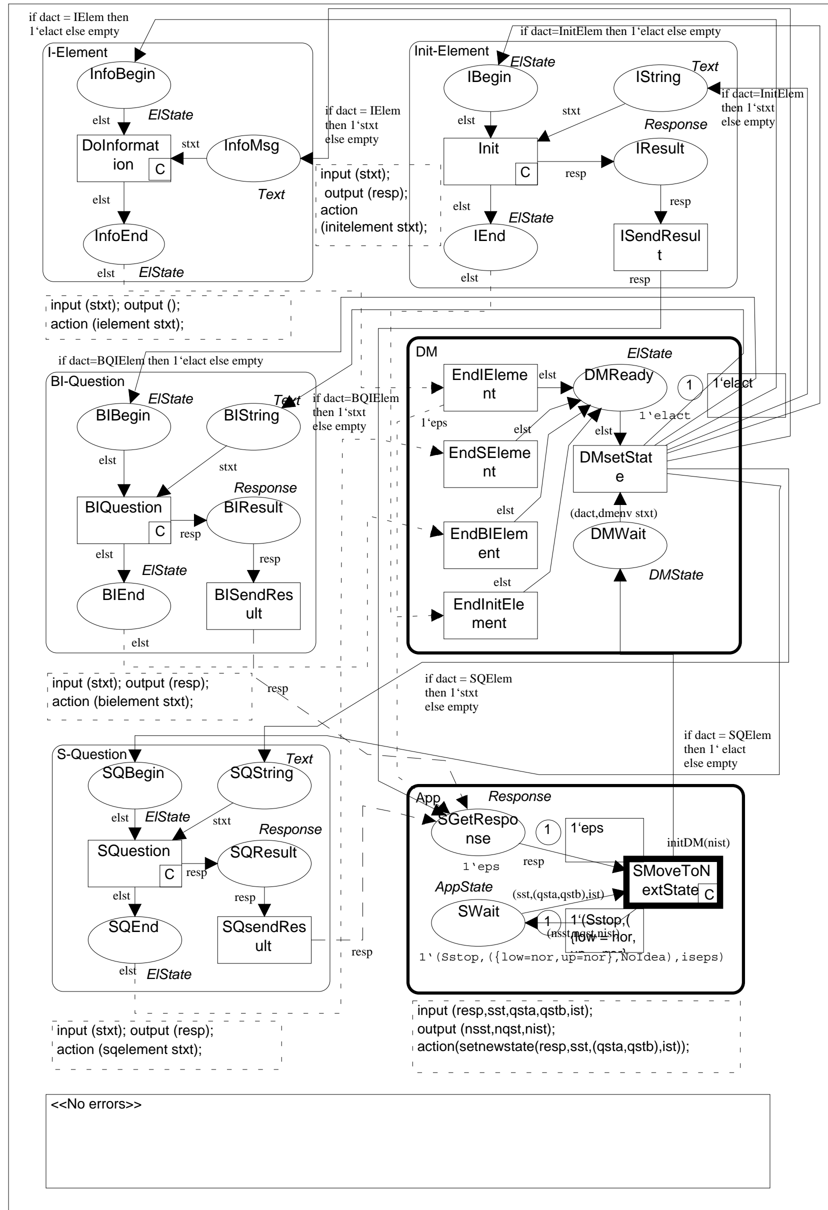


Fig. E.8: Analysis: Quiz v2.0 structure

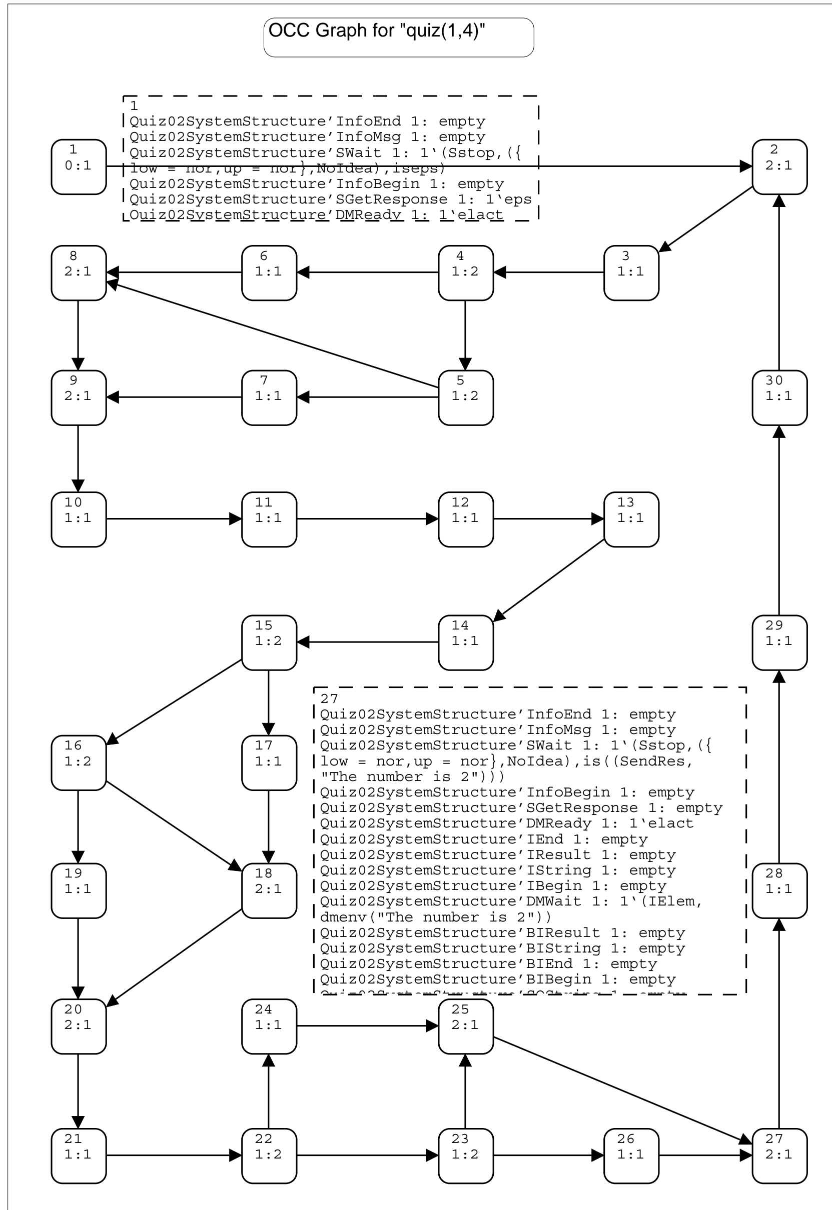


Fig. E.9: Analysis: Quiz v2.0 occurrence set

E.3 Mathematica Notebook - General View

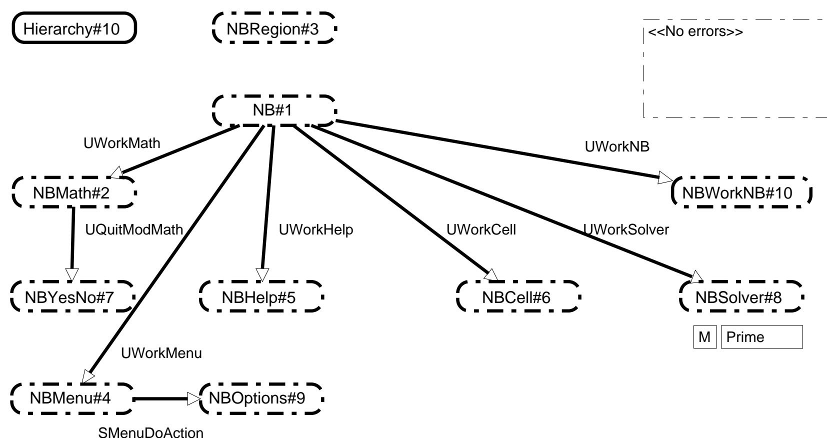


Fig. E.10: Analysis: Notebook v1.0 hierarchy

```
 Statistics
------------------------------------------------------------------------
  Occurrence Graph
    Nodes:  28
    Arcs:   51
    Secs:   1
    Status: Full

  Scc Graph
    Nodes:  4
    Arcs:   8
    Secs:   0


 Home Properties
------------------------------------------------------------------------
  Home Markings:  25 [9,22,24,10,23,...]


 Liveness Properties
------------------------------------------------------------------------
  Dead Markings:  None
```

<<No errors>>

```
color Sender = with App | DMgr | FMgr | Usr;
color Receiver = Sender;
color Type = with Info | Question | Interrupt;
color Content = string;
color FM = product Sender * Type * Content;
color DM = product Sender * Receiver * Type * Content;

color MathEnv = with menv;
color Expression = string;
color Result = with solvtext | solvgraph;
color Solver = product Expression * MathEnv * Result;

color HelpApp = with helppage;
color OptEl = with optelem;
color OptionList = list OptEl;
color OptionState = with OptMod | OptNotMod;
color Option = product OptionList * OptionState;
color Menu = with menuel;

color NBCell = with nbcell;
color NBPage = list NBCell;
color NBName = string;
color NBFlag = with nbnew | nbmod | nvsaved;
color NB = product NBPage * NBName * NBFlag;
color NBList = list NB;
color MathState = with mrun | mstop;
color Math = product MathState * NBList;
color MathAction = with mactfl;

color User = with ufl;
color UserC = with UCMenu | UCHelp | UCMath
      | UCSolver | UCCell | UCNB;
color UAction = with MStart | MQuit | MBrowse
      | MSelect | MNoSelect
      | HBrowse | HBrowseEnd
      | CBrowse | CEdit | CFinish
      | SafeYes | SafeNo
      | SolvEnter | SolvNoEnter
      | SolvInt | SolvNoInt
      | OptBrowse | OptChange
      | OptOK | OptCancel
      | NBnew | NBclose | NBload | NBsave;

var FMel : FM;
var DMel : DM;

var sol : Solver;
var hlp : HelpApp;
var opt : OptionList;
var optst : OptionState;
var men : Menu;
var nbc : NBCell;
var ma: Math;
var mst: MathState;
var nbl: NBList;
var mact: MathAction;

var u: User;
var uc: UserC;
var uact,nuact: UAction;
```

Fig. E.11: Analysis: Notebook v1.0 regions

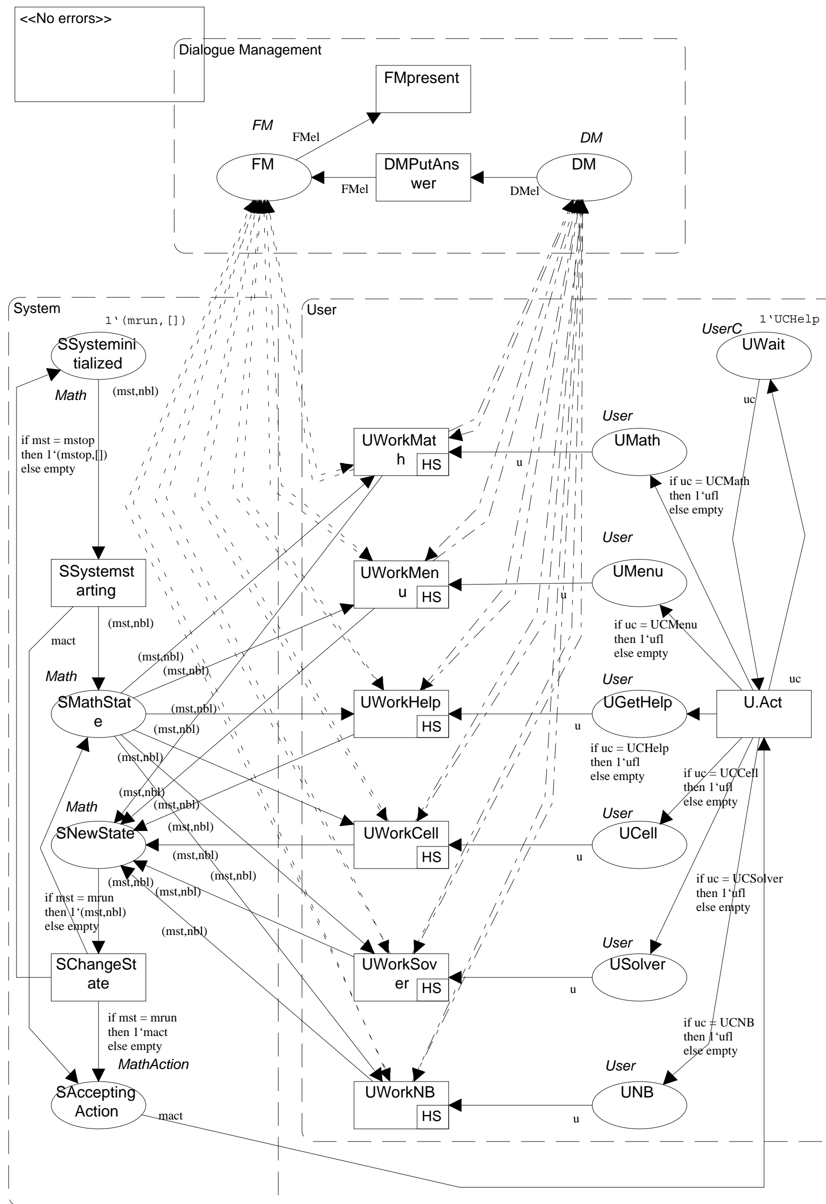


Fig. E.12: Analysis: Notebook v1.0 general structure

Fig. E.13: Analysis: Notebook v1.0 starting Mathematica

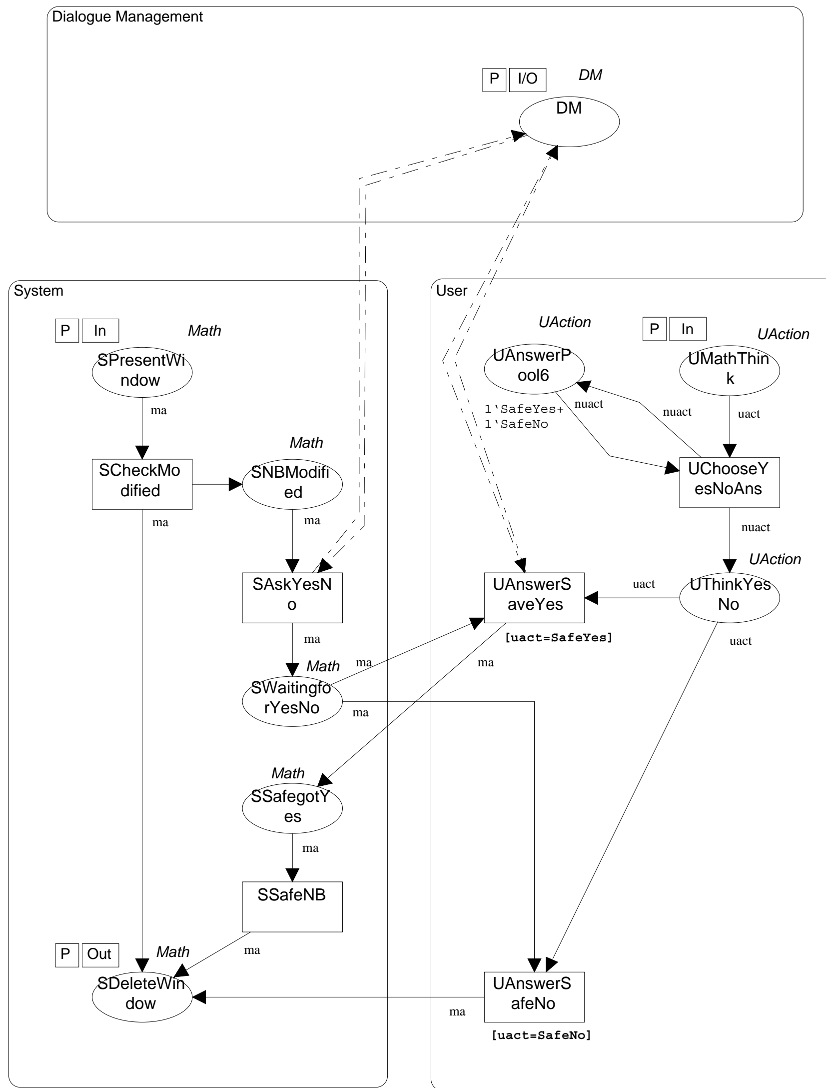


Fig. E.14: Analysis: Notebook v1.0 yes/no behaviour

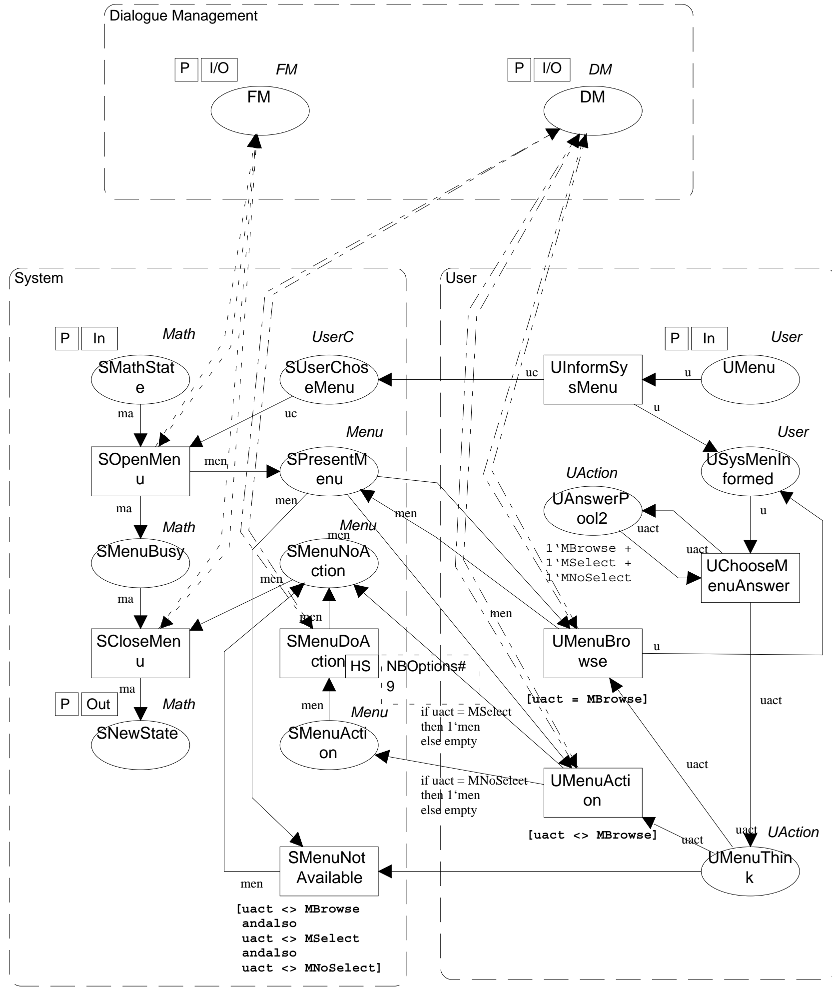


Fig. E.15: Analysis: Notebook v1.0 menu

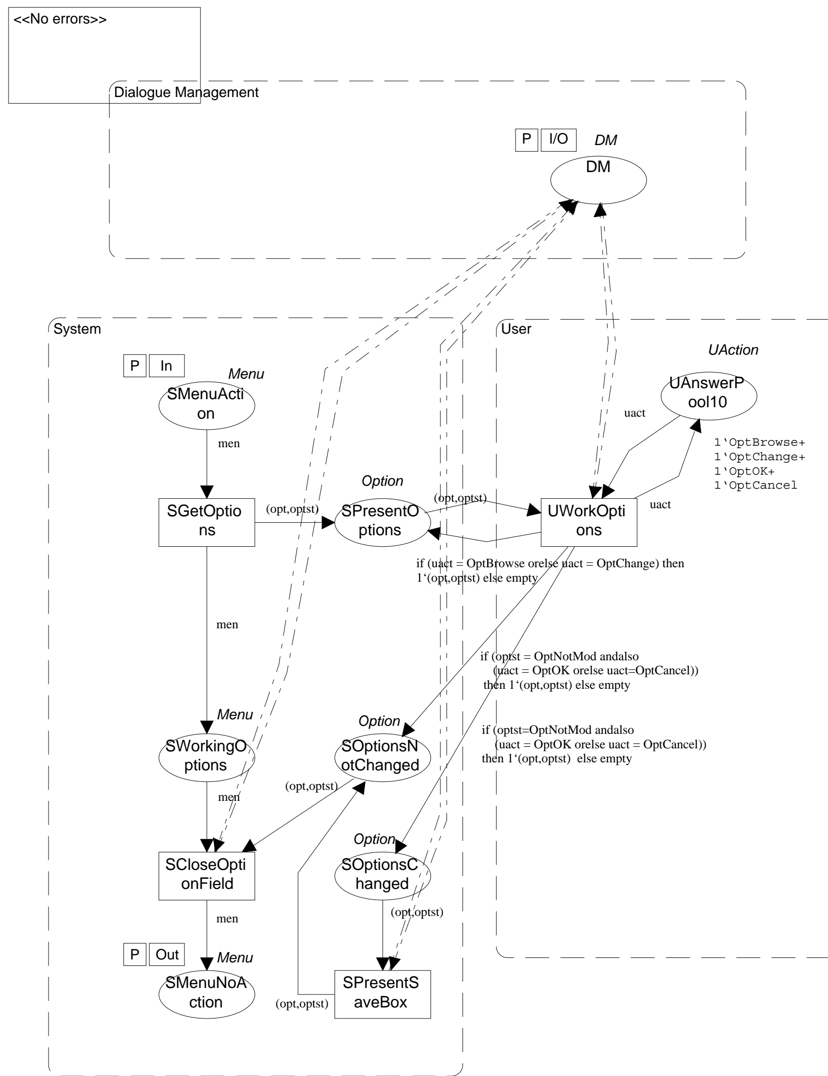


Fig. E.16: Analysis: Notebook v1.0 menu options

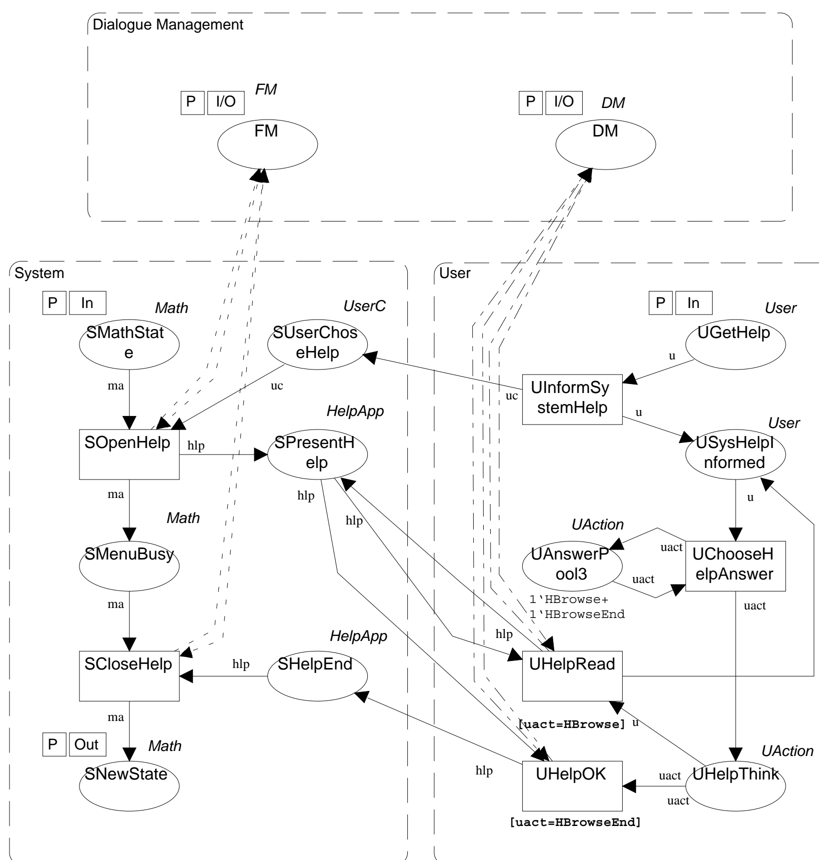


Fig. E.17: Analysis: Notebook v1.0 help system

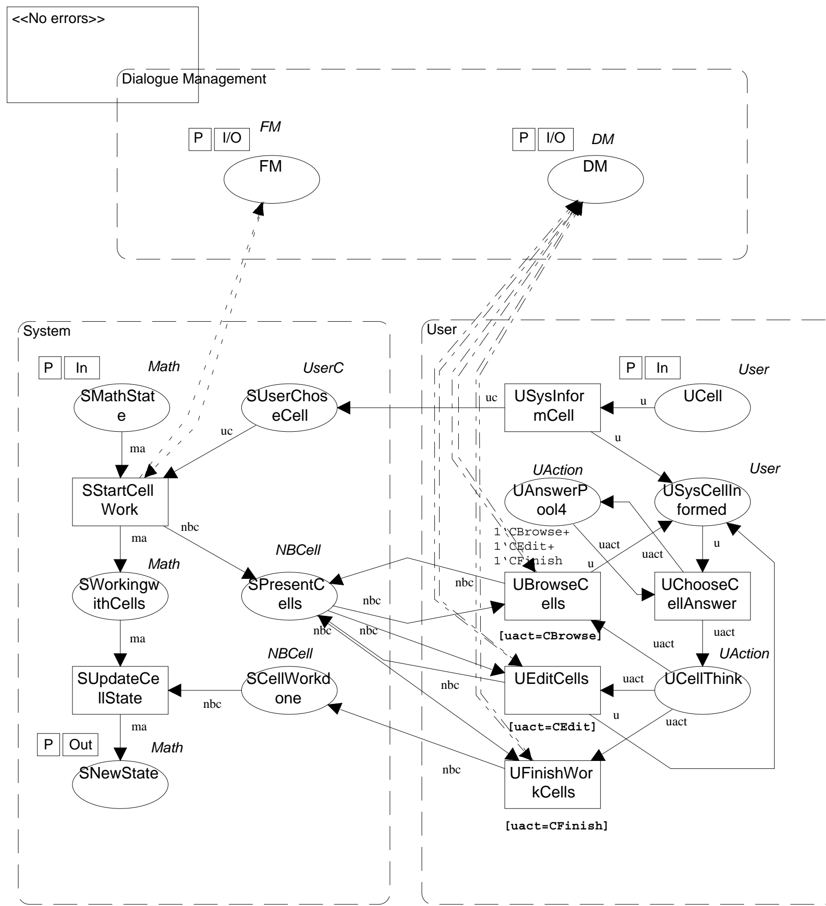


Fig. E.18: Analysis: Notebook v1.0 cell behaviour

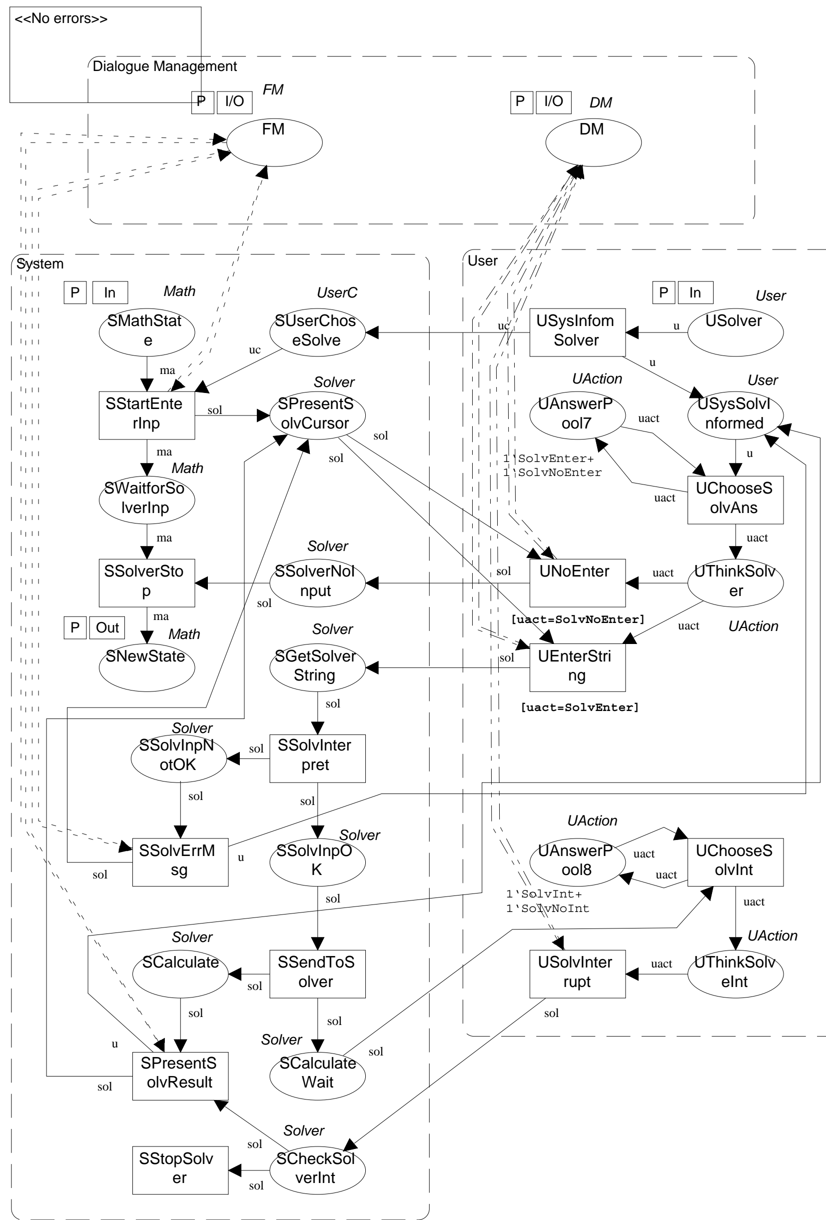


Fig. E.19: Analysis: Notebook v1.0 solver

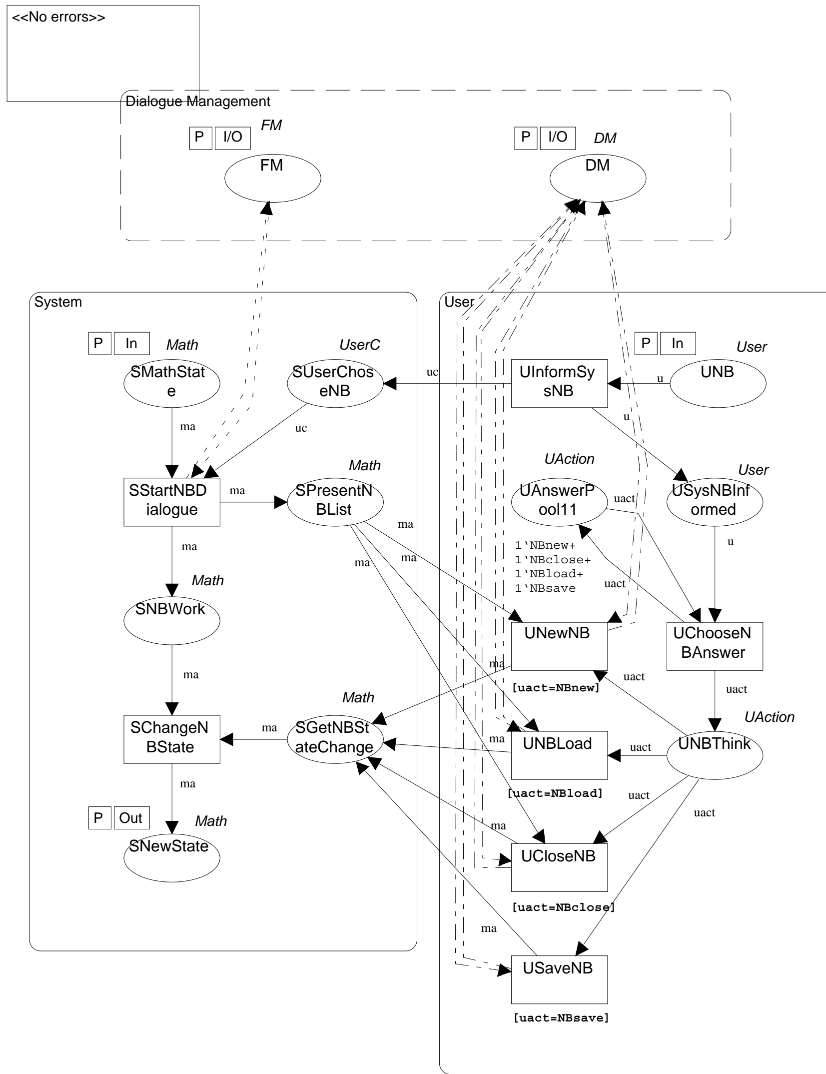


Fig. E.20: Analysis: Notebook v1.0 NB work

F. SYNTAX OF CPN/ML

The next nine pages describe the syntax of Design/CPN ML. The Design/CPN group of Aarhus University allowed the printing of this summary.

BIBLIOGRAPHY

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffry D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison - Wesley, 1986.

[AWM95] Gregory D. Abowd, Hung-Ming Wang, and Andrew F. Monk. A formal technique for automated dialogue development. In Ann Arbor, editor, *First Symposium on Designing Interactive Systems - DIS'95*, August 1995.

[Bau96] Bernd Baumgarten. *Petri-Netze*. Spektrum Akademischer Verlag, 2nd edition, 1996.

[BC92] L. Bernadinello and F. De Cindio. A survey of Basic Net Models and the Modular Net Classes. Technical report, University of Milano, 1992.

[BC94] Tommaso Bolognesi and Guiseppe Ciaccio. Cumulating constraints on the WHEN and the WHAT. *Formal Desciption Techniques*, 6, 1994.

[BS93] Dominique Borrione and Ashraf Salem. Denotational Semantics of a Synchronous VHDL Subset, 1993.

[Buc96a] Bruno Buchberger. Mathematica as a rewrite language. Technical report, Research Institute for Symbolic Computation, University of Linz, 1996. Invited paper at "The Second Fuji International Workshop on Functional and Logic Programming", November 1-4.

[Buc96b] Bruno Buchberger. Proving, Solving, Computing - A Language Environment Based on Mathematica. Invited talk at the "Multi-Paradigm Logic Programming" Workshop, Bonn, September 5-6, 1996.

[DFHP94] David Duke, Giorgio Faconti, Michael Harrison, and Fabio Paterno. Unifying Views of Interactors. *ACM*, 1994.

[DRO84] Jr. Dan R. Olsen. Pushdown Automata for User Interface Management. *ACM Transactions on Graphics*, 3(3):177–203, July 1984.

[Eis96] Richard Eisenmenger. *HTML 3.2*. Markt und Technik, 1996.

[Eng93] Hermann Engesser, editor. *Duden Informatik.* 2. Dudenverlag, 1993.

[Gre86] M. Green. A survey of three dialogue models. In *ACM Trans. Graphics*, volume 5, pages 244–275, July 1986.

[HH82] Michael Reichardt Hansen and Bo Stig Hansen. A Generic Application Programming System. Master's thesis, Instituttet for Datateknik Lyngby, February 1982.

[Jac85] R.J.K. Jacob. A State Transition Diagram Language For Visual Programming. *Computer*, 18:51–59, August 1985.

[Jen95] Kurt Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use - Volume 2.* Springer Verlag, 2nd edition, 1995.

[Jen97] Kurt Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use - Volume 1.* Springer Verlag, 2nd edition, 1997.

[Kop99] Helmut Kopka. *LaTeX Einführung Band 1*, volume 1. Addison-Wesley, 2nd edition, 1999.

[Low93] Gavin Lowe. Representing nondeterminism and probabilistic behaviour in reactive processes. Technical report, Oxford University Commputing Laboratory, 1993.

[Mar90] Lynn S. Marshall. Formally Describing Interactive Systems. *Case Studies in Semantic SW Developement*, pages 293–336, 1990. Prentice Hall.

[MJ98] Colin McCormack and David Jones. *Building a Web -Based Education System.* John Wiley & Sons, Inc., 1998.

[MS95] Herman Maurer and Nick Scherbakov. Support for teaching in a hypermedia university transaction, information and communication system. In Hermann Maurer, editor, *Educational Multimedia and Hypermedia 95*, pages 448–453. AACE, 1995.

[MTT95] K. Matsubayashi, T. Tsujino, and N. Tokura. A Denotational Approach for Formal Specification of Human-Computer Dialogue. In *Symbiosis of Human and Artifact*, pages 71 – 76. Elsevier Science B.V., 1995.

[Mye97] Brad A. Myers. *The Computer Science and Engineering Handbook*, chapter 72, pages 1571–1595. CRC Press, ACM, 1997.

[Neu97] Walther Neuper. Lern-Software und Mathematik-Systeme. Zukunftsaussichten anhand von Beispielen. IST-Softwaretechnology. Technical University Graz, Austria., January 1997.

[Nic94] Leon Eric Nicholls. Generic user interface package for windows. Master's thesis, Faculty of Science, University of Port Elizabeth, 1994.

[Nym95] Albert Nymeyer. A grammatical specification of human-computer dialogue. *Computer Languages*, 21(1):1–16, 1995.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Technische Hochschule Darmstadt, 1962.

[Rei81] Phyllis Reisner. Formal grammar and related features for defining interactive systems. In *IEEE Transactions on Software Engineering, SE-5*, pages 229–240, 1981.

[Sch86] David A. Schmidt. *Denotational Semantics - A Methodology for Language Developement.* Wm. C. Brown Publishers, 1986.

[SF85] Axel T. Schreiner and H. Georg Friedman. *Introduction to Compiler Construction with Unix.* Prentice Hall, 1985.

[Shn82] Ben Shneiderman. Multi-party grammars and related features for defining interactive systems. In *IEEE System, Man, and Cybernetics, SMC-12,2*, pages 148–154, March-April 1982.

[Shn97] Ben Shneiderman. *Designing the User Interface.* Addison Wesley, 3rd edition, 1997.

[Stö98] Harald Störrle. An evaluation of high-end tools for petri nets. Technical Report 9802, Ludwig-Maximilians-Universität München, June 1998.

[VM94] G. Viehstaedt and M. Minas. Interaction in Really Graphical User Interfaces. *IEEE*, 1994.

[Was85] Anthony I. Wassermann. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *ACM*, 1985.

[WB90] Charles Wiecha and William Bennet. ITS: A Tool for Rapidly Developing Interactive Applications. In *ACM Transactions on Information Systems*, volume 8, pages 204–236, 1990.

[Zus80] Konrad Zuse. *Petri Netze aus der Sicht des Ingenieurs.* Fried. Vieweg & Sohn, Braunschweig, 1980.

INDEX

LaTeX-ES, 86
1-safe systems, 113

ADI model, 71
ALG net, 115
Analysis
- application, 73
- dialogue manager, 73
- format manager, 73

Application
- definition of, 5
- programmer, 6
- semantics, 5

Application controller, 69
Application frameworks, 22
Application subclass, 72
Automatic generation, 23

Basic dialogue elements, 80
Basic elements
- information, 79
- initiative, 79
- navigation, 79
- question, 79

BI-Question, 75
BNF
- Backus Naur Form, 12

Building tools, 34

C/E systems, 113
Cards, 35
Classification
- approaches, 69
- Petri Nets, 42

Concurrent behaviour
- PT Net, 49

Constraint languages, 32
Construction tools, 35
Context free grammars, 12
Context subclass, 73
CP Net
- binding, 57

CPN
- advantages, 51
- analysis, 58
- arc expressions, 54
- binding, 54
- binding element, 54
- colour set, 52
- Coloured Petri Nets, 51
- CPN ML, 53
- current marking, 53
- declaration, 53
- dynamic properties, 60
- enabled transition, 54
- guard, 54
- initialisation expression, 53
- Scc graph, 61
- token colour, 52

Declarative languages, 32
Definition
- BFC net, 114
- boundedness, 60
- CP Net, 56
- EFC net, 114
- fairness, 61
- FC net, 114
- home properties, 60
- liveness, 61
- multi set, 51

net, 48
occurrence, 58
ordinary PN, 113
PT Net, 48
Scc graph, 61
state machine, 113
Denotational approach, 17
Design/CPN, 62
Desktop manager, 70
Desktop subclass, 72
Dialogue
forming subsets , 67
relevant, 67
semantical problem, 68
syntactical problem, 68
universe of discourse, 68
useful and useless, 67
Dialogue elements, 75
Dialogue manager, 69
Dialogue system, 68
Dialogue system
architecture, 70
basic sets, 68
major problems, 68
Dialogue unit, 80
Didactics
basics of, 1

Editing tools, 36
Element manager, 70
EN systems, 113
ER net, 115
Event languages, 30

FC net, 114
Feedback manager, 70

Graphical editors, 33
Graphical specification, 33

HL+ADT net, 116
HTML, 1
HyperTalk, 35

I-Element, 75, 77
Init-Element, 75, 77
Initiative elements
begin of dialogue, 79
context change, 79
dialogue type change, 79
end of dialogue, 79
general interrupt, 79
initiative change, 79
timeout, 79
Inter system controller, 71
Interface builders, 36
ITS, 23
ITS
layers of, 23

M-Question, 78
Marked graph, 115
Menu trees, 33
Methodology
parts of, 1
Multi set, 50
Multiparty BNF, 16

Nav-Element, 78
Navigation elements
context, 79
dialogue, 79
information, 79
NI-Question, 78

OBJSA net, 116
Occurrence graphs, 59

Petri Nets, 20, 29
Petri Nets
definition, 44
events, 41
graphical notation, 45
Place invariants, 59

Pr/t net, 116
Presentation subclass, 72
Problem subclass, 72
Product net, 116
Prototypes, 35

Question elements
  binary, 79
  choice, 79
  general, 79
  multiple choice, 79
  navigation, 79
  unary, 79

Reduction, 59
Regular net, 116
Resource allocation example, 45

S-Dialogue, 87
S-System, 115
Seeheim model, 3
State charts, 29
Structure subclass, 73
Style
  definition of, 24
System subclass, 72

T-System, 115
TAG
  Task action grammar, 13
Toolkits
  intrinsic layer, 37
  layers, 37
  normal, 37
  virtual, 37
Transition
  CP Net, 57
  PT Net, 48
Transition diagrams, 26

UAN
  User action notation, 21
UIMS, 6
USE approach, 27
User interface, 3
User interface
  definition of, 5
User interface designer, 6
User interface manager, 71
User interface subclass, 72
Users
  roles of, 5

Well formed net, 116

YACC
  parser, 14
  scanner, 14