Visual Programming and Modeling
Master Course

Automotive Embedded Software

Dr.ing. Loredana STANCIU, PhD
# Table of Content

## 1 VISUAL PROGRAMMING

1.1 INTRODUCTION ................................................................................................................................. 3  
1.2 HISTORY ............................................................................................................................................... 4 
1.3 STRATEGIES IN VISUAL PROGRAMMING ....................................................................................... 6 
1.4 CLASSIFICATION OF VISUAL PROGRAMMING LANGUAGES .......................................................... 7 
1.5 THEORY OF VISUAL PROGRAMMING LANGUAGES ......................................................................... 9 
1.5.1 Formal Specification of Visual Programming Languages ................................................................. 9 
1.5.2 Analysis of Visual Programming Languages .................................................................................. 10 
1.6 VISUAL LANGUAGE ISSUES .............................................................................................................. 11 
1.6.1 Control Flow ..................................................................................................................................... 11 
1.6.2 Procedural Abstraction ....................................................................................................................... 12 
1.6.3 Data Abstraction ................................................................................................................................. 12 
1.7 VPL EXAMPLES ................................................................................................................................... 12 
1.7.1 Imperative Visual Programming by Demonstration ......................................................................... 12 
1.7.2 Form/Spreadsheet Based Visual Programming ............................................................................... 13 
1.7.3 Dataflow Visual Programming ......................................................................................................... 14 
1.7.4 Rule-Based Visual Programming ...................................................................................................... 15 
1.7.5 3D VPL ............................................................................................................................................ 16 
1.8 VISUAL PROGRAMMING AND ABSTRACTION .............................................................................. 18 
1.9 CONCLUSIONS ON VISUAL PROGRAMMING ............................................................................... 18 

## 2 MODELING WITH PETRI NETS (13)

2.1 INTRODUCTION .................................................................................................................................... 20 
2.2 DESCRIPTION OF PETRI NETS .......................................................................................................... 22 
2.3 PROPERTIES OF PETRI NETS ............................................................................................................ 26 
2.3.1 Reachability ....................................................................................................................................... 26 
2.3.2 Boundedness and Safeness ............................................................................................................... 27 
2.3.3 Conservativeness .............................................................................................................................. 27 
2.3.4 Liveness ........................................................................................................................................... 28 
2.3.5 Reversibility and Home State .......................................................................................................... 30 
2.4 ANALYSIS METHODS ......................................................................................................................... 30 
2.4.1 The Coverability Tree ..................................................................................................................... 31 
2.4.2 The Incidence Matrix and State Equation ....................................................................................... 32 
2.4.3 An example ...................................................................................................................................... 34 
2.5 PETRI NETS: CONCLUSIONS ........................................................................................................... 36 

## 3 BIBLIOGRAPHY

.......................................................................................................................................................... 38
1 Visual Programming

1.1 Introduction

It is well-known that conventional programming languages are difficult to learn and use, requiring skills that many people do not have. However, there are significant advantages to supplying programming capabilities in the user interfaces of a wide variety of programs. For example, the success of spreadsheets can be partially attributed to the ability of users to write programs (as collections of “formulas”).

As the distribution of personal computers grows, the majority of computer users now do not know how to program, however. They buy computers with packaged software and are not able to modify the software even to make small changes. In order to allow the end user to reconfigure and modify the system, the software may provide various options, but these often make the system more complex and still may not address the users’ problems. “Easy-to-use” software, such as “Direct Manipulation” systems actually make the user—programmer gap worse since more people will be able to use the software (since it is easy to use), but the internal program code is now much more complicated (due to the extra code to handle the user interface).(1)

On the other hand, humans have long communicated with each other using images. The field of visual programming languages asks: why, then, do we persist in trying to communicate with our computers using textual programming languages? Would we not be more productive and would the power of modern computers not be accessible to a wider range of people if we were able to instruct a computer by simply drawing for it the images we see in our mind’s eye when we consider the solutions to particular problems? Obviously, proponents of visual programming languages argue that the answer to both these questions is yes.

The questions above highlight the primary motivations for most research into visual programming languages. First, many people think and remember things in terms of pictures. They relate to the world in an inherently graphical way and use imagery as a primary component of creative thought. Reducing or removing entirely the necessity of translating visual ideas into somewhat artificial textual representations can help to mitigate this steep learning curve problem. Furthermore, a variety of applications, including scientific visualization and interactive simulation authoring, lend themselves particularly well to visual development methods. (2)

Visual programming is programming in which more than one dimension is used to convey semantics. Examples of such additional dimensions are the use of multi-dimensional objects, the use of spatial relationships, or the use of the time dimension to specify “before-after” semantic relationships. Each potentially-significant multi-dimensional object or relationship is a token (just as in traditional textual programming languages each word is a token) and the collection of one or more such tokens is a visual expression. Examples of visual expressions used in visual programming include diagrams, free-hand sketches, icons, or demonstrations of actions performed by graphical objects. When a programming
language’s (semantically-significant) syntax includes visual expressions, the programming language is a visual programming language (VPL).

Although traditional textual programming languages often incorporate two-dimensional syntax devices in a limited way — an x-dimension to convey a legal linear string in the language, and a y-dimension allowing optional line spacing as a documentation device or for limited semantics (such as “continued from previous line”) — only one of these dimensions conveys semantics, and the second dimension has been limited to a teletype notion of spatial relationships so as to be expressible in a one-dimensional string grammar. Thus, multidimensionality is the essential difference between VPLs and strictly textual languages.

When visual expressions are used in a programming environment as an editing shortcut to generate code that may or may not have a different syntax from that used to edit in the code, the environment is called a visual programming environment (VPE). Visual programming environments for traditional textual languages provide a middle ground between VPLs and the widely-known textual languages. In contrast to just a few years ago, when strictly textual, command-line programming environments were the norm, today VPEs for traditional textual languages are the predominant kind of commercial programming environment. Commercial VPEs for traditional languages are aimed at professional programmers; these programmers use the textual languages they already know, but are supported by the graphical user interface techniques and accessibility to information that visual approaches can add. VPEs for traditional languages serve as a conduit for transferring VPL research advances into practice by applying these new ideas to traditional languages already familiar to programmers, thus affording a gradual migration from textual programming techniques to more visual ones. (3)

1.2 History

The field of visual programming has grown from a marriage of work in computer graphics, programming languages, and human-computer interaction. It should come as no surprise, then, that much of the seminal work in the field is also viewed as pioneering work in one of the other disciplines. Ivan Sutherland’s ground-breaking Sketchpad system stands out as the best example of this trend. Sketchpad, designed in 1963 on the TX-2 computer at MIT, has been called the first computer graphics application. The system allowed users to work with a lightpen to create 2D graphics by creating simple primitives, like lines and circles, and then applying operations, such as copy, and constraints on the geometry of the shapes. Its graphical interface and support for user-specifiable constraints stand out as Sketchpad’s most important contributions to visual programming languages. By defining appropriate constraints, users could develop structures such as complicated mechanical linkages and then move them about in real time. Ivan Sutherland’s brother, William, also made an important early contribution to visual programming in 1965, when he used the TX-2 to develop a simple visual dataflow language. The system allowed users to create, debug, and execute dataflow diagrams in a unified visual environment.
The next major milestone in the genesis of VPLs came in 1975 with the publication of David Canfield Smith’s PhD dissertation entitled “Pygmalion: A Creative Programming Environment”. Smith’s work marks the starting point for a number of threads of research in the field which continue to this day. For example, Pygmalion embodied an icon-based programming paradigm in which the user created, modified, and linked together small pictorial objects, called icons, with defined properties to perform computations. Much work has since gone into formalizing icon theory, as will be discussed below, and many modern VPLs employ an icon-based approach. Pygmalion also made use of the concept of programming-by-example wherein the user shows the system how to perform a task in a specific case and the system uses this information to generate a program which performs the task in general cases. In Smith’s system, the user sets the environment to “remember” mode, performs the computation of interest, turns off “remember” mode, and receives as output a program, in a simple assembly-like subset of Smalltalk, which performs the computation on an arbitrary input. (2)

The earliest work in visual programming was in two directions: visual approaches to traditional programming languages (such as executable flowcharts), and new visual approaches to programming that deviated significantly from traditional approaches (such as programming by demonstrating the desired actions on the screen). Many of these early systems had advantages that seemed exciting and intuitive when demonstrated with “toy” programs, but ran into difficult problems when attempts were made to extend them to more realistically-sized programs. These problems led to an early disenchantment with visual programming, causing many to believe that visual programming was inherently unsuited to “real” work and it was just an academic exercise.

To overcome these problems, visual programming researchers began to develop ways to use visual programming for only selected parts of software development, thereby increasing the number of projects in which visual programming could help. In this approach, straightforward visual techniques were widely incorporated into programming environments that support textual programming languages, to replace cumbersome textual specification of GUI layout, to support electronic forms of software engineering diagrams for creating and/or visualizing relationships among data structures, and to visually combine textually-programmed units to build new programs. Successful commercial VPEs soon followed; among the early examples were Microsoft’s Visual Basic (for Basic) and ParcPlace Systems’ VisualWorks (for Smalltalk). Another group of commercial VPEs, focused primarily on large-grained programming, are the Computer-Aided Software Engineering (CASE) tools that support visual specification (for example, using diagrams) of relationships among program modules, culminating in automatic code generation of composition code.

Other visual programming researchers took a different approach — they worked to increase the kinds of projects suitable for visual programming through the development of domain-specific visual programming systems. Under this strategy, the addition of each new supported domain increased the number of projects that could be programming visually. An added benefit that followed was improved accessibility — end-users were sometimes able to use these new systems. The developers of domain-specific VPLs and VPEs found that providing ways to write programs for one particular problem domain eliminated many of the disadvantages found in the earliest approaches, because they supported
working directly in the communication style of the particular problem domain—using visual artifacts (e.g., icons and menus) reflecting the particular needs, problem-solving diagrams, and vocabulary specific to that domain — and never forced users to abandon that communication style. This approach quickly produced a number of successes both in research and in the marketplace. Today there are commercial VPLs and VPEs available in many domains; examples include programming laboratory data acquisition (National Instruments’ LabVIEW), programming scientific visualizations (Advanced Visual Systems’ AVS), programming telephone and voice-mail behavior (Cypress Research’s PhonePro), and programming graphical simulations and games (Stagecoach Software’s Cocoa). A number of software-agent generators are starting to become embedded in personal computing software as well, allowing macros that assist with repetitive tasks to be inferred from end-user manipulations (as in Chimera, for example).

The original challenge — to devise VPLs with enough power and generality to address an ever-expanding variety of programming problems — is an ongoing area of research. One goal of this research is to continue to improve the ways visual programming can be used. Another goal is to provide the same kinds of improvements in general software development as are already available for programming in some domain-specific areas. But although this work is still primarily in the research stage, commercial VPLs with the characteristics needed for general-purpose programming have emerged and are being used to produce commercial software packages; one example is Pictorius International’s Prograph CPX. (3)

1.3 Strategies in Visual Programming

A common misunderstanding is that the goal of visual programming research in general and VPLs in particular is to eliminate text. This is a fallacy — in fact, most VPLs include text to at least some extent, in a multidimensional context. Rather, the overall goal of VPLs is to strive for improvements in programming language design. The opportunity to achieve this comes from the simple fact that VPLs have fewer syntactic restrictions on the way a program can be expressed (by the computer or by the human), and this affords a freedom to explore programming mechanisms that have not previously been tried because they have not been possible in the past.

The most common specific goals sought with VPL research have been:

- to make programming more accessible to some particular audience,
- to improve the correctness with which people perform programming tasks
- to improve the speed with which people perform programming tasks.

To achieve these goals, there are four common strategies used in VPLs:

- **Concreteness:** is the opposite of abstractness, and means expressing some aspect of a program using particular instances. One example is allowing a programmer to specify some aspect of semantics on a specific object or value, and another example is having the system automatically display the effects of some portion of a program on a specific object or value.
• **Directness:** in the context of direct manipulation is usually described as “the feeling that one is directly manipulating the object”. From a cognitive perspective, directness in computing means a small distance between a goal and the actions required of the user to achieve the goal. Given concreteness in a VPL, an example of directness would be allowing the programmer to manipulate a specific object or value directly to specify semantics rather than describing these semantics textually.

• **Explicitness:** some aspect of semantics is explicit in the environment if it is directly stated (textually or visually), without the requirement that the programmer infer it. An example of explicitness in a VPL would be for the system to explicitly depict dataflow relationships (program slice information) by drawing directed edges among related variables.

• **Immediate Visual Feedback:** in the context of visual programming, immediate visual feedback refers to automatic display of effects of program edits. Tanimoto has coined the term *liveness*, which categorizes the immediacy of semantic feedback that is automatically provided during the process of editing a program. Tanimoto described four levels of liveness. At level 1 no semantics are implied to the computer, and hence no feedback about a program is provided to the programmer. An example of level 1 is an entity-relationship diagram for documentation. At level 2 the programmer can obtain semantic feedback about a portion of a program, but it is not provided automatically. Compilers support level 2 liveness minimally, and interpreters do more so because they are not restricted to final output values. At level 3, incremental semantic feedback is automatically provided whenever the programmer performs an incremental program edit, and all affected onscreen values are automatically redisplayed. This ensures the consistency of display state and system state if the only trigger for system state changes is programmer editing. The automatic recalculation feature of spreadsheets supports level 3 liveness. At level 4, the system responds to program edits as in level 3, and to other events as well such as system clock ticks and mouse clicks over time, ensuring that all data on display accurately reflects the current state of the system as computations continue to evolve.

### 1.4 Classification of Visual Programming Languages

As the field of VPLs has matured, more and more interest has been focused on creating a robust, standardized classification for work in the area. Such a classification system not only aids researchers in finding related work but also provides a baseline with which to compare and evaluate different systems. Some of the most important names in the field, including Chang, Shu, and Burnett, have worked on identifying the defining characteristics of the major categories of VPLs. The following presents a summary of the classification scheme discussed below:

• Purely visual languages
• Hybrid text and visual systems
• Programming-by-example systems
• Constraint-oriented systems
• Form-based systems

Note that the categories are by no means mutually exclusive. Indeed, many languages can be placed in more than one category.

The single most important category has to be purely visual languages. Such languages are characterized by their reliance on visual techniques throughout the programming process. The programmer manipulates icons or other graphical representations to create a program which is subsequently debugged and executed in the same visual environment. The program is compiled directly from its visual representation and is never translated into an interim text-based language. Examples of such completely visual systems include VIPR, Prograph, and PICT. In much of the literature in the field, this category is further subdivided into sections like iconic and non-iconic languages, object-oriented, functional, and imperative languages.

One important subset of VPLs attempts to combine both visual and textual elements. These hybrid systems include both those in which programs are created visually and then translated into an underlying high-level textual language and systems which involve the use of graphical elements in an otherwise textual language. Examples in this category include Rehearsal World and work by Erwig et. al (5). In the former, the user trains the system to solve a particular problem by manipulating graphical “actors,” and then the systems generates a Smalltalk program to implement the solution. The latter involves work on developing extensions to languages like C and C++which allow programmers to intersperse their text code with diagrams. For instance, one can define a linked list data structure textually and then perform an operation like deletion of a node by drawing the steps in the process.

In addition to these two major categories, many VPLs fall into a variety of smaller classifications. For example, a number of VPLs follow in the footsteps of Pygmalion by allowing the user to create and manipulate graphical objects with which to “teach” the system how to perform a particular task. Rehearsal World, described above, fits into this category of programming by example. Some VPLs can trace their lineage back, in part, to Sutherland’s constraint manipulations in Sketchpad. These constraint-oriented systems are especially popular for simulation design, in which a programmer models physical objects as objects in the visual environment which are subject to constraints designed to mimic the behavior of natural laws, like gravity. Constraint-oriented systems have also found application in the development of graphical user interfaces. Thinglab and ARK, both primarily simulation VPLs, stand out as quintessential examples of constraint-based languages.

A few VPLs have borrowed their visualization and programming metaphors from spreadsheets. These languages can be classified as form-based VPLs. They represent programming as altering a group of interconnected cells over time and often allow the programmer to visualize the execution of a program as a sequence of different cell states which progress through time. Forms/3 is the current incarnation of the progenitor of this type of VPL. It is important to note that in each of the categories mentioned above, one can find examples of both general-purpose VPLs and languages designed for domain-specific applications.
The field of visual programming has evolved greatly over the last ten years. Continual development and refinement of languages in the categories discussed above have led to some work which was initially considered to be part of the field being reclassified as related to but not actually exemplifying visual programming. These VPL orphans, so to speak, include algorithm animation systems, such as BALSA, which provide interactive graphical displays of executing programs and graphical user interface development tools, like those provided with many modern compilers including Microsoft Visual C++. Both types of systems certainly include highly visual components, but they are more graphics applications and template generators than actual programming languages. (2)

1.5 Theory of Visual Programming Languages

To set up the framework for the discussion regarding the theoretical advances in the field of Visual Programming Languages, some definitions from (6) will be presented:

- **icon (generalized icon)**: an object with the dual representation of a logical part (the meaning) and a physical part (the image).
- **iconic system**: A structured set of related icons.
- **iconic sentence (visual sentence)**: A spatial arrangement of icons from iconic system.
- **visual language**: A set of iconic sentences constructed with given syntax and semantics.
- **syntactic analysis (spatial parsing)**: An analysis of an iconic sentence to determine the underlying structure.
- **semantic analysis (spatial interpretation)**: An analysis of an iconic sentence to determine the underlying meaning.

The following discussion is restricted to two-dimensional visual languages, although everything that follows can be generalized to three (and more) dimensions.

1.5.1 Formal Specification of Visual Programming Languages

A spatial arrangement of icons that constitutes a visual sentence is a two-dimensional counterpart of a one-dimensional arrangement of tokens in conventional (textual) programming languages. In those languages, a program is expressed as a string in which terminal tokens are concatenated to form a sentence whose structure and meaning are discovered by syntactic and semantic analysis, respectively. Thus, the construction rule is implicit in the language and need not be spelled-out as part of the language specification. Conversely, in visual programming languages we distinguish three construction rules that are used to arrange icons: horizontal concatenation (denoted by &), vertical concatenation (denoted by ^), and spatial overlay (denoted by +).

In formalizing visual programming languages, it is customary to distinguish process icons from object icons. The former express computations; the latter can be further subdivided into elementary object icons and composite object icons. The elementary object icons identify primitive objects in the language, whereas the composite object icons identify objects formed by a spatial arrangement of the elementary object icons. Finally, the term elementary icons is used to refer to both process icons and elementary
object icons and denotes those icons that are primitives in the language. Since a picture (or, icon, in our case) is worth a thousand words, all of the above concepts are illustrated in

![Visual Programming and Modeling](image)

**Fig. 1.1** which demonstrates a few icons from the Heidelber icons set(7) and a complete visual sentence.

A visual programming language is specified by a triple (ID,G0,B), where ID is the icon dictionary, G0 is a grammar, and B is a domain-specific knowledge base (8). The icon dictionary is the set of generalized icons each of which is represented by a pair (Xm,Xi), with a logical art Xm (the meaning) and a physical part Xi (the image). The grammar G0 specifies how composite object icons may be constructed from elementary icons by using spatial arrangement operators. Note that we need to specify spatial composition operators as terminals in the grammar precisely because they are no longer implicit in the language definition. The knowledge base B contains domain-specific information necessary for constructing the meaning of a given visual sentence. It contains information regarding event names, conceptual relations, names of resulting objects, and references to the resulting objects.
1.5.2 Analysis of Visual Programming Languages

As discussed above, visual sentences are constructed from elementary icons using iconic operators. The syntactic analysis of visual sentences (also known as spatial parsing (9)) is based upon a number of approaches (6).

- Picture-processing grammars: Originally designed to parse digital pictures on a square grid, these grammars are based on the fact that digital pictures are composed of pixels. These grammars discover the structure of visual sentence by composing individual pixels into recognizable visual elements (lines, arks, etc.) (10). This approach is useful when an iconic system needs to be able to recognize icons with a certain level of error tolerance (e.g. handwritten digits).

- Precedence grammars: This spatial parsing grammar can be used for two-dimensional mathematical expression analysis and printed-page analysis. Precedence grammars are more suitable for syntactic analysis of visual sentences constructed from elementary icons and iconic operators. The parse tree is constructed by comparing precedences of operators in a pattern and subdividing the pattern into one or more subpatterns.

- Context-free and context-dependent grammars: These grammars are used to specify composition of visual sentences using familiar formalisms, and so many standard methods of parsing such grammars are applicable.
• Graph grammars: These are by far the most powerful (albeit least efficient) specifications of visual languages. These formalisms provide for the most means for establishing context relationships and much recent work has been devoted to making parsing with graph grammars computationally feasible.

A parse tree produced by one of the above parsing methods is subsequently analyzed using traditional approaches to semantic analysis (e.g. attribute grammars, ad-hoc tree computations, etc.).

1.6 Visual Language Issues

Here are discussed some common language issues in light of which the following presentation of visual languages is cast (3). These issues are mostly applicable to general-purpose visual languages (suitable for producing executable programs of reasonable size), although certain issues will also be relevant to domain-specific languages (designed to accommodate a particular domain such as software engineering or scientific visualization).

1.6.1 Control Flow

Similarly to conventional programming languages, visual languages embrace two notions of flow of control in programs: imperative and declarative.

With the imperative approach, a visual program constitutes one or more control-flow or dataflow diagrams which indicate how the thread of control flows through the program. A particular advantage of such approach is that it provides an effective visual representation of parallelism. A disadvantage of this method is that a programmer is required to keep track of how sequencing of operations modifies the state of the program, which is not always an intended feature of the system (especially if it is designed to accommodate novices).

An alternative to imperative semantics of flow control is to use a declarative style of programming. With this approach, one only needs to worry what computations are performed, and not how the actual operations are carried out. Explicit state modification is avoided by using single assignment: a programmer creates a new object by copying an existing one and specifying the desired differences, rather than modifying the existent object’s state. Also, instead of specifying a sequence of state changes, the programmer defines operations by specifying object dependencies. For example, if the programmer defines \( Y \) to be \( X +1 \), this explicitly states that \( Y \) is to be computed using object in \( X \), allowing the system to infer that \( X \)'s value needs to be computed first. Thus, the sequencing of operations is still present, but must be inferred by the system rather than defined by the programmer. Of course, special care must be taken by the system that circular dependencies are detected and signaled as errors.

1.6.2 Procedural Abstraction

There are two levels of procedural abstraction. High-level visual programming languages are not complete programming languages, i.e. it is not possible to write and maintain an entire program in such a
language and inevitably there’s some underlying non-visual modules that are combined using a visual language. This approach to visual programming is found in various domain-specific systems such as software maintenance tools and scientific visualization environments. At the opposite end of the scale, are low-level visual languages which do not allow the programmer to combine fine-grained logic into procedural modules. This methodology is also useful in various domain-specific languages such as logic simulators. General-purpose visual programming languages normally cover the entire spectrum of programming facilities ranging from low-level features, including conditionals, recursion, and iteration, to high-level facilities that allow one to combine low-level logic into abstract modules (procedures, classes, libraries, etc.).

1.6.3 Data Abstraction

Data abstraction facilities are only found in general-purpose programming languages. The notion of data abstraction in visual programming is very similar to the notion of data abstraction in conventional programming languages, with the only requirements being that abstract data types be defined visually (as opposed to textually), have a visual (iconic) representation, and provide for interactive behavior.

1.7 VPL Examples

In this section, four example VPLs will be presented to demonstrate several ways in which the strategies of the previous section have been employed.

1.7.1 Imperative Visual Programming by Demonstration

Chimera is an example of the most common way imperative programming is supported in VPLs, namely by having the programmer demonstrate the desired actions. In the case of Chimera, the “programmer” is an end user: hence, Chimera is an example of a VPL aimed at improving accessibility of programming certain kinds of tasks.

The domain of Chimera is graphical editing. As an end user works on a graphical scene, he or she may find that repetitive editing tasks arise, and can indicate that a sequence of manipulations just performed on a scene should be generalized and treated as a macro. This is possible because the history of the user’s actions is depicted using a comic strip metaphor (Fig. 1.2), and the user can select panels from the history, indicate which of the objects should be viewed as example “parameters,” (graphically) edit the actions depicted in any of the panels if desired, and finally save the sequence of edited panels as a macro. Chimera uses inference in determining the generalized version of the macro; use of inference is common in by-demonstration languages, and its success depends on limited problem domains such as Chimera’s. However, there are also a number of by-demonstration languages that do not use inference, one example of which is Cocoa.
Fig. 1.2 Programming by demonstration in Chimera. In this example, the user has drawn a box with an arrow pointing to it (as in a graph diagram), and this demonstration is depicted after-the-fact in a series of intelligently-filtered panels. This set of demonstrations can be generalized into a macro for use in creating the other nodes in the graph semi-automatically.

1.7.2 Form/Spreadsheet Based Visual Programming

Forms/3 is an example of a VPL that follows the form-based paradigm. In this paradigm, a programmer programs by creating a form and specifying its contents. This paradigm is most commonly seen in commercial spreadsheets, in which the form is grid-shaped, and the contents are specified by the cells’ formulas.

Forms/3 programs include forms (spreadsheets) with cells, but the cells are not locked into a grid. A Forms/3 programmer creates a program by using direct manipulation to place cells on forms, and defines a formula for each cell using a flexible combination of pointing, typing, and gesturing (Fig. 1.3). A program’s calculations are entirely determined by these formulas. The formulas combine into a network of (one-way) constraints, and the system continuously ensures that all values displayed on the screen satisfy these constraints.

Forms/3 is a Turing-complete language. The aim is to enhance the use of ordinary spreadsheet concepts to support the advanced functionality needed for full-featured programming. Thus it supports such features as graphics, animation, and recursion, but without resorting to state-modifying macros or links to traditional programming languages. For example, Forms/3 supports a rich and extensible collection of types by allowing attributes of a type to be defined by formulas, and an instance of a type to be the value of a cell, which can be referenced just like any cell. In (Fig. 1.3), an instance of type “box” is being specified by graphically sketching it; this specification can be changed if necessary by stretching the box by direct manipulation. Immediate visual feedback at liveness level 4 is provided in either case. Concreteness is present in the fact that the resulting box is immediately seen when enough formulas have been provided to make this possible; directness is present in the direct-manipulation mechanism for specifying a box because one demonstrates the specification directly on the box.
Fig. 1.3 Defining the area of a square using spreadsheet-like cells and formulas in Forms/3. Graphical types are supported as first-class values, and the programmer can enter cell square’s formula either by sketching a square box or by typing textual specifications (e.g., “box 30 30”).

The intended audience for Forms/3 is “future” programmers — those whose job will be to create applications, but whose training has not emphasized today’s traditional programming languages. A goal of Forms/3 has been to reduce the number and complexity of the mechanisms required to do application programming, with the hope that greater ease of use by programmers will result than has been characteristic of traditional languages, with an accompanying increase in correctness and/or speed of programming. In empirical studies, programmers have demonstrated greater correctness and speed in both program creation and program debugging using Forms/3’s techniques than when using a variety of alternative techniques.

1.7.3 Dataflow Visual Programming

Prograph is a dataflow VPL aimed at professional programmers. The dataflow paradigm is currently the approach to visual programming used most widely in industry. Prograph exemplifies its use for programming at all levels, from low-level details that can be grouped into procedures and objects (Fig. 1.4), to compositions of procedures and objects. The dataflow paradigm is also commonly used by domain-specific VPEs for composition of low-level components that have been written some other way; for example, scientific visualization systems and simulation systems often make heavy use of visual dataflow programming.

Prograph provides strong debugging support by making extensive use of dynamic visualization techniques. The liveness level is 2 for the data values themselves — the programmer explicitly requests display of a value each time he/she wants to see it. However, the runtime stack activity and the order in which nodes fire can be viewed throughout execution, and if the programmer changes a bit of data or source code mid-execution, the stack window and related views automatically adjust to proceed from that point on under the new version, and this aspect is liveness level 3.
One way in which the dataflow paradigm distinguishes itself from many other paradigms is through its explicitness (through the explicit rendering of the edges in the graph) about the dataflow relationships in the program. Since many dataflow languages govern even control flow by dataflow, these edges are also sufficient to reflect control flow explicitly in a purely dataflow language.

### 1.7.4 Rule-Based Visual Programming

Cocoa (formerly known as KidSim) is a rule-based VPL in which the programmer specifies the rules by demonstrating a postcondition on a precondition (Fig. 1.5). The intended “programmers” are children, and the problem domain is specification of graphical simulations and games. Cocoa is a Turing-complete language, but its features have not been designed to make general-purpose programming convenient; rather, it has been designed to make accessible to children the ability to program their own simulations.

The way concreteness and directness are seen in Cocoa is quite similar to Chimera, since both use by-demonstration as the way semantics are specified. The liveness level is different though; in Cocoa, liveness is between level 2 and level 3. It is not level 3 for some kinds of program changes (e.g., addition of new rules) that do not affect the current display of variables until the child requests that the program resume running, but for other kinds of program changes (e.g., changing the appearance of an object), the changes are automatically propagated into the display immediately.

In listing the properties common to rule-based systems, Hayes-Roth includes the ability to explain their behavior (11). In Cocoa, a child can open (by selecting and double-clicking) any character participating in the simulation, and a window containing the rules governing that character’s behavior is displayed, as in the figure. In each execution cycle, each character’s rules are considered top-down in the character’s list. The indicators next to each rule are “off” (gray) prior to a rule being considered. Then, if the rule-matching fails, the indicator next to the rule turns red; if the pattern-matching succeeds, the rule fires, the indicator next to it turns green. Once a rule has fired for a character, that character’s “turn” if over, and no more rules for that character are checked until the next cycle. (3)
Fig. 1.5  A Cocoa wall-climber (The Wall Climber: Main window) is following the rules (Mascot 1 window) that have been demonstrated for it. Each rule is shown with the graphical precondition on the left of the arrow and the graphical postcondition on the right of the arrow. The wall climber has just finished following rule 2, which places it in a position suitable for following rule 1 next.

1.7.5  3D VPL

Cube, by M. Najork, represents an important advance in the design of visual programming languages in that it is the first three dimensional VPL. Since Cube programs are translated into simpler internal representations for type checking and interpreting, the language would fall into the category of a hybrid according to our taxonomy. However, the user is never exposed to any textual representation, so the argument could be made that Cube comes very close to being a completely visual language. The language uses a dataflow metaphor for program construction. Working in 3D provides a number of important benefits over more traditional 2D VPLs. For example, working in three dimensions allows the system to display more information in an environment which is easier to interact with than a 2D representation which uses the same screen size (2). In the 3D display, the programmer is free to move his or her viewpoint anywhere inside the virtual world in order to look at any particular section of a program from any viewpoint. This sort of flexibility is not available in most 2D VPLs.

Fig. 1.6 shows the main components of a Cube program as they appear in a recursive function to compute the factorial of a given number (12). Cube programs are composed primarily of holder cubes, predicate cubes, definition cubes, ports, pipes, and planes. The entire structure in Fig. 1.6 is surrounded by a definition cube which associates the icon “!” with the function defined within the cube. The definition cube has two ports connected to it, one on the left and on the right. The left-hand port serves as the input while the right-hand port provides output, although ports in Cube are bi-directional, so technically either port can serve either function. Both ports are connected through pipes to holder cubes in the bottom plane of the diagram which represents the base case of the recursion. Note that each plane represents a dataflow diagram. In the case of the bottom plane, the diagram simply supplies default values for the ports and indicates what type of values each port can accept or produce. If the
value at the input port is 0, then the bottom plane is active and the value from the right-hand holder cube, i.e. one, flows to the output port. If the input is greater than zero, the greater than predicate in the top plane is satisfied, and one is subtracted from the input by the bottom branch of the upper dataflow diagram. This difference is fed into the recursive call to the factorial function, and the result is multiplied by the original input. The product then flows to the output port. After defining the factorial function within a program, the programmer can then call it by simply connecting a predicate cube labeled by the “!” icon to holder cubes at the two ports.

1.8 Visual Programming and Abstraction

One of the challenges in visual programming research is scaling up to the support of ever-larger programs. This is a greater issue for VPLs than for traditional textual languages (although it certainly can be said to exist in both) for reasons relating to representation, language design and implementation, and relative youth of the area. For example, some of the visual mechanisms used to achieve characteristics such as explicitness can occupy a great deal of space, making it harder to maintain context. Also, it is hard to apply in a straightforward way techniques developed for traditional languages, because doing so often results in a reintroduction of the very complexities VPLs have tried to remove or simplify.

Recent developments in the area of abstraction have been particularly important to the scalability of VPLs. The two most widely-supported types of abstraction, both in visual and textual languages, are procedural abstraction and data abstraction. In particular, procedural abstraction has shown itself to be
supportable by a variety of VPLs. A key attribute to supporting procedural abstraction in a VPL has been consistency with the rest of programming in the same VPL. Representative solutions include allowing the programmer to select, name, and iconify a section of a dataflow graph (Fig. 1.4), which adds a node representing the subgraph to a library of function nodes in a dataflow language; setting up separate spreadsheets (Fig. 1.3), which can be automatically generalized to allow user-defined “functions” in a form-based language; and recording and generalizing a sequence of direct manipulations (Fig. 1.2) in a by-demonstration language.

Data abstraction has been slower in coming to VPLs, largely because it is sometimes difficult to find a way to maintain characteristics such as concreteness or feedback, while adding support for ideas central to data abstraction such as generality and information hiding. Still, support for data abstraction has emerged for a number of VPLs. For example, in Forms/3, a new data type is defined via a spreadsheet, with ordinary cells defining operations or methods, and with two distinguished cells that allow composition of complex objects from simpler ones and definition of how an object should appear on the screen. In Cocoa, each character’s appearance is painted using a graphical editor, and each demonstration of a new rule “belongs” to the character type being manipulated, providing roughly the functionality of an operation or method. Both Forms/3 and Cocoa also support limited forms of inheritance.

1.9 Conclusions on Visual Programming

The field of visual programming languages abounds with examples of unique efforts to widen the accessibility and enhance the power of computer programming. Although the various projects discussed above vary in a number of details, particularly the visual metaphor employed and the targeted application domain, they all share the common goal of improving the programming process. In addition, recent research into solidifying the theoretical foundations of visual programming and serious efforts to develop standardized formal classifications for VPLs indicate that the field has begun to reassess itself and mature. Even as the area has grown over the past twenty years, important historical contributions from work such as Sketchpad and Pygmalion have maintained their influence on various VPL designs.

Despite the move toward graphical displays and interactions embodied by VPLs, a survey of the field quickly shows that it is not worthwhile to eschew text entirely. While many VPLs could represent all aspects of a program visually, such programs are generally harder to read and work with than those that use text for labels and some atomic operations. For example, although an operation like addition can be represented graphically in VPL, doing so can results in a rather dense, cluttered display. On the other hand, using text to represent such an atomic operation produces a less complicated display without losing the overall visual metaphor.

As computer graphics hardware and processors continue to improve in performance and drop in price, three dimensional VPLs like Cube should begin to garner more attention from the research community. 3D VPLs not only address the problem of fitting large amounts of information on a rather small screen,
but they also exemplify the inherent synergy between programming languages, computer graphics, and human-computer interfaces which has been a hallmark of visual programming from its inception.
2 Modeling with Petri Nets (13)

2.1 Introduction

The growth in the complexity of modern industrial systems, such as production, process control, communication systems, etc., creates numerous problems for their developers. In the planning stage, one is confronted with increased capabilities of these systems due to the unique combination of hardware and software, which operate under a large number of constraints arising from the limited system resources. In view of the capital intensive and complex nature of modern industrial systems, the design and operation of these systems require modeling and analysis in order to select the optimal design alternative, and operational policy. It is well-known that flaws in the modeling process can substantially contribute to the development time and cost. The operational efficiency may be affected as well. Therefore special attention should be paid to the correctness of the models that are used at all planning levels.

Petri nets, as graphical and mathematical tools, provide a uniform environment for modeling, formal analysis, and design of discrete event systems. One of the major advantages of using Petri net models is that the same model is used for the analysis of behavioral properties and performance evaluation, as well as for systematic construction of discreteevent simulators and controllers. Petri nets were named after Carl A. Petri who created in 1962 a net-like mathematical tool for the study of communication with automata. Their further development was facilitated by the fact that Petri nets can be used to model properties such as process synchronization, asynchronous events, concurrent operations, and conflicts or resource sharing. These properties characterize discrete-event systems whose examples include industrial automated systems, communication systems, and computer-based systems. These, and other factors discussed in this paper, make Petri nets a promising tool and technology for application to Industrial Automation.

Petri nets as graphical tools provide a powerful communication medium between the user, typically requirements engineer, and the customer. Complex requirements specifications, instead of using ambiguous textual descriptions or mathematical notations difficult to understand by the customer, can be represented graphically using Petri nets. This combined with the existence of computer tools allowing for interactive graphical simulation of Petri nets, puts in hands of the development engineers a powerful tool assisting in the development process of complex systems.

As a mathematical tool, a Petri net model can be described by a set of linear algebraic equations, or other mathematical models reflecting the behavior of the system. This opens a possibility for the formal analysis of the model. This allows one to perform a formal check of the properties related to the behavior of the underlying system, e.g., precedence relations amongst events, concurrent operations, appropriate synchronization, freedom from deadlock, repetitive activities, and mutual exclusion of shared resources, to mention some. The simulation based model validation can only produce a limited set of states of the modeled system, and thus can only show presence (but not absence) of errors in the model, and its underlying requirements specification. The ability of Petri nets to verify the model
formally is especially important for realtime safety-critical systems such as air-traffic control systems, rail-traffic control systems, nuclear reactor control systems, etc. Petri nets were used to model real-time fault tolerant and safety-critical system or fault detection and in-process monitoring.

One of the most successful application areas of Petri nets has been modeling and analysis of communication protocols. The work in this area can be dated back to the early 1970s. In the past few years, a number of approaches have been proposed which allow for the construction of Petri net models of protocols from specifications written in a relatively skill-free languages.

Petri nets have been used extensively to model and analyze manufacturing systems. In this area, Petri nets were used to represent simple production lines with buffers, machine shops, automotive production systems, flexible manufacturing systems, automated assembly lines, resource-sharing systems, and recently just-in-time and kanban manufacturing systems.

The application of Petri nets to modeling sequence controllers is another success story. Programmable Logic Controllers are commonly used for the sequence control in automated systems. They are designed using ladder logic diagrams, which are known to be very difficult to debug and modify. Petri net based sequence controllers, on the other hand, are easy to design, implement, and maintain. In the early 80's, Hitachi Ltd, developed a Petri net based sequence controller which was successfully used in real applications to control parts assembly system, and automatic warehouse load/unload system. The use of Petri nets, as reported, substantially reduced the development time compared with the traditional approach.

Petri nets have been extensively used in software development. The work in this area focused on modeling and analysis of software systems using Petri nets. The most mature developments involve the use of colored Petri nets. Colored Petri nets have been demonstrated to be a useful language for the design, specification, simulation, validation and implementation of large software systems.

Petri nets, as a mathematical tool, allow for the performance evaluation of the modeled systems. Both deterministic and stochastic performance measures can be evaluated by using a broad class of Petri net models incorporating in their definitions deterministic and/or probabilistic time functions. The performance evaluation can be conducted using either analytical techniques, based on solving the underlying (semi)Markov processes, or discrete event simulation. The use of models which incorporate time functions having probabilistic distributions allows one to obtain production rates for manufacturing system models, throughput, delays, capacity for communication and microprocessor system models, as well as critical resource utilization and reliability measures for these and other systems. In recent years, this class of Petri net models has been extensively used to model and study analytically performance of multiprocessor systems, multiprocessor system buses, DSP communication channels, parallel computer architectures, as well as parallel distributed algorithms.

Another area of applications was communication networks. Work was conducted on Fiber Optics Local Area Networks such as Expressnet, Fastnet, D-Net, U-Net, Token Ring. Fieldbuses, such as FIP and ISA-SP50, have attracted lots of attention in the last two years. This is not surprising, since they are very
important networks for factory automation. The interest steadily grows in modeling and evaluation of High Speed Networks, crucial for the successful development of Multimedia Systems.

Petri nets with time extensions, combined with heuristic search techniques, were used to model and study scheduling problems involving manufacturing systems, as well as robotic systems. Petri nets with time extensions were also used to model and analyze dynamics of continuous chemical processes.

2.2 Description of Petri Nets

A Petri net may be identified as a particular kind of bipartite directed graph populated by three types of objects. These objects are places, transitions, and directed arcs connecting places to transitions and transitions to places. Pictorially, places are depicted by circles and transitions as bars or boxes. A place is an input place to a transition if there exists a directed arc connecting this place to the transition. A place is an output place of a transition if there exists a directed arc connecting the transition to the place. In its simplest form, a Petri net may be represented by a transition together with its input and output places. This elementary net may be used to represent various aspects of the modeled systems. For instance, input (output) places may represent preconditions (postconditions), the transition an event. Input places may represent the availability of resources, the transition their utilization, output places the release of the resources. An example of a Petri net is shown in Fig. 1.7. This net consists of five places, represented by circles, four transitions, depicted by bars, and directed arcs connecting places to transitions and transitions to places. In this net, place \( p_1 \) is an input place of transition \( t_1 \). Places \( p_2 \) and \( p_3 \) are output places of transition \( t_1 \).

![Fig. 1.7 Example of graphical representation of a Petri net](image)

In order to study dynamic behavior of the modeled system, in terms of its states and their changes, each place may potentially hold either none or a positive number of tokens, pictured by small solid dots, as shown in Fig. 1.7. The presence or absence of a token in a place can indicate whether a condition
associated with this place is true or false, for instance. For a place representing the availability of resources, the number of tokens in this place indicates the number of available resources. At any given time instance, the distribution of tokens on places, called Petri net marking, defines the current state of the modeled system. A marking of a Petri net with \( m \) places is represented by an \((m \times 1)\) vector \( M \), elements of which, denoted as \( M(p) \), are nonnegative integers representing the number of tokens in the corresponding places. A Petri net containing tokens is called a marked Petri net. For example, in the Petri net model shown in Fig. 1.7, \( M = (1,0,0,0,0) \).

Formally, a Petri net can be defined as follows:

\[
\text{PN} = (P, T, I, 0, M_0);
\]

where

1) \( P = \{ p_1, p_2, \ldots, p_m \} \) is a finite set of places,
2) \( T = \{ t_1, t_2, \ldots, t_m \} \) is a finite set of transitions, \( P \cup T \neq \emptyset \), and \( P \cap T \neq 0 \),
3) \( I: (P \times T) \rightarrow N \) is an input function that defines directed arcs from places to transitions, where \( N \) is a set of nonnegative integers,
4) \( O: (P \times T) \rightarrow N \) is an output function which defines directed arcs from transitions to places, and
5) \( M_0: P \rightarrow N \) is the initial marking.

If \( I(p, t) = k \) (\( O(p, t) = k \)), then there exist \( k \) directed (parallel) arcs connecting place \( p \) to transition \( t \) (transition \( t \) to place \( p \)). If \( I(p, t) = 0 \) (\( O(p, t) = 0 \)), then there exist no directed arcs connecting \( p \) to \( t \) (\( t \) to \( p \)). Frequently, in the graphical representation, parallel arcs connecting a place (transition) to a transition (place) are represented by a single directed arc labeled with its multiplicity, or weight \( k \). This compact representation of multiple arcs is shown in Fig. 1.8.

By changing distribution of tokens on places, which may reflect the occurrence of events or execution of operations, for instance, one can study dynamic behavior of the modeled system. The following rules are used to govern the flow of tokens.

**Enabling Rule.** A transition \( t \) is said to be enabled if each input place \( p \) of \( I \) contains at least the number of tokens equal to the weight of the directed arc connecting \( p \) to \( t \).

**Firing Rule:**
(a) An enabled transition $t$ may or may not fire depending on the additional interpretation, and
(b) A firing of an enabled transition $t$ removes from each input place $p$ the number of tokens equal to the weight of the directed arc connecting $p$ to $t$. It also deposits in each output place $p$ the number of tokens equal to the weight of the directed arc connecting $t$ to $p$.

**Fig. 1.9 (a) Transition $t_1$ enabled. (b) Enabled transition $t_1$ fires**

The enabling and firing rules are illustrated in Fig. 1.9. In Fig. 1.9 (a), transition $t_1$ is enabled as the input place $p_1$ of transition $t_1$ contains two tokens, and $I(p_1, t_1) = 2$. The firing of the enabled transition $t_1$ removes from the input place $p_1$ two tokens as $I(p_1, t_1) = 2$, and deposits one token in the output place $p_3$, $O(p_3, t_1) = 1$, and two tokens in the output place $p_2$, $O(p_2, t_1) = 2$. This is shown in Fig. 1.9 (b).

**Fig. 1.10 Petri net with an inhibitor arc**

The modeling power of Petri nets can be increased by adding the zero testing ability, i.e., the ability to test whether a place has no token. This is achieved by introducing an inhibitor arc, The inhibitor arc connects an input place to a transition, and is pictorially represented by an arc terminated with a small circle. A Petri net with an inhibitor arc is shown in Fig. 1.10. The presence of an inhibitor arc connecting an input place to a transition changes the transition enabling conditions. In the presence of the inhibitor arc, a transition is regarded as enabled if each input place, connected to the transition by a normal arc (an arc terminated with an arrow), contains at least the number of tokens equal to the weight of the arc, and no tokens are present on each input place connected to the transition by the inhibitor arc. The transition firing rules are the same as for normally connected places. The firing, however, does not change the marking in the inhibitor arc connected places.

**Fig. 1.11 Self-loop removal**
A Petri net is said to be pure or self-loop free if no place is an input place to and output place of the same transition. A Petri net that contains self-loops can always be converted to a pure Petri net as shown in Fig. 1.11.

In order to illustrate how Petri nets can be used to model properties such as concurrent activities, synchronization, mutual exclusion etc., let’s consider a simple example of a multirobot system. This system is represented by a Petri net model shown in Fig. 1.12, and Table I. In this model, two robot arms perform pick-and-place operations accessing a common workspace at times to obtain or transfer parts. In order to avoid collision, it is assumed that only one robot can access the workspace at a time. In addition, it is assumed that the common workspace contains a buffer with a limited space for products. This could represent an operation of two robot arms servicing two different machining tools, with one robot arm transferring semiproducts from one machining tool to the buffer, and the other robot arm transferring semiproducts from the buffer to the other machining tool.

![Fig. 1.12 Petri net model of a multirobot system](image)

**TABLE I**

<table>
<thead>
<tr>
<th>Place (with token)</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1(p1)</td>
<td>Robot R1 (R2) performs tasks outside the common workspace</td>
</tr>
<tr>
<td>p2(p2)</td>
<td>Robot R1 (R2) waits for the access to the common workspace</td>
</tr>
<tr>
<td>p3(p0)</td>
<td>Robot R1 (R2) performs in the common workspace</td>
</tr>
<tr>
<td>p7</td>
<td>mutual exclusion</td>
</tr>
<tr>
<td>p8</td>
<td>number of empty (full) positions in buffer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1(t4)</td>
<td>Robot R1 (R2) requests access to the common workspace</td>
</tr>
<tr>
<td>t2(t5)</td>
<td>Robot R1 (R2) enters the common workspace</td>
</tr>
<tr>
<td>t3(t6)</td>
<td>Robot R1 (R2) leaves the common workspace</td>
</tr>
</tbody>
</table>

In this model, places p1, p2, p3 and transitions t1, t2, t3 model activities of robot arm R1. Places p4, p5, p6 and transitions t4, t5, t6 model activities of robot arm R2. Transitions t1 and t4 represent concurrent
activities of $R1$ and $R2$. Either of these transitions can fire before or after, or in parallel with the other one. The access to the common workspace requires synchronization of the activities of the arms in order to avoid collision. Only one robot arm can access the common workspace at a time. This synchronization is accomplished by the mutual exclusion mechanism implemented by a subnet involving places $p_7$, $p_9$, $p_6$ and transitions, $t_2$, $t_3$, $t_5$, $t_6$. Firing transition $t_2$ disables $t_6$, assuming $t_5$ is enabled, and vice versa. Thus only one robot arm can access the common workspace at a time. In addition, it is assumed that the buffer space is “b”. Thus, for instance, if $p_8$ is empty, then $t_5$ cannot be enabled. This prevents $R2$ from attempting to transfer to the buffer a part when there is no space in the buffer. Also, $R2$ cannot access the buffer if there is no part in the buffer, or place $p_9$ is empty.

2.3 Properties of Petri Nets

Petri nets as mathematical tools possess a number of properties. These properties, when interpreted in the context of the modeled system, allow the system designer to identify the presence or absence of the application domain specific functional properties of the system under design. Two types of properties can be distinguished: behavioral and structural properties. The behavioral properties are those which depend on the initial state, or marking, of a Petri net. The structural properties, on the other hand, do not depend on the initial marking of a Petri net. These properties depend on the topology, or net structure, of a Petri net. In this section, it is provided an overview of some of the most important, from practical point of view, behavioral properties. The behavioral properties discussed in this section are reachability, boundedness, conservativeness, liveness, reversibility and home state.

2.3.1 Reachability

An important issue in designing distributed systems is whether a system can reach a specific state, or exhibit a particular functional behavior. In general, the question is whether the system modeled with Petri nets exhibits all desirable properties, as specified in the requirements specification, and no undesirable ones.

In order to find out whether the modeled system can reach a specific state as a result of a required functional behavior, it is necessary to find such a sequence of firings of transitions which would result in transforming a marking $M_0$ to $M_f$, where $M_f$ represents the specific state, and the sequence of firings represents the required functional behavior. It should be noted that real systems may reach a given state as a result of exhibiting different permissible patterns of functional behavior. In a Petri net model, this should be reflected in the existence of specific sequences of transitions firings, representing the required functional behavior, which would transform a marking $M_0$ to the required marking $M_f$. The existence in the Petri net model of additional sequences of transitions firings which transform $M_0$ to $M_f$ indicates that the Petri net model may not be reflecting exactly the structure and dynamics of the underlying system. This may also indicate the presence of unanticipated facets of the functional behavior of the real system, provided that the Petri net model accurately reflects the underlying system requirements specification. A marking $M_f$ is said to be reachable from a marking $M_0$ if there exists a sequence of transitions firings which transforms a marking $M_0$ to $M_f$. A marking $M_f$ is said to be
immediately reachable from a marking $M_0$ if a firing of an enabled transition in $M_0$ results in marking $M_1$. For instance, in the Petri net model of the multirobot assembly system shown in Fig. 1.12, the state in which robot arm $R1$ performs tasks in the common workspace, with robot arm $R2$ waiting outside, is represented by the marking vector $M_1 = \{0, 0, 1, 0, 1, 0, 2, 1\}^T$. $M_1$ can be reached from the initial marking $M_0$ where $M_0 = \{1, D, D, L D, 0, L 3, \text{of}, \}$ by the following sequence of transitions firings-$t_{12}t_{13}$. The marking $M_0 = \{1, 0, 0, 1, 0, 1, 3, 0\}^T$, which represents the state of the system in which robot arm $R1$ waits for the access to the common workspace and robot arm $R2$ performs tasks outside the common workspace, is immediately reachable from the initial marking $M_0$ when transition $t_1$ fires. It should be noted that in $M_0$ transitions $t_j$ and $t_k$ are both enabled. The set of all possible markings reachable from $M_0$ is called the reachability set, and denoted by $R(M_0)$. The set of all possible firing sequences from $M_0$ is denoted by $L(M_0)$. Thus the problem of identifying the existence of a specific state $M_k$, the system can take on, can be redefined as the problem of finding $M_k \in R(M_0)$.

### 2.3.2 Boundedness and Safeness

Places are frequently used to represent information storage areas in communication and computer systems, product and tool storage areas in manufacturing systems, etc. It is important to be able to determine whether proposed control strategies prevent from the overflows of these storage areas. The information storage areas can hold, without corruption, only a restricted number of pieces of data. In manufacturing systems, attempts to store more tools, for instance, in the tool storage area may result in the equipment damage. The Petri net property which helps to identify in the modeled system the existence of overflows is the concept of boundedness. A Petri net is said to be $k$-bounded if the number of tokens in any place $p$, where $p \in P$, is always less or equal to $k$ ($k$ is a nonnegative integer number) for every marking $M$ reachable from the initial marking $M_0$, $M \in R(M_0)$. A Petri net is safe if it is 1-bounded.

A Petri net shown in Fig. 1.13 is safe. In this net no place can contain more then one token. An example of a Petri net which is unbounded is shown in Fig. 1.14. This net is unbounded because place $p_4$ can hold an arbitrarily large number of tokens.

**Fig. 1.13 Petri net that is safe**

**Fig. 1.14 Petri net that is unbounded**

### 2.3.3 Conservativeness

In real systems, the number of resources in use is typically restricted by the financial as well as other constraints. If tokens are used to represent resources, the number of which in a system is typically fixed, then the number of tokens in a Petri net model of this system should remain unchanged irrespective of
the marking the net takes on. This follows from the fact that resources are neither created nor destroyed, unless there is a provision for this to happen. For instance, a broken tool may be removed from the manufacturing cell, thus reducing the number of tools available by one.

A Petri net is conservative if the number of tokens is conserved. From the net structural point of view, this can only happen if the number of input arcs to each transition is equal to the number of output arcs. However, in real systems resources are frequently combined together so that certain tasks can be executed, then separated after the task is completed. For instance, in a flexible manufacturing system an automatic guided vehicle collects a pallet carrying products from a machining cell, and subsequently delivers it to the unloading station where the vehicle and pallet are separated. This scenario is illustrated in Fig. 9. Transition \( t_1 \) models loading a pallet onto a vehicle; transition \( t_2 \) represents the pallet being delivered to the unload station and subsequently removed from the vehicle. Although the number of tokens in the net changes from two to one when \( t_1 \) fires, and then back to two tokens when \( t_2 \) fires, the number of resources in the system does not change. In order to overcome this problem, weights may be associated with places allowing for the weighted sum of tokens in a net to be constant. A Petri net is said to be conservative if there exists a vector \( w = (w_1, w_2, ..., w_m) \), where \( m \) is the number of places, and \( w(p) > 0 \) for each \( p \in P \), such that the weighted sum of tokens remains the same for each marking \( M \) reachable from the initial marking \( M_0 \). A Petri net is said to be strictly conservative if all entries of vector \( w \) are unity. The Petri net shown in Fig. 1.15 is conservative with respect to vector \( w = [1,1,2,1,1]\) as the weighted sum of tokens in each marking is two. An example of a Petri net which is not conservative is shown in Fig. 1.14; place \( p_4 \) can hold an arbitrarily large number of tokens. If a Petri net is conservative with respect to a vector with all elements equal to one, then the net is said to be strictly conservative. An example of a Petri net which is strictly conservative is shown in Fig. 1.16.

2.3.4 Liveness

The concept of liveness is closely related to the deadlock situation, which has been studied extensively in the context of operating systems. It has been showed that four conditions must hold for a deadlock to occur. These four conditions are:

1) Mutual exclusion: a resource is either available or allocated to a process which has an exclusive access to this resource.
2) Hold and wait: a process is allowed to hold a resource(s) while requesting more resources.
3) No preemption: a resource(s) allocated to a process cannot be removed from the process, until it is released by the process itself.
4) Circular wait: two or more processes are arranged in a chain in which each process waits for resources held by the process next in the chain.

![Petri net with different levels of liveness of transitions](image)

For instance, in a flexible manufacturing system, a deadlock occurs when the input/output buffer of a machining tool holds a pallet with already machined products, and another pallet with products to be machined has been delivered to the buffer. Assuming that the buffer can hold one pallet only at a time, and an automated guided vehicle (AGV), for instance, has a space for one pallet, a deadlock occurs. The pallet with machined parts cannot be moved from the buffer to the AGV. The pallet with parts to be machined cannot be moved from the AGV to the buffer. In this example, all four conditions hold, with the buffer and AGV space for pallets regarded as resources. Unless there is a provision in the control software for deadlock detection and recovery, a deadlock situation, initially confined to a small subsystem, may propagate to affect a large portion of a system. This frequently results in a complete standstill of a system. A Petri net modeling a deadlock free system must be live. This implies that for all markings $M$, which are reachable from the initial marking $M_0$, it is ultimately possible to fire any transition in the net by progressing through some firing sequence. The Petri net shown in Fig. 1.16 is live. This requirement, however, might be too strict to represent some real systems or scenarios which exhibit deadlock-free behavior. For instance, the initialization of a system can be modeled by a transition (or transitions) which fires a finite number of times. After initialization, the system may exhibit a deadlock free behavior, although the Petri net representing this system is no longer live as specified above. For this reason, different levels of liveness for transition $t$, and marking $M_0$ were introduced. A transition $t$ in a Petri net is said to be:

- **L0-live** (or dead) if there is no firing sequence in $L(M_0)$ in which $t$ can fire,
- **L1-live** (potentially firable) if $T$ can be fired at least once in some firing sequence in $L(M_0)$,
- **L2-live** if $t$ can be fired at least $k$ times in some firing sequence in $L(M_0)$ given any positive integer $k$,
- **L3-live** if $t$ can be fired infinitely often in some firing sequence in $L(M_0)$, and
- **L4-live** (or live) if $t$ is L1-live (potentially firable) in every marking in $R(M_0)$.
Following this classification, a Petri net is said to be \( Li \)-live, for marking \( M_0 \), if every transition in the net is \( Li \)-live. Different levels of liveness of transitions are shown in Fig. 1.17. In this example, transitions \( t_0 \), \( t_1 \), \( t_2 \) and \( t_3 \) are \( L0 \), \( L1 \), \( L2 \), and \( L3 \)-live, respectively, and strictly.

### 2.3.5 Reversibility and Home State

An important issue in the operation of real systems, such as manufacturing systems, process control systems, etc., is the ability of these systems for an error recovery. These systems are required to return from the failure states to the preceding correct states. This requirement is closely related to the reversibility and home state properties of a Petri net. A Petri net, for the initial marking \( M_0 \), is said to be reversible if for each marking \( M \) in \( R(M_0) \), \( M \) is reachable from \( M_0 \). The home state property is less restrictive, and more practical, then the reversibility property of a Petri net. A Petri net state \( M \) is said to be a home state if for each marking \( M \) in \( R(M) \), \( M \) is reachable from \( M \). The Petri net shown in Fig. 1.13 is reversible. The Petri net shown in Fig. 1.14 is nonreversible.

### 2.4 Analysis Methods

In the previous section, a number of properties of Petri nets has been defined which are useful for analyzing modeled systems. An important issue to be considered during analysis is whether there exists one-to-one functional correspondence between the Petri net model and the original requirements specification; typically expressed in an informal way. The construction of Petri net models from informal requirements specifications is not a trivial task, which requires a great deal of modeling experience, as well as the knowledge of the techniques assisting in the model construction. As a result, a Petri net model may differ considerably from its original specification. This is especially true when large Petri net models of complex systems are involved. The existence of the one-to-one functional correspondence between an original requirements specification and its Petri net representation allows projection of the analysis results, obtained for the Petri net model, onto the original description. This provides feedback to the customers which can, in many instances, help the customers clarify their perception of the system. Another important issue to be addressed during the analysis stage is the completeness of the requirements specification. In most cases, the requirements specification defines the external functional behavior of a system. This is typically expressed in terms of the system input output relationships. Inputs are generated by the environment of the system. Outputs are responses of the system to these inputs. If some inputs, generated by the environment of the system, are not included in the requirements specification, then the system will be unable to respond to these inputs properly when they occur during the system normal operation. The completeness of the requirements is especially important in the case of safety-critical systems. In these systems, the incompleteness of the requirements specification may lead to catastrophic events to occur in the environment of the system. For instance, the occurrence of unanticipated states in the operation of a nuclear reactor may result in the failure of the control system to respond to them properly, or at all, thus potentially leading to the reactor system failure. The consistency of the requirement specification is another issue to be considered during analysis. The inconsistency occurs when for a given permissible, temporal combination of inputs, a requirements specification allows for two or more different permissible temporal combinations of
outputs. It is mainly due to a vague, incomplete, and frequently incorrect perception of the system functionality. In this chapter an overview of two fundamental methods of analysis is presented. One is based on the reachability tree, and the other on the matrix equation representation of a net. In addition to the two methods, a number of techniques were proposed to assist in the analysis of Petri net models. These approaches allow for a systematic transformation of a Petri net, by reducing the number of places and transitions in a net, and at the same time preserving the properties such as boundedness, conservativeness, liveness, etc. Smaller nets are easier to analyze.

2.4.1 The Coverability Tree
This approach is based on the enumeration of all possible markings reachable from the initial marking $M_0$. Starting with an initial marking $M_0$, one can construct the reachability set by firing all possible transitions enabled in all possible markings reachable from the initial marking $M_0$. In the reachability tree, each node is labeled with a marking; arcs are labeled with transitions. The root node of the tree is labeled with an initial marking $M_0$. The reachability set becomes unbounded for either of two reasons: The existence of duplicate markings, and a net is unbounded. In order to prevent the reachability tree from growing indefinitely large, two steps need to be taken when a tree is constructed. The first step involves eliminating duplicate markings. If on the path from the initial marking $M_0$ to a current marking $M$ there is a marking $M'$, which is identical to the marking $M$, then the marking $M$, as a duplicate marking, becomes a terminal node. The occurrence of a duplicate marking implies that all possible markings reachable from $M$ have been already added to the tree. For unbounded nets, in order to keep the tree finite, the symbol $\omega$ is introduced. The symbol $\omega$ can be thought of as the infinity. Thus, for any integer $n$, $\omega + n = \omega$, $\omega - n = \omega$, $n < \omega$. In this case, if on the path from the initial marking $M_0$ to a current marking $M$ there is a marking $M'$, with its entries less or equal to the corresponding entries in the marking $M$, then the entries of $M$, which are strictly greater than the corresponding entries of $M'$, should be replaced by the symbol $\omega$. In some paths the existence of markings with the corresponding entries equal or increasing (as moving away from the root node) indicates that the firing sequence which transforms $M'$ to $M$ can be repeated indefinitely. Each time this sequence is repeated, the number of tokens on places labeled by the symbol $\omega$ increases. The coverability tree is constructed according to the following algorithm:

1) Let the initial marking $M_0$ be the root of the tree and tag it "new."
2) While "new" markings exist do the following:
3) Select a "new" marking $M$
   a. If $M$ is identical to another marking in the tree, then tag $M$ "old," and go to another "new" marking.
   b. If no transitions are enabled in $M$, tag $M$ "terminal."
4) For every transition $t$ enabled in marking $M$ do the following:
   a. Obtain the marking $M'$ which results from firing $t$ in $M$.
   b. If on the path from the root to $M$ there exists a marking $M''$ such that $M'(p) \geq M''(p)$ for each place $p$, and $M' = M''$, then replace $M'(p)$ by $\omega$ for each $p$, wherever $M'(p) > M''(p)$.
   c. Introduce $M'$ as a node, draw an arc from $M$ to $M'$ labeled $t$, and tag $M'$ "new."
The following example will illustrate the approach. Consider the net shown in Fig. 1.18, and its coverability tree in Fig. 1.19. For the given initial marking, the root node is $M_0 = (1,0,1,0)^T$. In this marking, transition $t_3$ is enabled.

When $t_3$ fires a new marking is obtained: $M_1 = (1,0,0,1)^T$. This is a "new" marking in which transition $t_2$ is enabled. Firing $t_2$ in $M_1$ results in $M_2 = (1,1,1,0)^T$. Since $M_2 = (1,1,1,0)^T \geq M_0 = (1,0,1,0)^T$, the second component should be replaced by the symbol $\omega$. This reflects the fact that the firing sequence $t_3t_2$ may be repeated arbitrarily large number of times. In marking $M_2 = (1,\omega,1,0)^T$ two transitions are enabled: transition $t_3$, and transition $t_1$. Firing $t_1$ results in marking $M_3 = (l,\omega,0,0)^T$, which is a "terminal" node. Firing $t_2$ results in a "new" marking $M_4 = (l,\omega,0,1)^T$, which enables transition $t_3$. Firing $t_2$ in $M_4$ results in an "old" node: $M_5 = (1,\omega,1,0)^T$ which is identical to $M_2$.

A number of properties can be studied by using the coverability tree. For instance, if any node in the tree contains the symbol $\omega$, then the net is unbounded since the symbol $\omega$ can become arbitrarily large. Otherwise, the net is bounded. If each node of the tree contains only zeros and ones, then the net is safe. A transition is dead if it does not appear as an arc label in the tree. If a marking $M$ is reachable from a marking $M_0$, then there exists a node $M'$, such that $M \leq M'$. However, since the symbol $\omega$ can become arbitrarily large, certain problems, such as coverability and liveness, cannot be solved by studying the coverability tree only. For a bounded Petri net, the coverability tree contains, as nodes, all possible markings reachable from the initial marking $M_0$. In this case, the coverability tree is called the reachability tree. For a reachability tree any analysis question can be solved by inspection.

2.4.2 The Incidence Matrix and State Equation
An alternative approach to the representation and analysis of Petri nets is based on matrix equations. In this approach matrix equations are used to represent dynamic behavior of Petri nets. The fundamental to this approach is the incidence matrix which defines all possible interconnections between places and transitions in a Petri net. The incidence matrix of a pure Petri net is an integer $n \times m$ matrix $A$, where $n$ is
the number of transitions, and \( m \) is the number of places. The entries of the incidence matrix are defined as follows: \( a_{ij} = a^+_{ij} − a^-_{ij} \), where \( a^+_{ij} \) is equal to the number of arcs connecting transition \( t_i \) to its output place \( p_j \) \( (a^+_{ij} = O(p_j, t_i)) \), and \( a^-_{ij} \) is equal to the number of arcs connecting transition \( t_i \) to its input place \( p_j \) \( (a^-_{ij} = I(p_j, t_i)) \). When transition \( t_i \) fires, \( a^+_{ij} \) represents the number of tokens deposited on its output place \( p_j \); \( a^-_{ij} \) represents the number of tokens removed from its input place \( p_j \). \( a_{ij} \) represents the change in the number of tokens in place \( p_j \). Therefore, transition \( t_i \) is said to be enabled in marking \( M \) if

\[
a^-_{ij} \leq M(p_j), \quad i = 1, 2, \ldots, m
\]

For Petri nets with self-loops. \( a_{ij} = 0 \) for a place \( p_j \) and transition \( t_i \) which belong to a self-loop. For this reason, in order to make sure that the incidence matrix properly reflects the structure of a Petri net, the net is assumed to be pure or made pure by introducing two additional places (see Fig. 1.11). The state equation for a Petri net represents a change in the distribution of tokens on places (marking) as a result of a transition firing. This equation is defined as follows:

\[
M_k = M_{k-1} + A^T u_k, \quad k = 1, 2, \ldots
\]

\( M_k \) is an \( m \times 1 \) column vector representing a marking \( M_k \) immediately reachable from a marking \( M_{k-1} \) after firing transition \( t_i \). The \( k \)-th firing vector \( u_k \), an \( n \times 1 \) column vector, has only one nonzero entry. This entry, a 1 in the \( i \)-th position, represents a transition \( t_i \) firing in the \( k \)-th firing of the net firing sequence starting with an initial marking \( M_0 \). This entry corresponds to the \( i \)-th row of the incidence matrix \( A \) which represents a change of a marking as a result of a firing transition \( t_i \). The matrix equation is useful in studying the reachability problem.

Two concepts related to the incidence matrix are especially useful in studying properties of Petri net models. They are \( T \)-invariant and \( P \)-invariant.

An integer solution \( x \) of \( A^T x = 0 \) is called a \( T \)-invariant. The nonzero entries in a \( T \)-invariant represent the firing counts of the corresponding transitions which belong to a firing sequence transforming a marking \( M_0 \) back to \( M_0 \). Although a \( T \)-invariant states the transitions comprising the firing sequence transforming a marking \( M_0 \) into \( M_0 \), and the number of times these transitions appear in this sequence, it does not specify the order of transitions firings.

An integer solution \( y \) of \( Ay = 0 \) is called a \( P \)-invariant. The \( P \)-invariants can be explained intuitively in the following way. The nonzero entries in a \( P \)-invariant represent weights associated with the corresponding places so that the weighted sum of tokens on these places is constant for all markings reachable from an initial marking.

The subset of places (transitions) corresponding to the nonzero entries of a \( T \)-invariant (\( P \)-invariant) is called the support of an invariant, and denoted by \( \|x\| (\|y\|) \). A support is said to be minimal if no proper nonempty subset of the support is also a support.
2.4.3 An example

In this section, it will be demonstrated how the coverability tree and invariant based techniques can be used to analyze the Petri net model of the multirobot system which is shown in Fig. 1.12. Without losing the discussion generality, assume \( b = 1 \). The coverability tree, in this case a reachability tree, is shown in Fig. 1.20. The incidence matrix of this net is shown in Fig. 1.21.

The P-invariants obtained for this net are as follows:

\[
\begin{align*}
y_1 &= (1 1 1 0 0 0 0 0 0)^T \\
y_2 &= (0 0 0 1 1 1 0 0 0)^T \\
y_3 &= (0 0 1 0 0 1 1 0 0)^T \\
y_4 &= (0 0 0 0 0 1 1)^T
\end{align*}
\]

The following are the corresponding invariant supports:

\[ \|y_1\| = \{p_1, p_2, p_3\} \]
\|y_2\| = \{p_4, p_5, p_6\}
\|y_3\| = \{p_3, p_6, p_7\}
\|y_4\| = \{p_8, p_9\}

**Roundedness and Safeness:** The Petri net shown in Fig. 1.12 is bounded. This is evident from the reachability tree; no marking reachable from the initial marking $M_0$ contains the $\omega$ symbol. In addition, since for each marking, no entry is greater than one, the net is safe. These properties can be also easily established using $P$-invariants. Since each place in the net belongs to some invariant support, and the net starts from a bounded initial marking, the net is bounded. In addition, since the token count in each invariant support in the initial marking is one, the net is also safe. Two properties related to the operation of the actual system can be deduced from the boundedness property of the Petri net model. There is no buffer overflow, no provision for $R1$ to access the buffer area when it is full. Also, there is no buffer underflow, no provision for $R2$ to access the buffer area when it is empty. When using the reachability tree, these properties follow from the net safeness. The entries in each marking, which represent the number of tokens in places $p_8$ and $p_9$, are either zero or one. Using invariants: The invariant support $\|y_4\|$ covers places $p_8$ and $p_9$. Since the token content in $\|y_4\|$ in the initial marking is one, there is only one token either on $p_8$ or $p_9$ at a time. Therefore there is neither butler overflow nor underflow.

**Conservativeness:** The Petri net shown in Fig. 1.12 is conservative. From the reachability tree, the net is conservative with respect to vector $\omega = [1, 1, 2, 1, 1, 1, 1]$. The weighted sum of tokens remains the same for each marking reachable from the initial marking, and equals four. Using invariants: The token content in each invariant support in the initial marking is one. The invariant supports $\|y_1\|$, $\|y_2\|$, and $\|y_4\|$ are mutually exclusive. The invariant supports $\|y_1\|$ and $\|y_3\|$ contain place $p_3$ as a common element. The invariant supports $\|y_2\|$ and $\|y_3\|$ contain place $p_6$ as a common element. Thus the weight of places $p_3$ and $p_6$ should be two for the net to be conservative. The implication of this property is that the number of robot arms operating in the assembly system is two and does not change. Also, the number of the space resources in the buffer area is one and does not change.

**Liveness:** The Petri net shown in Fig. 1.12 is live; all transitions are live.

Fig. 1.21 shows a reachability graph of the Petri net of Fig. 1.12. The reachability graph shown in Fig. 1.21 is a directed graph consisting of a set of nodes, and a set of directed arcs. The set of nodes represents all distinct labeled nodes in the reachability tree. The set of directed arcs, where each arc is labeled with a transition, represents all possible transitions between all distinct labeled nodes in the reachability tree.

By inspection, the net is $L4$-live, since for any marking reachable from making $M_0$, it is possible to ultimately fire any transition by executing some firing sequence. The invariants could be used to demonstrate "manually" that the net is live. However, for the net of this size, this would be a tedious procedure. As the net is live, the system cannot come to a standstill where no operation is possible.
Reversibility: The Petri net shown in Fig. 1.12 is reversible. Also by inspection, using the reachability graph, \( M_0 \) is reachable from any marking \( M \in R(M_0) \).

2.5 Petri Nets: conclusions

The development of Petri nets has been, to a large extent, motivated by the need to model the industrial systems. Ordinary Petri nets are not always sufficient to represent and analyze complex industrial and other systems. This prompted the development of new classes of nets, some of which are introduced briefly in this section.

Tokens in the ordinary nets have no identity. This poses some problems for modeling systems such as manufacturing and communication systems, which require the physical resources or messages, if represented by tokens, to have identity. Without this identity, it is impossible to trace the flow of different resources or messages in the system. A potential solution is to construct a model in such a way that the flow of each resource or message is associated with a dedicated subnet. Since the resources or messages share, in most cases, the same system, all these subnets are identical. This approach increases the graphical complexity of the model. In order to address this issue, Petri nets which allow for tokens to have distinct identity were proposed. These nets, referred to as high-level Petri nets, include predicate-transition nets, colored nets, and nets with individual tokens. In high-level Petri nets, a token can be a compound object carrying data, This data can be of arbitrary complexity involving integers, reals, text strings, records, lists and tuples. Ordinary and high-level Petri nets have the same descriptive power. High-level Petri nets provide much better structuring facilities than ordinary nets. The colored and predicate-transition nets are almost identical with respect to their description and simulation. However, there are some considerable differences with respect to formal analysis. The colored Petri nets were used in numerous application areas. These areas include communication protocols, production systems, etc. An important development in the area of high-level Petri nets was the introduction of object-oriented Petri nets. Object-oriented Petri nets can be considered as a special kind of high level Petri nets which allow for the representation and manipulation of an object class. In this class of nets, tokens are considered as instances or tuples of instances of object classes which are defined as lists of attributes. This type of net was used to model and analyze FMS systems, assembly tasks, and assembly systems.

The recognition of the need for the qualitative specification of the industrial control, as well as the need for representing approximate and uncertain information has led to the development of various types of fuzzy Petri nets. The definitions of these nets, to a large extent, were influenced by the various application areas. Fuzzy Petri nets have been used for knowledge representation and reasoning. Fuzzy Petri nets (Petri nets with objects) have been also used to model monitoring and control of a FMS system.

Ordinary Petri nets are not powerful enough for representing and studying some of the important properties of concurrent systems, such as eventuality (certain transitions must eventually fire; certain places must eventually have tokens), and fairness (if a transition becomes firable infinitely often, then it
must fire infinitely often), for instance. In order to address these issues, temporal Petri nets were introduced. In this class of nets, the timing constraints are represented by the operators of a variant of the propositional temporal logic of linear time, Typical operators used in this class of nets are next, henceforth, eventually, until, etc. Temporal Petri nets were used to model and analyze a handshake daisy chain arbiter, and the alternating bit protocol. The ability of temporal Petri nets to express eventuality makes this model suitable to represent and study the external functional behavior of systems. This functionality is expressed in terms of the input-output relationship, i.e., if a certain input pattern has been established, than eventually a certain output pattern will be generated.

Although attempts to combine Petri nets with other techniques, such as neural networks, fuzzy logic, etc., seem to be on the increase, it appears that the use of Petri nets is still restricted to research laboratories and academic institutions. This situation, to a large extent, results from the lack of widely available inexpensive software tools suitable for the development of industrial type of systems. These types of tools would be required to provide facilities for dealing with the application domain specific problems at a relatively skill-free level which would not require the knowledge of Petri nets, and the analysis methods. The facility for translating Petri net models to the executable code will be also essential. They will allow for rapid prototyping of the developed systems in the operational environment. In the past few years, a large number of tools have been reported in the Petri net literature. However, a majority of these tools are used mostly for research and educational purposes.

Another reason why the use of Petri nets is largely confined to academic and research institutions is the difficulty involved in constructing Petri net models. Constructing Petri net models of systems, especially large scale systems, is not a trivial task. It requires a great deal of experience. No methodology is available yet, which would allow for a fully automatic construction of Petri net models. In most cases, Petri net models are constructed in an ad hoc manner. However, attempts have been recently made to make this particular approach more systematic. In the past two decades, numerous approaches to the systematic construction of Petri net models have been proposed, and the work in this area still continues. These approaches, using the terms of software engineering, can be broadly classified into bottom-up, top-down, and hybrid approaches The reuse of Petri net models is also restricted. This is mainly due to the fact that Petri net models are, typically, constructed on a one-off basis. The development is, in most cases, not supported by proper documentation. It is clear that if Petri nets were to be widely used, especially by the industry people, methods, and the supporting tools, allowing for an automatic or semiautomatic construction of Petri net models from requirements specifications would have to be developed. In the past few years, a number of approaches have been reported which allow for the automatic construction of restricted classes of Petri net models from requirements specifications expressed using production rules, flow diagrams, state machines, temporal logic, application domain dependent semi-formal languages, etc.
3 Bibliography


