Software Design Methodologies

A Survey *

by

Jarir K. Chaar

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor

October 1987

Center for Research on Integrated Manufacturing

Robot Systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109-2110, U.S.A.

*This work has been supported by General Motors research grant 388975

Abstract

Software design methodologies are intended to permit a systematic approach to designing large software systems. Their development follows a life-cycle composed of a set of phases identified as requirements specification, design, implementation, system testing and system evolution respectively.

This report presents a review of the various software design methods that were developed to deal with the increasing complexity of today's computer controlled systems and the improved capabilities offered by modern computer systems. It is concluded that these methods do not address all the unique aspects of distributed real-time embedded systems, and that there is a need for a software design methodology that can model both their structural and behavioral aspects. This methodology could be managed by an improved software life-cycle model based on the use of system prototyping and simulation techniques. The Software Engineering principles of modular programming and information hiding could be promoted by the above methodology through the use of the Ada programming language.

Contents

1 Introduction 1

2 Software Engineering Methodologies 2

2.1 Requirements Definition 4

2.1.1 Relational Notations 4

2.1.2 State-oriented Notations 6

2.2 Architectural Design 10

2.2.1 Process-Oriented Design Techniques 10

2.2.2 Object-Oriented Design Techniques 14

2.2.3 Conclusion 16

3 Real-Time Design Methods 17

3.1 SREM 18

3.2 SARA 20

3.3 GRAFCET 22

3.4 MASCOT 24

3.5 DARTS 25

3.6 PROT nets 27

3.7 The Transformation Schema 29

3.8 PAISLey 31

3.9 DAPSE 32

4 Towards a real-time design methodology 33

4.1 Concurrency Requirements 34

4.2 Timing Requirements 34

4.3 Reliability Requirements 35

4.4 Hardware Requirements 36

5 Conclusion 37

1 Introduction

The first burst of activity in software design methods occurred during the late 1960's and early 1970's. It was assumed that software development began with writing a natural-language requirements document and ended with programming, testing and installation. *Design* was a well-defined activity that happened between requirements analysis and programming, and was largely concerned with how to decompose the required system into manageable pieces. Many methods and criteria were proposed for performing this decomposition. These activities resulted in a standard software life-cycle model; a framework used in managing software design projects.

The most obvious difference between design research of the 1970's and design research of the 1980's is that there is no longer consensus on the software life-cycle or on the scope of design activities during software development. The standard software life-cycle was extended to include more steps due to the increased complexity of the systems that were automated, and the increased capabilities the computer systems offered in hardware and software; technological advancements led to distributed systems and computer networks, while software development resulted in distributed languages. The requirements specification, design, implementation, system testing and system evolution phases were identified in the improved software life-cycle, and, techniques for improving the outcome of each of the phases were researched. The software design process was improved by automating many steps in the process in order to reduce the cumulative effect of an error committed in any of the software design phases. System prototyping and simulation techniques were introduced to help assess the performance of the resulting software system. Currently, a large set of techniques for designing general-purpose centralized and distributed systems is available. The design philosophy and the specific steps followed by each method are application-dependent.

The domain of distributed real-time embedded systems has only recently received increased attention. This domain is characterized by a set of properties not present in general-purpose systems; the interaction between a distributed real-time embedded system and its environment is a crucial part of the behavior of the system. A computer-aided software technology for developing distributed real-time embedded systems should address the particular problems for the requirements analysis, design, implementation, system testing and system evolution phases of these systems. An adequate model of the above systems should encompass both data and control structures, and model both the structure and behavior of the system. In particular, timing constraints, priorities, exception-handling mechanisms, fault-tolerance features, underlying hardware components and interaction with the outside environment are essential for the proper functioning of a distributed real-time embedded system.

This report presents an overview of software engineering methodologies; software design techniques in general and real-time design methods in particular are discussed. The recent attempts to model certain aspects of distributed real-time embedded systems are reported; it is concluded that there is a need for a complete methodology for designing these systems. This methodology could be managed by an improved software life-cycle model based on the use of system prototyping and simulation techniques. The Software Engineering principles of modular programming and information hiding could be promoted by the above methodology through the use of the Ada programming language.

2 Software Engineering Methodologies

The IEEE standard glossary of software engineering terminology [IEEE 83] defines **Software Engineering** as: *The systematic approach to the development, operation, maintenance, and retirement of software,* where **software** is defined as: *Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.*

The primary goals of Software Engineering are to improve the quality of software products and to increase the productivity and job satisfaction of Software Engineers. Software Engineers are concerned with issues of analysis, design, verification and testing, documentation, software evolution, and project management. These issues cover the entire scope of software development, and, managerial and technical methods were developed to address their specific aspects. A collection of such methods is termed a *methodology* [Bergland 81].

The software development process starts by defining a product life-cycle model. The life-cycle encompasses all activities required to define, develop, test, deliver, operate, and maintain a software product. The model provides a basis for categorizing and controlling these various activities. The standard life-cycle model is the *phased model* [Fairley 85].

The phased model segments the software life-cycle into a series of successive activities. Each phase requires well-defined input information, utilizes well-defined processes, and results in well-defined products. Resources are required to complete the processes in each phase, and each phase is accomplished through the application of explicit methods, tools, and techniques. Products cascade from one level to the next in smooth progression; the result is a **waterfall chart** [Fairley 85]. The activities of analysis, software design, implementation, system testing and evolution are performed iteratively.

- *Analysis* consists of *Planning* and *Requirements Definition*:

  1. *Planning* includes understanding the customer's problem, performing a feasibility study, developing a recommended solution strategy, determining the acceptance criteria, and planning the development process. The products of planning are a *System Definition* and a *Project Plan.* The System Definition is expressed in a natural language, and may incorporate charts, figures, graphs, tables, and equations of various kinds. The Project Plan contains the organizational structure for the project, the preliminary development schedule, preliminary cost and resource estimates, preliminary staffing requirements, tools and techniques to be used, and standard practices.
  2. *Requirements Definition* is concerned with identifying the basic functions of the software component in a hardware/software/people system. The product of requirements definition is a specification that describes the processing environment, the required software functions, performance constraints on the software (size, speed, machine configuration), exception handling, subsets and implementation priorities, probable changes, and likely modifications, and the acceptance criteria for the software.

- *Software Design* consists of *Architectural Design* and *Detailed Design.*

1. *Architectural Design* involves identifying the software components, decoupling and decomposing them into processing modules and conceptual data structures, and specifying the interconnections among components. The product of architectural design is a conceptual schema of the software system. This schema identifies the system components and their associated data structures, and, defines their relationships with the other components in the system and their environment.
2. *Detailed Design* is concerned with the details of *how to*: how to package the processing modules and how to implement the processing algorithms, data structures, and interconnections among modules and data structures. Detailed Design involves adaptation of existing code, modification of standard algorithms, invention of new algorithms, design of data representations, and packaging of the software product. Detailed design is strongly influenced by the programming language used to implement the system, but detailed design is not concerned with the syntactic aspects of the implementation language or the level of detail inherent in expression evaluation and assignment statements. The product of detailed design is a pseudo-code specification of the system.

- *Implementation* involves translation of design specifications into source code and debugging, documentation, and unit testing of the source code. Modern programming languages provide many features to enhance the quality of source code. These include structured control constructs, built-in and user-defined data types, secure type checking, flexible scope rules, exception handling mechanisms, concurrency constructs, and separate compilation of modules.

- *System Testing* involves two kinds of activities: *integration testing* and *acceptance testing*. Developing a strategy for integrating the components of a software system into a functioning whole requires careful planning so that modules are available for integration when needed. Acceptance testing involves planning and execution of various types of tests in order to demonstrate that the implemented software system satisfies the requirements stated in the requirements document.

- *System evolution* activities include enhancement of capabilities, adaptation of the software to new processing environments, and correction of the resulting software bugs.

The phased model of the software life-cycle is admittedly simplistic. There are no milestones in the model, no mention of the documents generated or the reviews conducted during the development process, no indication of the relative effort devoted to each phase, no indication of the role of prototypes in software development, no indication of quality assurance activities, and only cursory indication of the constant verification of work products that must occur throughout the life-cycle.

Software development never proceeds in a smooth progression of activities as indicated in the waterfall chart. There is more overlap and interaction between phases than can be indicated in a simple two-dimensional representation. Nevertheless, the phased model of the software life-cycle is a valid model of the development process in situations where it is possible to write a reasonably complete set of specifications for the software product at the beginning of the life-cycle. This typically occurs when the developers have previously developed similar systems. The software development process

is improved by establishing milestones, review points, standardized documents and management sign-offs, and, the incorporation of prototyping.

A prototype is a mock-up or model of a software product. In contrast to a simulation model, a prototype incorporates components of the actual product. Typically, a prototype exhibits limited functional capabilities, low reliability, and/or inefficient performance. Prototyping illustrates input data formats, messages, reports, and interactive dialogues for the customer. It is implemented to explore technical issues in the proposed product. Development of a new product will probably involve some prototyping during the planning and analysis phase, or the product may be developed by iterating through a series of successive design and implementation refinements.

The following sections detail the notations used in defining the requirements of a software product and the techniques used in designing its architecture.

2.1 Requirements Definition

Consistency and completeness of the Software Requirements Specification document can be enhanced by the use of formal notations and automated tools. Formal notations have the advantage of being concise and unambiguous, they support formal reasoning about the functional specifications, and they provide a basis for verification of the resulting software product.

Both *relational* and *state-oriented* notations are used to specify the functional characteristics of software [Fairley 85]. Relational notations are based on the concepts of entities and attributes. Entities are named elements in a system. Attributes specify permitted operations on entities, relationships among entities, and data flow between entities. Relational notations include *implicit equations*, *recurrence relations*, *algebraic axioms*, and *regular expressions*.

State-oriented notations are based on the concept of state. The state of a system is the information required to summarize the status of system entities at any particular point in time; based on the system definition, the current state and the current stimuli determine the next state. State-oriented notations include *decision tables*, *event tables*, *transition tables*, *finite-state mechanisms*, and *Petri nets*.

2.1.1 Relational Notations

- *Implicit equations* state the properties of a solution without stating a solution method. The desired functional behavior is embedded within relationships among system entities. *Mathematical problems* are often specified using implicit equations; however, not all implicitly specified problems are guaranteed to have algorithmic solutions.

- *Recurrence relations* consist of an initial part called the basis and one or more recursive parts. The recursive part(s) specify a desired outcome in terms of identical outcomes for simpler versions of the problem. Recurrence relations are often used to specify *signal processing* and other similar *time series operations*. Although specifications in the form of recurrence relations

suggest recursive implementations, it is often possible to develop more efficient algorithms using iteration.

- *Algebraic axioms* are used to specify *data abstraction.* Data abstraction emphasizes functional properties and suppresses representation details. Specification of an abstract data type using algebraic axioms involves defining the syntax of the operations and specifying axiomatic relationships among the operations. The syntactic definition specifies names, domains, and ranges of operations to be performed on the data objects, and the axioms specify interactions among operations.

  Using a state-oriented notation in conjunction with algebraic axioms allows precise specification of the entity names used in the axioms. This technique combines the advantages of the algebraic approach (precise specification of interactions among operations) and finite-state approach (precise specification of the behavior of the individual operations).

  Algebraic axioms can be used in three distinct ways: as *definitional tools* of new operations in terms of existing ones; as foundations for *deductive proofs* of desired properties, and as frameworks for examining the *completeness* and *consistency* of functional requirements.

  There are two types of completeness concerns for a requirement specification: *external completeness* and *internal completeness.* A requirement specification is externally complete if all the desired properties are specified. A requirement specification that is internally complete has no undefined entities in the specification. Consistency is concerned with a unified interpretation of the relationships among specifications.

- *Regular expressions* can be used to specify the syntactic structure of symbol strings. Each set of symbol strings specified by a regular expression defines a formal language. They are formed by recursively applying the rules of *alternation, concatenation* and *closure* to the *atoms* in the alphabet of interest. Hierarchical specifications can be constructed by assigning names to regular expressions and using the names in other regular expressions. Typical applications of regular expressions include specification of valid data streams, the syntax of user command languages, and legal sequences of events in a system.

  Regular expression notation can be extended to allow modeling of concurrency. By definition, the effect of concurrent execution of two software components is the same as interleaving their execution histories. The syntax of each software component will be specified by a regular expression and, when applied, the *shuffle* operator interleaves the regular expressions for both components while preserving the original ordering of atoms within each expression; the resulting expressions are called *event expressions* and *flow expressions* [Shaw 80].

  Event expressions impose some restrictions on shuffling; only the subset of all possible shuffles that preserves the *atomicity* of critical sections, message passing, and other synchronization constraints is allowed. The synchronization scheme in flow expressions has a more direct programming connection than that in event expressions, and is the major distinguishing difference between the notations. A flow expression can be blocked for shuffling by enclosing it in a wait/signal pair similar to binary semaphores. The enclosed expression will be treated

as indivisible when interleaving it with other flows. Hence, atomicity is implicit in event expressions and explicit in flow expressions resulting in a direct mapping between the latter and concurrent programming languages constructs.

Path expressions are another useful notation based on regular expressions. They can be used to specify the sequencing of operations in concurrent systems, and handle some aspects of the description analysis and implementation of the constraints on operation executions [Lauer 80].

An extension of flow expressions and path expressions forms the syntax of the *DREAM Design Notation* (DDN), a design description language developed as the basis of the *Design Realization, Evaluation and Modeling* (DREAM) system [Riddle 79]. DREAM is considered an early attempt to provide automated support for the design of concurrent systems by providing closed-form descriptions of the sequences of certain events occurring in behaviors of the system.

While DREAM could only describe embedded systems in which the set of constituent processes and the communication paths connecting them remained static, the *Dynamic Process Modeling Scheme* (DPMS) was developed for describing systems with dynamic structure [Avrunin 85]. DPMS is based on *constrained expressions*. The constrained expression representation of a distributed system consists of a *system expression* and a collection of *constraints*. The system expression is a regular expression over an augmented alphabet derived from a description of the system. The constraints may be thought of as imposing requirements on a sequence of events that must be satisfied if the sequence is to occur in a behavior of the system [Avrunin 86]. In DPMS, the structure of a dynamically structured distributed system can be altered either by adding or deleting processes or by adding or deleting interprocess communication channels.

2.1.2 State-oriented Notations

- *Decision tables* provide a mechanism for recording complex decision logic. Decision rules are used to specify the desired actions of the system. Contradictory rules permit the specification of nondeterministic and concurrent actions. Multiply-specified action may be desired or may indicate a specification error. A *Karnaugh map* can be used to check a decision table for completeness and multiply-specified actions.

- *Event tables* specify actions to be taken when events occur under different sets of conditions. A two-dimensional event table relates actions to two variables; $f(M, E) = A$, where M denotes the current set of operating conditions, E is the event of interest, and A is the action to be taken. Tables of higher dimensions can be used to incorporate more independent variables. A set of sequential or concurrent actions may be specified.

- *Transition tables* can be used to specify the next desired state of a system given the current state and the current stimuli; if, when in state S_i, condition C_j results in a transition to state S_k, we say $f(S_i, C_j) = S_k$. A transition table can be augmented to indicate actions to be performed and outputs to be generated in the transition to the next state. Transition diagrams are alternative representations of transition tables. They both are representations for finite state automata.

Using techniques from automata theory, it can be shown that every regular expression has a corresponding transition table and vice versa. Transition tables and transition diagrams thus provide mechanisms for specifying the various states that a system must occupy when processing symbols from strings specified by the corresponding regular expressions.

Extended transition diagrams [Wasserman 79] [Wasserman 86] form useful mechanisms for modeling interactive systems. The *User Software Engineering* (USE) methodology extends the transition diagrams syntactically by allowing a hierarchical structure to manage the complexity of the diagrams, and semantically by allowing input variables at specified states in the diagram [Wasserman 85]. When a transition hits the entry point of a lower level diagram, the traversal of the higher level diagram is suspended and control is transferred to the lower level one. The new diagram is traversed until its exit point is reached, at which point control returns to the higher level diagram. Variables may be assigned by appearing on a transition, so that user input causes the transition and makes the assignment. Once a variable has been assigned a value, it may subsequently be updated or used in messages and in actions

Decision tables, event tables, and transition tables are notations for specifying actions as functions of the conditions that initiate those actions. Decision tables specify actions in terms of complex decision logic, event tables relate actions to system conditions, and transition tables incorporate the concept of system state. The notations are of equivalent expressive power; a specification in one of the notations can be readily expressed in the other two. The best choice among them for requirements specification depends on the particular situation being specified.

- *Finite-state mechanisms* model a software system as a set of processes interconnected by data streams. Each of the data streams can be specified using a regular expression, and each of the processes can be described using a transition table. This modeling scheme, however, does not allow the specification of concurrency in the system.

  One drawback of finite-state mechanisms is the so-called *state explosion phenomenon.* Complex systems may have large numbers of states and many combinations of input data. Specifying the behavior of a software system for all combinations of current state and current input can become unwieldy. Hierarchical decomposition is one technique for controlling the complexity of a finite-state specification. Using hierarchical decomposition, higher level concepts are given names, the meaning and validity of which are established at a lower level.

  Finite-state machines have been extended to model the flow of control in real-time systems [Du Plessis 86] [Auernheimer 86] [White 87]. The extension incorporates the concept of transition event representing conditions which reflect the arrival of input signals. Enabling predicates, represented as a transition event, may be required to be true for a transition to occur. In this sense, a state may be represented as a Boolean combination of conditions. Transition actions are outputs activated when the associated transition event occurs, and are a means of specifying the flow of control within the system.

- *Petri nets* overcome the limitations of finite state mechanisms in specifying parallelism. They are used to model various aspects of concurrent systems, and detect *synchronization, mutual*

exclusion, and *deadlock* situations in these systems.

A Petri net (fig. 1) is represented as a bipartite directed graph [Peterson 81]. The two types of nodes in a Petri net are called *places* or *conditions* and *transitions* or *events*. Places are marked by *tokens*. The maximum number of tokens a place can hold defines its *capacity*. Places and transitions are connected via directed edges. The maximum number of tokens an edge can transfer defines its *weight*. Each transition in the net corresponds to a task activation and places are used to synchronize processing.

A Petri net is characterized by an initial marking of places and a firing rule. A firing rule has two aspects: a transition is enabled if each of its input places has at least as many tokens as the weight of the edge from the place to the transition. An enabled transition can fire if the capacity of each of its output places will not be exceeded after the firing; when a transition fires, each of its input places is decreased by the weight of the edge connecting the input place to the transition, and the number of tokens of each of its output places is increased by the weight of the edge connecting the transition to the output place. The new configuration of tokens in the net defines a new marked net.

Transitions in a Petri net could represent events in a real system. A marked net then represents the coordination or synchronization of these events. The movement of tokens clearly shows which conditions cause a transition to fire and which conditions come into being on the completion of firing. The Petri net serves to relate the models to the specified operating rules to form an event-based simulation [Agerwala 79].

For a Petri net which is to model a real hardware device, one of the more important properties is *safeness*. A place in a Petri net is safe if the number of tokens in that place never exceeds one. Thus, the place can be implemented by a single flip-flop; the flip-flop is active when a token is present in the place it implements, and is inactive in the absence of the token. A Petri net is *safe* if all places in the net are safe. Every path expression defines a safe net which decomposes into a set of state machines corresponding to the regular expressions from which the path expression is composed [Lauer 80].

Safeness is a special case of the more general *boundedness* property. A place is *k-safe* if the number of tokens in that place cannot exceed an integer k. Thus, the place can be implemented by a modulo-k counter. A place is bounded if it is k-safe for some k; a Petri net is bounded if all places are bounded. A bounded net can be realized in hardware, while a Petri net with an unbounded place could not in general be implemented in hardware.

Petri nets can be used to model resource allocation systems. In these systems some tokens may represent the resources, and, are neither created nor destroyed. For these types of Petri nets, *conservation* is an important property. Conservation requires that the total number of tokens in the net remains constant.

The resource allocation systems may be analyzed for potential deadlocks using their Petri nets. A deadlock in a Petri net is a transition (or set of transitions) which cannot fire. A transition is *live* if it is not deadlocked. Thus, it is always possible to maneuver the Petri net from its current marking to a marking which would allow the transition to fire. Liveness of the

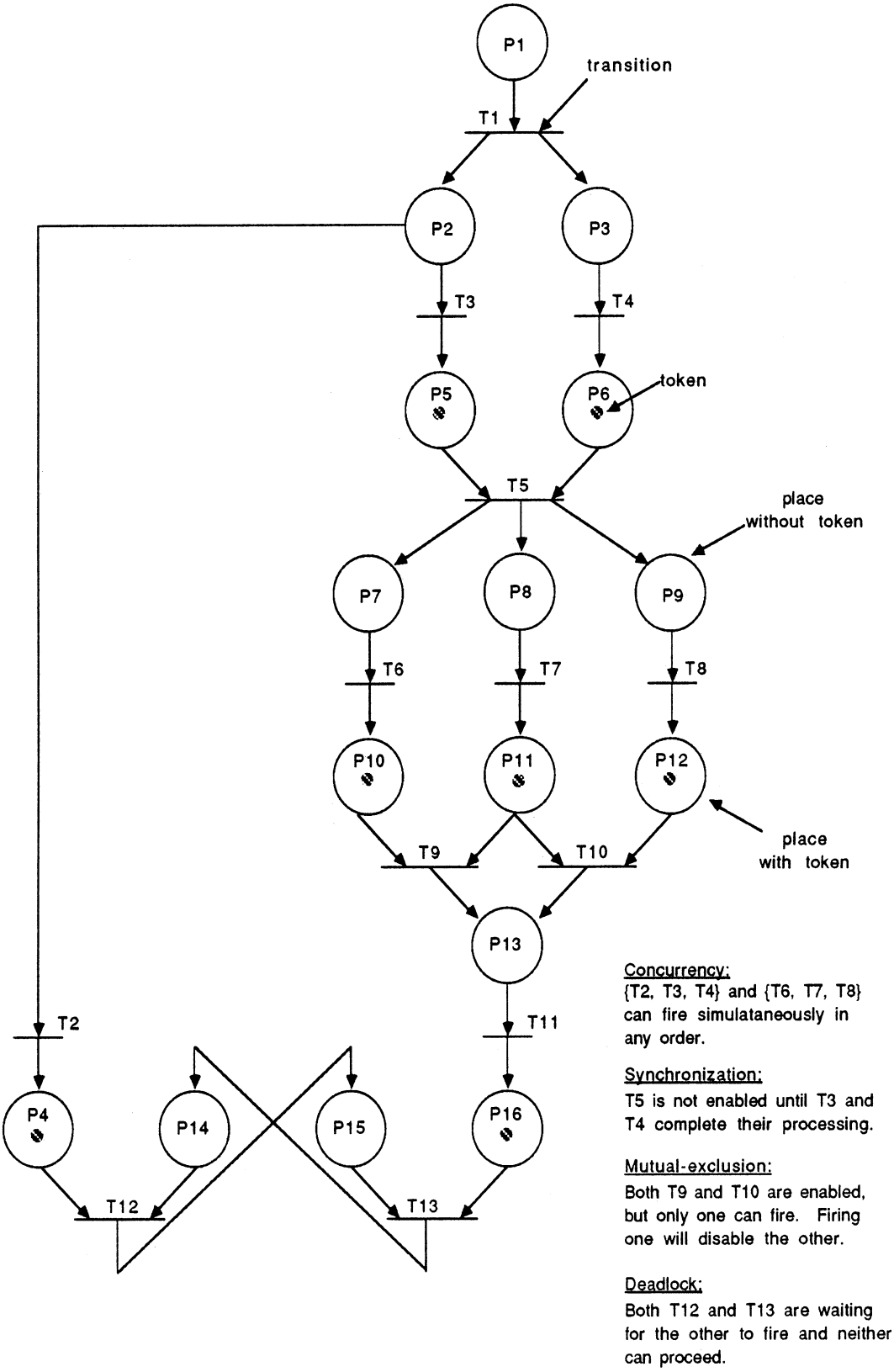


Figure 1: A Petri net model

Petri net marks the absence of deadlock in the real system.

In a Petri net, an *Invariant* is a minimal set of places whose total number of tokens remains constant. The net is bounded if each place in the net is in some invariant, and the net has a constant total number of tokens. The net is conservative if the invariants are disjoint and inclusive.

Bounded, conservative Petri nets are equivalent to finite-state automata. In a finite-state automaton representation of a Petri net, the state of the net is given by the marking configuration, and transitions leading to next states change the markings to reflect changes in the next state. Despite the above equivalence, Petri nets provide a more convenient notation for specifying and analyzing concurrent systems; unbounded and non-conservative Petri nets provide more powerful mechanisms than do finite-state automata.

2.2 Architectural Design

In the architectural design phase, requirement specification is transformed into a system structure [Freeman 83]. Techniques in the architectural design stage can be divided into two groups: *process-oriented* and *object-oriented* approaches. The process-oriented approach models the behavioral aspects of a software system, in contrast with the object-oriented approach where the structural aspects of a software system are modeled [Yau 86].

2.2.1 Process-Oriented Design Techniques

The process-oriented design technique emphasizes the data flow component of software architecture. This section will discuss the process-oriented design techniques of *Functional Decomposition, Data Flow Design* and *Data Structure Design.*

- *Functional Decomposition:* The divide-and-conquer strategy of *stepwise refinement* forms the basis of this approach. Stepwise refinement is a top-down technique for decomposing a system from high-level specifications into more elementary levels. *Information hiding* is used as a decomposition criterion in the above approach. Using this criterion, each module in the system hides the internal details of its processing activities and modules communicate only through well-defined interfaces. Functions can be decomposed with respect to time, shared data, data flow, and control flow [Ramamoorthy 84]. Functional decomposition lacks a measure of how good or bad a decomposition of a function is.

- *Data Flow Design Methods:* These methods use various mapping functions to transform information flow into software structure. They are best suited for design problems where a well-defined data flow can be derived from the problem specification. Data flow is used to decompose the functions of a given process, and, to model message passing between the asynchronous processes of a distributed or parallel architecture [Ramamoorthy 84]. Typical data flow design methods are:

- *Structured Design* (SD). The basic approach in structured design is the systematic conversion of *data flow diagrams* developed during Requirements Definition into *structure charts*. Design heuristics such as *coupling* and *cohesion* are used to guide the design process [Stevens 74].

  Data flow diagrams (fig. 2) specify data sources and data sinks, data stores, transformations to be performed on the data, and, the flow of data between sources, sinks, transformations and stores. A data store is a conceptual data structure, in the sense that physical implementation details are suppressed; only the logical characteristics of data are emphasized on a data flow diagram. Data flow diagrams can be hierarchically decomposed by specifying the inner workings of the functional nodes using additional data flow diagrams.

  A structure chart is a hierarchical ordering of the modules in a software system. It is represented as an acyclic, directed graph. The data passed between modules is indicated on the arcs connecting them or in an associated table. Directly recursive routines are indicated by placing closed arcs on the nodes for these routines. Indirectly recursive routines violate the hierarchical structuring rule, and should generally be avoided. Allowable exceptions include mutually recursive routines and coroutines.

  Coupling measures the degree to which two distinct modules are bound together, and cohesion is a measure of the relationship of elements within a module to one another. A well-designed system exhibits a low degree of coupling between modules and a high degree of cohesion among elements in each module. Parameter passing between modules, referencing other modules by name and maximizing the number of data parameters while minimizing the number of control parameters help achieve a low degree of coupling between the modules of a program. On the other hand, a high degree of cohesion is achieved when each module performs a single function or contains a complex data structure and the routines to manipulate this data structure. The approach is more suitable for a design problem with no data structures available.

- *Structured Analysis Design Technique* (SADT). SADT is based on *Structured Analysis* (SA) [Ross 77]. SA is a graphical language used for explicitly expressing hierarchical and functional relationships among objects and activities.

  A SADT model consists of an ordered set of SA diagrams (fig. 3). Two basic types of SA diagrams are the *activity diagram* (actigram) and the *data diagram* (datagram). On an actigram the nodes denote activities and the arcs specify data flow between activities. Actigrams are thus the SA version of data flow diagrams. Datagrams specify data objects in the nodes and activities on the arcs. Actigrams and datagrams are thus duals. In practice, activity diagrams are used more frequently than data diagrams; however, data diagrams are used to indicate all activities affected by a given data object, and to check the consistency and completeness of an SADT model by constructing data diagrams from a set of activity diagrams.

  Four types of arcs can be attached to each node of a SADT model. Arcs coming into the left side of a node carry inputs and arcs leaving the right side of a node convey outputs.

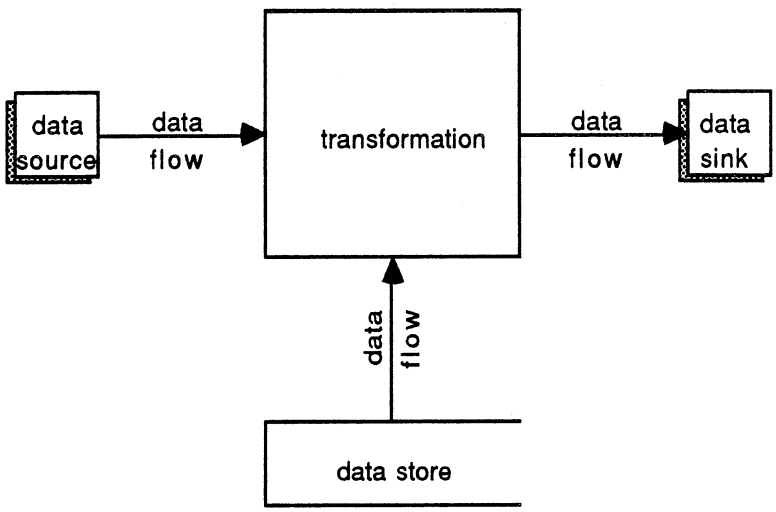


Figure 2: A formal data flow diagram

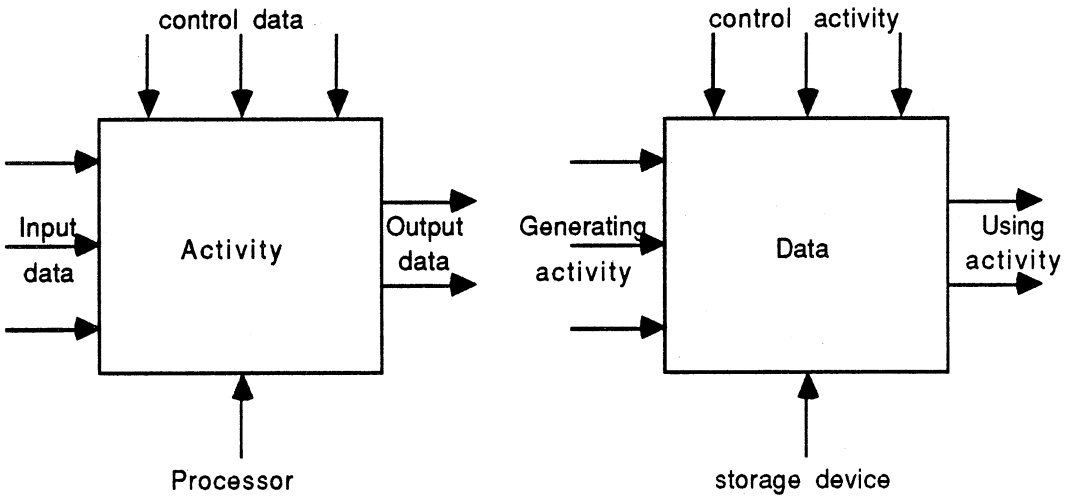


Figure 3: SA diagram components

Arcs entering the top of a node convey control and arcs entering from the bottom specify mechanism. Outputs provide input and control for other nodes. Outputs from some nodes are the system outputs to the external environment. Every output must connect to other nodes or to the external environment. Similarly, input and control must come from the output of other nodes or from the external environment.

In an activity diagram the inputs and outputs are data flows and the mechanisms are processors. Control is data that is used, but not modified by the activity. In a data diagram the input is the activity that creates the data object and the output is the activity that uses the data object. Mechanisms in data diagrams are the devices used to store the representations of data objects. Control in a datagram controls the conditions under which the data object will be accessed. Major strengths of SADT include the top-down methodology inherent in decomposing high-level nodes into subordinate diagrams, the distinction between input, output, control and mechanism for each node, and the duality of activity and datagrams.

- *Structured System Analysis* (SSA). SSA is used primarily in traditional data processing environments [Gane 80]. Like SADT, SSA uses a graphical language to build models of systems. Unlike SADT, SSA incorporates database concepts; however, SSA does not provide the variety of structural mechanisms available in SADT. There are four basic features in SSA: *data flow diagrams*, *data dictionaries*, *procedure logic representations*, and *data store structuring techniques*.

  SSA data flow diagrams are similar to SADT actigrams, but they do not indicate mechanism and control, and an additional notation is used to show data stores. A data dictionary is used to define and record data elements. Processing logic representations, such as decision tables and structured English, are used to specify algorithmic processing details. An SSA specification can be defined to the level of detailed design. A relational model is used to specify data flows and data stores. The relations are stored in *third-normal form* (3NF).

- *Data Structure Design Methods:* These methods emphasize the structure of data. They view software as a mechanism by which input data are transformed into output data. They construct architectural and detailed design concurrently. They are too difficult to apply for highly concurrent programs with a lot of interprocess communication [Ramamoorthy 84]. Two methods fit in this category:

  - *Jackson's Design Methodology.* **Jackson Structured Programming** (JSP) method is a systematic technique for mapping the structure of a problem into a program structure to solve the problem.

    Input and output structures are specified using a *graphical notation* to specify data hierarchy, sequences of data, repetition of data items, and alternate data items. This notation is the graphical equivalent of *regular expressions* [Jackson 76] [Cameron 82]. The order in which the actions can happen is a tree structure; the leaves are the actions; all the other nodes describe sequential relationships between actions or between groups of actions.

Next, both input and output structures are combined into a program structure that maps inputs into outputs. Labels on data items in the resulting structure are converted to process names that perform the required processing of the data items. The final step expands the structural model of the program into a detailed design model containing the operations needed to solve the problem.

Detailed design involves developing a list of operations needed in the program, associating the operations with program structure, and translating the annotated structure diagram into schematic logic (pseudo-code).

Jackson Structured Programming is widely used and is quite effective in situations where the input and output data structures can be defined in a precise manner. It appears to be most effective in data processing applications.

More recently, the **Jackson System Development** (JSD) for software design was developed [Cameron 86]. This approach involves modeling the real world phenomenon of interest as a network of sequential processes that communicate using serial data streams. The processes communicate by writing and reading messages and by a one-writer-many-readers form of shared variable communication. Detailed design of the processes can be accomplished using JSP. Because the process model may result in excessive numbers of processes, it may be necessary to perform a number of transformations on the model to reduce the number of processes. Transformations are performed during detailed design and implementation to produce a system that can be implemented on conventional machine architectures.

- *Warnier's Methodology.* **Structured Systems Design** (SSD) concentrates on the principal outputs of the system and works *backwards* to design the logical database, the physical database, and the inputs of the system [Orr 78].

  The method uses four types of design representations: *data organization diagram, logical sequence diagram, instruction list and pseudo-code.* Data organization diagram describes the logical hierarchy of input and output data in a tree structure format. Logical sequence diagram represents its logical flow. Instruction list contains instructions used in the design. Pseudo-code is the final description of the design. This approach provides more detailed procedures to software design than Jackson's Methodology.

2.2.2 Object-Oriented Design Techniques

The object-oriented design technique emphasizes the data design component of software systems and the techniques for deriving the data design, and, is best illustrated by the *Object-Oriented Design Methodology.*

- *Object-Oriented Design Methodology.* Object-oriented development of a system is a partial life-cycle method based upon the concept of an object [Booch 86]. It focuses upon the design and implementation stages of software development.

An *object* is an entity whose behavior is characterized by its present state, and the actions it suffers and/or requires of other objects. Because of the existence of state, objects are not input/output mappings as are procedures or functions. Thus, objects and processes, which are input/output mappings, are distinct.

In a functionally decomposed system, there tends to be a single thread of control that follows the hierarchical lines of decomposition. The primary criteria for decomposition is that each module in the system represents a major step in the overall process. On the other hand, the objects of an object-oriented system may be independent and autonomous, resulting in many simultaneous active threads of control.

Objects with identical characteristics and behavior are grouped into classes. Large object-oriented systems tend to be built in layers of abstraction, where each layer denotes a collection of objects and classes of objects with restricted visibility to other layers; such a collection of objects forms a *subsystem*. The components of a subsystem tend to be structurally flat rather than being strictly hierarchical and deeply nested.

Object-oriented development starts by identifying the objects and their attributes. Similar objects are grouped into object classes. Next, the operations suffered by and required of each object are identified together with any constraints upon time or space that they must observe. The operations suffered by an object define the activity of an object when acted upon by other objects. Those required of the object help establish its dependency upon any specific objects in the system; the object is inherently reusable when dependent only upon other classes of objects. The interface of each object in the system is next designed and implemented.

Object-oriented development is a problem-oriented approach more suitable to control-dominant or real-time applications where the autonomy of entities in the system and their synchronized behavior are the keys to a proper functioning of the system. However, its theoretical foundation needs more study [Ramamoorthy 84] [Yau 86].

Object-oriented development should be coupled with appropriate requirements and analysis methods for expressing the software requirements of systems. Two such methods are appropriate:

- PSL/PSA. The *Problem Statement Language* (PSL) and *Problem Statement Analyzer* (PSA) were developed as components of the Information System Design and Optimization System (ISDOS) project [Teichroew 77]. PSL is based on the Entity Relationship Attribute (ERA) model of systems. This model describes a system as a set of *objects (entities)*, where each object may have *properties (attributes)*, and each property may have property values. Objects may be interconnected; the connections are called *relationships* [Teichroew 80]. PSL constructs are used to formally describe these objects, properties and relationships.

  The *Problem Statement Analyzer* (PSA) is an automated analyzer for processing requirements stated in PSL. PSA operates on a database of information collected from a PSL description. It is a useful tool for documenting and communicating software requirements [Winters 81]. The PSL description of a system is checked for consistency and completeness using PSA. Next, a concise requirements specification document of the system is

generated.

PSL/PSA is sometimes criticized because it does not incorporate any specific problem-solving techniques. PSA does provide the capability to define new PSL constructs and new report formats, which allows the system to be tailored to specific problem domains and specific solution methods [Masiero 86].

PSL/PSA concepts were used as the basis of many information systems design methodologies. SYSDOC/SYSTEMATOR [Aschim 82] coupled with a natural language interface and a semantic database, and, the System Descriptor and Logical Analyzer (SDLA) [Knuth 82] coupled with a semantic database are two PSL/PSA based systems.

- *Requirements Modeling Framework* (RMF). RMF is based on SADT, semantic database modeling, and knowledge representation. The view is adopted that software requirements involve the representation (modeling) of considerable real-world knowledge, not just functional specifications [Greenspan 82].

  RMF captures the properties of three types of conceptual entities (*objects, activities,* and *assertions*). By grouping all entities into *classes* or *metaclasses*, and by organizing classes into hierarchies, RMF supports three abstraction principles (*aggregation, classification* and *generalization*). Aggregation allows an entity to be viewed as a collection of components or parts. Classification allows a new entity, the class, to capture common characteristics shared by a group of entities. Generalization captures the common characteristics of several classes.

  Aggregation allows one to consider an entity while ignoring further details about the components. Classification allows one to consider only the characteristics shared by the instances of a class and to ignore individual differences. Generalization allows one to consider only those properties that a collection of classes have in common without considering the classes' individual differences. These capabilities are incorporated in the Requirements Modeling Language (RML), an object-oriented language [Greenspan 86]. They are used to capture knowledge of the conceptual model of a system by describing its components, their attributes and their interfaces with other system components.

  The Entity Relation Attribute Event (ERAE) model [Dubois 86] extends RMF by expressing both invariant and time-dependent real-world knowledge.

2.2.3 Conclusion

In principle, process-oriented and object-oriented techniques represent complementary views of a system. Systems are not strictly process-oriented or object-oriented. They have aspects of both. The object-oriented approaches are more suitable for modeling the static structure of a system by identifying the different entities in the system and expressing their attributes and dependencies on other entities. The process-oriented approaches are more fit for modeling the dynamic behavior of a system through the specification of both data and control flows.

Most current process-oriented and object-oriented techniques are database-oriented techniques. They fail to properly address control-dominant or real-time applications by completely ignoring the

issues of performance and security. Nevertheless, Structured Analysis Design Technique (SADT) and Object-Oriented Design Methodology have been classified as real-time design methods [Kelly 87] [Gomaa 84].

A complete methodology for software development in real-time embedded systems should provide integrated support to two fundamental issues: specification and simulation [Bruno 86b]. In fact, the specification phase is the starting point for control software development since it states what the system is required to accomplish and what constraints and performance objectives are to be satisfied [Dasarathy 85] [Davis 82]. On the other hand, simulation models are extensively used during the design of the system in order to get insight into its behavior and to investigate and evaluate the effects of alternative decisions in advance.

In addition, the design process of real-time embedded systems should stress the *reusability* concept. Reusability allows using designs as well as code repeatedly, avoiding redundant work wherever possible and amortizing the cost of a software package over the largest possible number of systems [Freeman 87].

3 Real-Time Design Methods

The traditional considerations of hierarchy, information hiding, and modularity are important concepts in the design of real-time systems. However, these concepts are typically applied to the individual components of a real-time system. High-level issues of networking, performance, and reliability must be analyzed before the component nodes or processes are developed.

Real-time systems typically sense and control external devices, respond to external events, and share processing time between multiple tasks. Processing demands are both cyclic and event-driven in nature. Event-driven activities may occur in bursts, thus requiring a high ratio of peak to average processing. Real-time systems often form distributed networks; local processors may be associated with sensing devices and actuators.

A real-time network for process control may consist of several minicomputers and microcomputers connected to one or more large processors. Each small processor may be connected to a cluster of real-time devices. Process control systems often utilize communication networks having fixed, static topology and known capacity requirements. In contrast, more elaborate real-time systems provide dynamic configuration of the network topology and support unpredictable load demands.

Several methods have been developed to support the analysis and design of real-time systems. They include *SREM, SARA, GRAFCET, MASCOT, DARTS, PROT nets, the Transformation Schema, PAISLey* and *DAPSE*. Most of these methods use Petri nets to specify the requirements and high level design of real-time systems and support the modeling of their dynamic behavior and response-time requirements. This requires the capability to express concurrency of tasks, task communication and synchronization, and, state dependencies in transaction processing [Kelly 87]. The following sections will present a detailed discussion of these design methods.

3.1 SREM

The *Requirements Statement Language* (RSL) permits concise, unambiguous specification of requirements for real-time software systems. The *Requirements Engineering Validation System* (REVS) processes and analyzes RSL statements; both RSL and REVS are components of the *Software Requirements Engineering Methodology* (SREM) [Bell 77] [Alford 77] [Davis 77], a software specification and design method for distributed real-time software systems. SREM describes a sequence of steps for defining requirements, expressing them in RSL and using REVS tools to check their consistency and completeness. Many of the concepts in RSL are based on PSL. For example, RSL has four primitive concepts: *elements* used to name objects; *attributes* used to describe the characteristics of elements; *relationships* used to describe binary relations between elements; and *Structures* used to describe the flow of control in a system. RSL models the stimulus-response nature of process-control systems; each flow originates with a stimulus and continues to the final response. This approach to specification makes explicit the sequences of processing steps required, and, provides for direct testability of the requirements [Bell 77].

Data and control flows in a system are specified in RSL as Petri-net based *requirements networks* (R_Nets). R_Nets have both graphical and textual representations [Alford 85] [Ramamoorthy 85]. R_Nets can be used to specify parallel (order-independent) operations using the AND node (&). Fan-in at the end of an AND structure is a synchronization point; all parallel operations must be completed before any subsequent processes are initiated. The OR node (+) has a condition associated with each operation. Each OR node must have an Otherwise operation that is executed if none of the conditions are true. If more than one condition is true, the first operation, as indicated by either implicit or explicit ordering, is executed. The third structure type is the FOR-EACH, which contains only one operation. The operation is executed once for each element in a specified set of system entities. Iteration over the set is not implied to be in any particular order.

A *Subnet* specifies the processing flow at a lower level in the hierarchy. A path of processing through an R_Net and its associated subnets is specified by attaching named nodes, called *validation points*, onto the R_Net graph and then defining the path in terms of a sequence of these validation points. A path can traverse multiple R_Nets by using the EVENT node (which enables another R_Net) as a connector. The path concept is used to verify the functional and performance requirements in the R_Net.

The *Requirements Engineering and validation System* (REVS) operates on RSL statements. REVS consists of three major components:

1. A translator for RSL.

2. A centralized database, the *Abstract System Semantic Model* (ASSM).

3. A set of automated tools for processing information in ASSM, and, for generating functional and analytical simulations to permit the evaluation of the performance characteristics of various system configurations.

ASSM is a relational database similar in concept to the PSL/PSA database. Automated tools for processing information in the ASSM include an interactive graphics package to aid in specifying flow paths, static checkers that check for completeness and consistency of the information throughout the system, and an automated simulation package that generates and executes simulation models of the system.

The first major extension of the SREM concepts sought to address the problem of defining system requirements and allocating them to the data processing subsystem. The result was called the *System Requirements Engineering Methodology* or SYSREM [Alford 85].

System requirements are specified in terms of time functions, their decompositions, and their allocation to system components [Alford 79]. A time function is specified in terms of its input messages, output messages, invariants of its transformation, completion criteria and performance. The *System Specification Language* (SSL) is used to describe the time functions. These functions are decomposed to the stimulus-response level by representing them in terms of subnets of RSL.

The SSL and RSL languages are thus interlocked and can be viewed as a single language with subsets applying to system and software requirements. Similarly, the methodology steps are interlocked: the end point of SYSREM provides all the information necessary to perform the steps of SREM. SYSREM provides a high-level description of a software system, while SREM helps describe the processing details of the functions specified in SYSREM.

Next, design languages and tools were developed and integrated into the SYSREM/SREM baseline. The result was the Distributed Computing Design System (DCDS): a software engineering environment for recording and controlling the development of requirements, design, code, and tests for a distributed data processing subsystem.

The DCDS design model separates the *distribution* phase from the *algorithm design* phase. Distributed design addresses the problem of selecting the data processor architecture and mapping the required processing onto this architecture to meet the timing and sizing requirements - in other words, it is *programming in the large.* Module design addresses the problem of constructing the algorithms that accomplish the required transformations and satisfy the overall accuracy requirements; module design is *programming in the small.*

The *Distributed Design Language* (DDL) is used to define the software processes, their allocation to processor nodes, and their relationships to the software requirements-level R_Nets, subnets, and data. The mapping of required processing and data onto modules and data structures is expressed in the *Module Design Language* (MDL); thus, RSL and MDL interlock. The DCDS tools are used to verify the completeness of the allocation and that the distributed design preserves the data flow and stimulus-response properties of the requirements. Simulation verifies satisfaction of response-time as well as processor and communication constraints.

The use of RSL and SSL in the requirements specification, and, DDL and MDL in the design and implementation of a distributed real-time software system in DCDS presents a major limitation in applying the method.

3.2 SARA

SARA is an acronym for *System ARchitects' Apprentice* [Estrin 86]. The name symbolizes an interactive design philosophy and a framework within which computer based tools can extend the capabilities of computer system designers and analysts in the domain of concurrent processing and distributed systems.

SARA supports the design of concurrent hardware and software systems, and, the building of explicit models of the environment in which a designed system is assumed to function. It permits the separation of structure and behavior in models, and, supports top-down structure partitioning (refinement) of a complex system into manageable pieces as well as bottom-up composition (abstraction) of reusable building blocks [Campos 77]. SARA emphasizes interfaces as places where inconsistencies are revealed and where information hiding is enforced. The *Structural Model* and *Behavioral Model* of SARA are detailed next:

- *Structural Model:* The primitives used to create fully nested, hierarchical structure models in SARA are *modules, sockets,* and *interconnections.* Named parent modules contain fully nested child modules. Sockets are associated with a module, and the name and services (i.e., behavior) associated with a socket are known both inside and outside the module. Interconnections provide binding between sockets.

  The highest level partition of the design universe results in two named modules: one to represent the system under design and the other to represent the environment of that system.

  The *Structure Language* (SL) describes the modules of the design and the *Module Interface Description* (MID) language describes the interconnections between these modules.

- *Behavioral Model:* The principal language for specifying the behavior of a system in SARA is the *Graph Model of Behavior* (GMB). This language allows a system designer or system analyst to build a functional model using three separate domains: control, data, and interpretation.

  - *Control Domain:* The GMB control graph primitives include *nodes,* which represent processing activity, and *control arcs,* which define the sequencing, or partial ordering, of node initiation events. A node can have many input and output control arcs. The GMB control graph, with appropriate restrictions, is equivalent to the Petri net model, and, supports similar reachability analysis methods.

    Logic expressions specify which control inputs are absorbed when a control node is initiated, and how control outputs are distributed when the node terminates. The dynamic behavior of the graph is characterized by the flow of tokens in the graph. A control node is initiated when there are sufficient tokens on its input arcs to satisfy its input logic. When a node terminates, tokens are distributed on its output arcs in some combination that satisfies its output logic. A token can represent information about the state of *active* system processes and/or *passive* system resources.

    The logical operator AND (*) can be used to model *fork* and *join* operations. The OR (+) input operator specifies that token(s) will be absorbed from either one of the input arcs

when the node initiates. The OR output operator specifies that token(s) will be placed on either one of the output arcs when the node terminates. The priority input operator (>) is used to specify a static order in which tokens are absorbed from alternative inputs. The deterministic-OR output operator (– – –) specifies that the choice among alternative outputs is determined directly by which of the inputs initiated the node. The inclusive-OR (+*) operator specifies that all tokens on the inclusive-OR arcs will be removed when the node is initiated.

Arrival of new inputs to a control node before previous inputs have been processed implies that the control graph is not safe. The GMB explicitly supports nonsafe models. A *serverType* attribute defines the capacity of a node to respond to control inputs.

- *Data Domain:* The GMB data graph primitives include *processors, datasets,* and *data arcs.* Processors are mapped one-to-one to control nodes in an associated control graph. Datasets represent data values, and data arcs define read/write access capabilities which processors have to datasets.

  Each processor may embody control flow decisions, processing delays, and/or data transformations as specified in the interpretation domain. The data domain only specifies which datasets, a processor may read or write.

  Datasets can be *simple,* in which case they hold only one copy of data satisfying the data type. They can also model data structures such as queues and stacks of data. When control and data are logically related or physically indistinguishable, dataset values will be conceptually paired with control tokens.

- *Interpretation Domain:* The GMB interpretation domain defines the data types of the various datasets and the algorithms which specify the behavior of the node-processor pairs in a model. Ideally, a language chosen for the interpretation domain should support formal verification of specified behavior, and should be translatable into programming languages used in the system being designed.

The *Control Flow Analyzer* (CFA) performs an exhaustive state-space analysis to determine if potential deadlocks or other errors exist in the modeled system. The basic analysis method used in the CFA centers on constructing a graph of all reachable states of a system (i.e., a reachability graph, or *computational flow graph*). A strong *reduction algorithm* reduces the size of the state-space which must be explored, yet allows most of the analysis properties to be determined. On the other hand, SARA is equipped with an interactive simulator that allows designers to perform experiments on the GMB model of a system. The GMB thus supports both performance analysis and correctness analysis of the same model.

The use of two orthogonal languages SL and MID in SARA can be substituted for by a distributed language. Recently, it has been shown that the requirements for the MID model tool are satisfied by Ada.

3.3 GRAFCET

A GRAFCET is a requirements specification model derived from Petri net. The formalism of the GRAFCET model is more suitable to model real-time systems because it overcomes two drawbacks inherent in Petri nets: non deterministic evolution and infinite creation of tokens [Boucher 84].

A GRAFCET (fig. 4) is composed of *steps* and *transitions* connected via *directed arcs* [AFCET 77]. Steps in a GRAFCET can only be active or inactive (there are never several tokens in one step). Moreover, transitions in a GRAFCET are always deterministic [Olive 83].

A step specifies a set of *actions*. It is in the active state when executing these actions. Otherwise, it is inactive. *Level actions* are executed for the period of time their step is activated. *Pulse actions* are executed for a fixed duration associated with them when their step is activated. The execution of an action might be conditional on input variables and/or the states of some specified steps. Tokens are used to show the active steps being carried out at a given time.

A transition has a Boolean condition associated with it, called its *receptivity*. The receptivity of a transition can be a function of input variables, the states of some specified steps, and the state of level variables or state variation of edge–triggered variables. Level variables are denoted by Y or $\overline{Y}$, positive-edge-triggered variables are denoted $Y\uparrow$ and, negative-edge-triggered variables are denoted by $Y\downarrow$. Time is included in a receptivity condition by specifying a time origin and duration. The origin is the time of the last activation of a preceding step. A receptivity condition can be used to implicitly assign priorities to step activations. When the receptivity of a transition is true, it can fire by deactivating its input steps and activating its output steps.

Additional concepts were added to the original definition of GRAFCET to suit the modeling of large systems [Carrière 83]:

- The *hierarchy* is a refinement concept. It allows to detail a step by means of a whole graph. It may have several initial/final steps

- The *subgraph* models the concept of mutual exclusion in accessing shared resources in concurrent systems. The access to a subgraph is protected by a semaphore.

The extended version of GRAFCET was implemented using the description language LEDA *(Language Evolué de Description d'Automatismes)* and the structured programming language ORACLE *(Outil de Représentation d'Algorithmes Complexes en LEDA)* [Carrière 83]. The resulting environment accepts a GRAFCET, LEDA or ORACLE description of a real-time software system, and, generates simulation programs and executable FORTRAN and assembly language code for the system.

A GRAFCET–based graphical language, FA–BASIC/C *(Factory Automation BASIC for Control)* has been implemented as an experimental language for specification of control sequences [Matsuzaki 85].

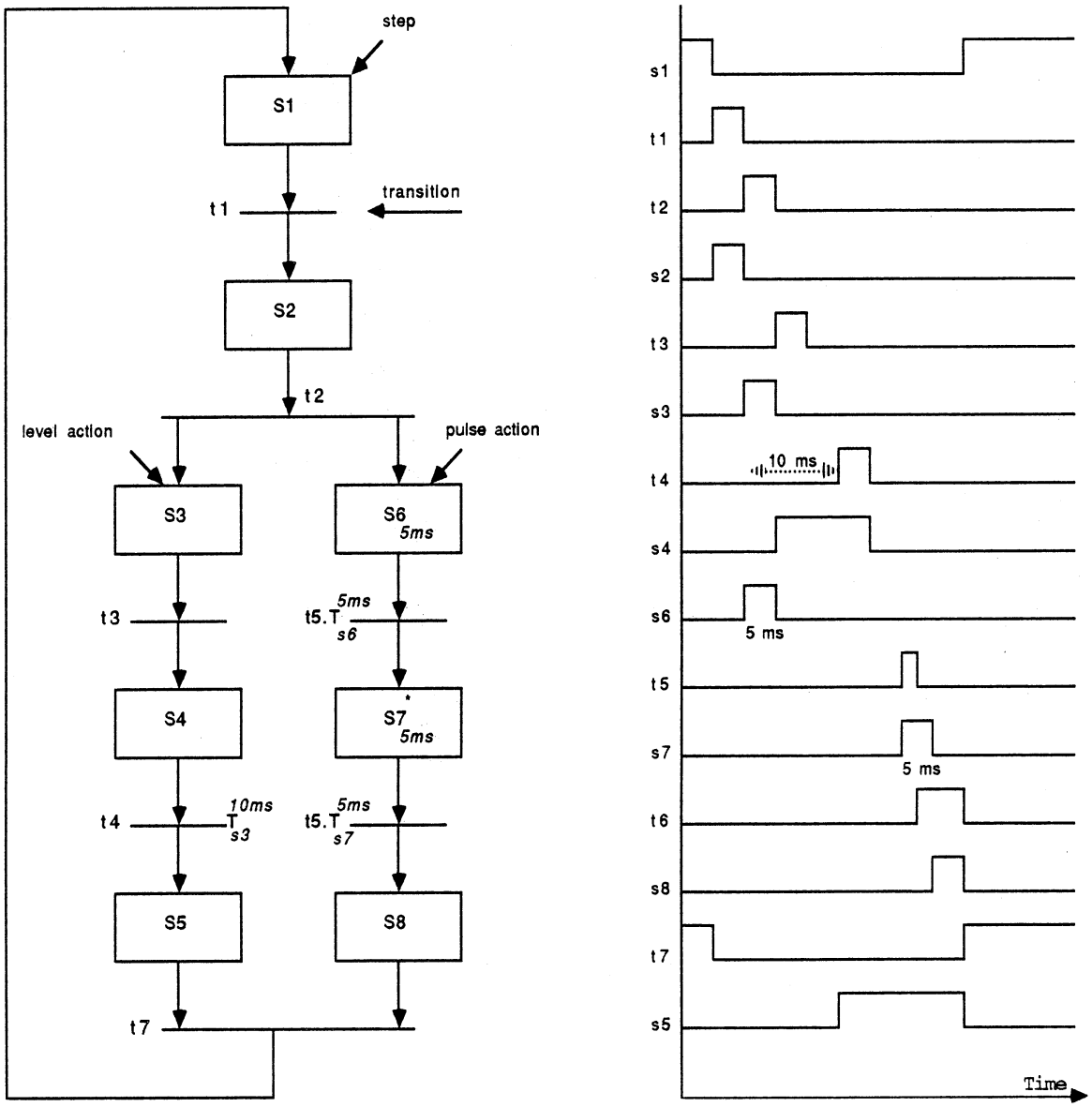


Figure 4: A GRAFCET model

3.4 MASCOT

The acronym MASCOT stands for *Modular Approach to Software Construction Operation and Test*. MASCOT is a language and machine-independent design tool, supported by a programming system. It can provide support through the design, construction and testing stages of the conventional software life cycle [Simpson 79] [Simpson 82].

MASCOT software consists of *subsystems* which run under the control of a *kernel*. The two main components of the *MASCOT machine* are the *activity-channel-pool (ACP) diagram* and the *programming system*. The (ACP) diagram (fig. 5) is used to express the structure of a subsystem, while the programming system refers to the construction tools and run-time support software used to realize the design encapsulated in an ACP diagram [Simpson 79].

A MASCOT subsystem consists of one or more *activities* which are connected by *intercommunication data areas* (IDAs). Each activity is essentially a process, and represents a single sequential thread of code concerned with the performance of one principal task. The *root procedure* is the program that determines the function of an activity. The formal parameters of a root procedure specify the intercommunication data areas and their types which will be required when it is used to support an activity.

IDAs fall into two broad categories, *channels* and *pools*. Channels are used exclusively for passing message data between activities. Pools form data buffers between activities, and are used for data storage. Conceptually, a channel has two unidirectional interfaces whereas a pool has one bi-directional interface. Access to pools and channels is generally by means of *access procedures*. On an ACP diagram, all connections between activities must be via an IDA.

Root procedures, channels and pools are known collectively as *system elements*, and are built up individually at compile, link or load time. They are combined together into subsystems by the use of the *form* facility.

In concept MASCOT assumes that its executive will have complete control of the system resources, i.e. that it is operating on a bare machine. The main resources provided by the run-time machine include those needed for process synchronization and for process execution. Synchronization and mutual exclusion of processes are provided by *control queues* and their primitives. Control queues are a form of combined semaphore/signal mechanism.

Four synchronizing primitives can operate upon the control queue, and their actions are defined as:

1. JOIN – if the control queue is not currently owned by any process then it is claimed by the caller, else the calling process is suspended until the owner of the control queue releases it. Where several processes are suspended waiting to complete the JOIN operation, the mechanism for selecting which one acquires the control queue is not defined and may be determined by the system implementors.

2. LEAVE – releases the control queue.

3. STIM – acts as a *signal* to the control queue, which has the ability to remember one STIM. If multiple STIMs are received in succession then they have no additional effects beyond those of the first one. Any process may STIM any control queue.

4. WAIT – may only be used with a control queue that has successfully JOINed the process. If the control queue is already STIMmed then the process will simply continue execution, clearing the STIM; if not, then it will be suspended until a STIM is received.

The JOIN-LEAVE operations on a control queue give an identical facility to P, V operations on a binary semaphore. Likewise the WAIT-STIM combination is similar to wait, signal operations on a condition variable. The coordination of JOIN-LEAVE-WAIT-STIM into a unified set of operations on a single type of control variable produces a natural and logical solution to the problem of intercommunication between loosely coupled processes. Therefore, MASCOT supports asynchronous interaction between cooperating parallel processes.

The MASCOT executive is also expected to provide certain other defined run-time support primitives including a DELAY facility, where a clock is available, and certain primitives concerned with initializing or suspending processes.

MASCOT provides an option for dealing explicitly with interrupts by defining a new type of parameterless procedure: the *response procedure*. The response procedure is associated with a control queue, and is called immediately without the intervention of the system scheduler whenever that queue is STIMmed.

One of the main purposes of MASCOT was to aid the reliable design of concurrent systems. The design phase is provided for by the MASCOT ACP diagram which allows the representation of the modular breakdown, without explicitly including the concurrency requirements in the diagram itself. MASCOT and Modula–2 have been used for the construction of *embedded software* [Budgen 85].

MASCOT is well suited for real–time systems since it deals specifically with structuring a system into tasks and defining the interfaces between them. However, it starts with a network diagram of tasks and addresses neither the issue of how to structure a system into tasks nor the structure of the individual tasks themselves [Gomaa 84]. MASCOT ACP diagrams can be used to express the structure of a system implemented in Ada [Dibble 82]; an *activity* maps onto a task (type) with no entries, and, a *channel/pool* maps either onto a task (type) with task entries representing the access mechanisms, or, onto (generic) packages with procedures representing the access mechanisms. On the other hand, MASCOT is more fit for modeling concurrent systems, and does not incorporate any of the timing constructs essential to modeling distributed real-time embedded systems.

3.5 DARTS

Design Approach for Real Time Systems (DARTS) is an extension to the *Structured Analysis/Structured Design* (SA/SD) method. It provides an approach for structuring a system into tasks and a mechanism for defining the interfaces between tasks.

The data-flow approach to software design is particularly appropriate for the design of real-time systems because the data in these systems may be considered to flow from input to output and in between to be transformed by software tasks. Because real-time systems are usually data-flow oriented, the DARTS method starts with a data flow analysis of the system [Gomaa 84].

The transforms in the data flow diagrams are analyzed next to identify which may run concurrently and which are sequential in nature. By this means, tasks are identified: one transform may correspond to one task, or one task may encompass several transforms.

The criteria for deciding whether a transform should be a separate task or grouped with other transforms into one task are the following:

- *Dependency on I/O.* A transform constrained to run at a speed dictated by the I/O device with which it is interacting needs to be a separate task.

- *Time-critical functions.* A time-critical function needs to run as a separate high-priority task.

- *Computational requirements.* A computationally intensive function (or set of functions) can run as a lower priority task consuming spare CPU cycles.

- *Functional Cohesion.* Transforms that perform a set of closely related functions can be grouped together into a task. Since the data traffic between these functions may be high, having them as separate tasks will increase system overhead, whereas implementing each function as a separate module within the same task ensures functional cohesion both at the module and task levels.

- *Temporal cohesion.* Certain transforms perform functions that are carried out at the same time. These functions may be grouped into a task so that they are executed each time the task receives a stimulus. Each function should be implemented as a separate module to achieve functional cohesion at the module level. These modules in turn are grouped into the task thereby achieving temporal cohesion at the task level.

- *Periodic execution.* A transform that needs to be executed periodically can be structured as a separate task that is activated at regular intervals.

DARTS can be classified as a top-down design method that starts with a system specification and results in a specific number of tasks by applying the above heuristics for task structuring. A constraint imposed on the resulting number of tasks might lead to the violation of one or more structuring rule.

DARTS, like SA, makes no distinction between data and control flow on the data flow diagrams. When using DARTS, there is sometimes a temptation to make each transform on the system data flow diagram a task. This usually leads to too many tasks and to unnecessary complexity in dealing with the accompanying communication and synchronization issues resulting in poor performance of the system. On the other hand, DARTS tasks can be easily mapped into Ada task types and Ada tasks.

3.6 PROT nets

Classical Petri nets have been extended to fulfill the issues of specification and simulation and support the rapid prototyping of a system [Bruno 86a]. The resulting high level Petri nets are known as *process translatable* (PROT) nets because their associated semantics allow the implementation of processes and of their synchronizations to be generated from the net automatically.

A PROT net is a strongly connected directed graph with two kinds of vertices – places and transitions – connected by arcs in such a way that no two places and no two transitions can be linked directly. Tokens circulate in the net according to the firing rules associated with transitions. The features of a PROT net are [Bruno 86a] [Bruno 87]:

- Types: A PROT net models the interactions among the instances of different process types. Such instances synchronize and exchange information at transitions. In order to distinguish instances of the same process type and to support data exchange among the instances of different process types, data structures are associated with process types through type declaration.

- *Places*: Places are assigned types because they represent the states of the corresponding process types. Each process type consists of the ordered collection of all the places of the corresponding type. Such a collection is, in general, a tree called the *process tree;* however, it is a simple sequence if there are no fork places. The root of this tree corresponds to the initial state of the process and a path from the root to a leaf represents the execution of a single cycle of the process. A cycle consists of the ordered collection of the successive states assumed by an executing process, the initial and final states being identical.

- *Tokens*: Tokens represent process instances. The presence of a token in a place indicates that the corresponding process is in the state identified by this place. Tokens carry pieces of information – called *attributes* – whose structure is specified in the type declaration. Tokens can be given priorities.

- *Transitions*: Transitions perform synchronizations among process instances at their ingoing arcs; data exchange can also be specified by means of actions inscribed on outgoing arcs. A transition is called simple if it is not decomposable into a PROT subnet. PROT subnets enable top-down structuring and information hiding. Predicates can be inscribed on transitions: they are Boolean expressions to be evaluated on the attributes of the tokens present in the input places of the transitions. In case of ambiguity, the dot notation can be used to select a particular attribute starting from a place name.

  Transitions having some input places in common are said to be mutually exclusive because the firing of one of them disables the others.

  Additional pieces of information can be associated with transitions such as timing requirements and the firing priority in the case of mutually exclusive transitions.

  Timed transitions have a delay associated with them, non-timed transitions are called *immediate transitions.*

- *Arcs*: Arcs connect transitions to places and places to transitions. Actions can be inscribed on outgoing arcs of transitions. Basically, actions are assignments that can change the value of the attributes carried by the tokens which caused the transition to fire.

- *Initial Marking*: This is the initial distribution of tokens into places. All tokens of the same type must be put into a single place. The initial marking specifies the initial values of the attributes of each token as well.

- *Transition Firing*: A transition firing takes place when there is at least one token in each of its input places and the predicate inscribed on it is satisfied. The firing of an immediate transition consists of removing the tokens satisfying the predicate from the input places, and adding tokens to the output places after performing the actions inscribed on the outgoing arcs.

  The firing of a timed transition consists in removing the tokens satisfying the predicate from the input places, and after waiting the associated delay time, adding tokens to the output places after performing the actions inscribed on the outgoing arcs.

The dynamics of each object in a system can be modeled adequately by a PROT net. Objects in a system communicate with other objects by sharing places of the same type, called *communication places*. Only the input places (places with no entering arcs) of a PROT net can be shared with the output places (places with no exiting arcs) of another PROT net to become communication places.

PROT nets can be translated into Ada program structures in order to obtain a rapid prototype of the system [Bruno 86a]. The translation process involves two steps: the identification of process types and the implementation of synchronizations according to the logic expressed in the net.

Each process type consists of an ordered collection of states to be extracted from the net starting from its initial place. The states of a process type are scattered in the net and can be gathered in a process tree. Each process tree can be translated into an Ada task type whose body consists of states alternated with pieces of code implementing interactions with transitions.

Transitions carry out synchronization and communication among processes. When a transition fires, control is passed from ingoing process states to outgoing ones. With respect to a transition, a process can be in one of the following synchronization conditions:

- *Property Synchronized:* When the transition has both an input place and an output place belonging to this process type; in such a case the corresponding states of the process are to be synchronized with the states of other processes.

- *Suspended:* When the transition has only an input place of the given process type; the firing of the transition interrupts the execution of the process.

- *Resumed:* When the transition has only an output place of the given process type; the firing of the transition resumes the execution of the process.

Transitions are implemented as Ada tasks which interact with processes through two rendezvous; this occurs because the presence of a predicate in a transition prevents it from handling processes in a first-come-first-served (FIFO) manner.

PROT nets form the basis of an object-oriented approach for specifying and designing distributed systems [Bruno 86b]. First, each object is modeled focusing on its control structure using PROT nets. Next, the net is augmented with the data structures needed for the object to exchange information with its environment. Finally, all the objects are connected to make up the overall system.

The PROT net approach decouples the structure of a system from its behavior. The tokens model the atomic entities of a system together with their attributes. PROT nets provide an excellent model of the behavioral aspects of a system, while partially modeling its structural aspects [Bruno 87].

3.7 The Transformation Schema

The *Transformation Schema* defines an extension of the data flow diagram that has a generalized capability to represent the control and timing aspects of systems [Ward 85] [Ward 86]. In this extended data flow diagram (fig. 6):

- *Data transformations* model portions of systems that do one or more of: transform input data to output data; modify or create stored data based on input data; transform previously stored data into output data; report the occurrence of events. In other words, a data transformation is an abstraction on a low-level system function.
- *Control transformations* control the behavior of data transformations by activating and deactivating them. In other words, a control transformation is an abstraction on some portion of a system's control logic.
- *Discrete data flows* are associated with a set of variable values that is defined at discrete points in time. The discrete data flow is an abstraction on a transaction or other data aggregate sent or received as a unit by a system.
- *Continuous data flows* are associated with a set of variable values defined continuously over a time interval. A continuous data flow is an abstraction on a real-world quantity such as a temperature or pressure monitored by a system, or on a continuously variable control output produced by a system.
- *Event flows* report a happening or give a command at a discrete point in time. An event flow is an abstraction on an interrupt or other pulse-type message within a system. Event flows can be classified into *signals, activations* and *deactivations.*
- A *signal* shows the sender's intention to report that something has happened, and the absence of any knowledge on the sender's part of the use to which the signal is put. *Activations* show the sender's intention to cause a receiver to produce some output. *Deactivations* show the sender's intention to prevent a receiver from producing some output.

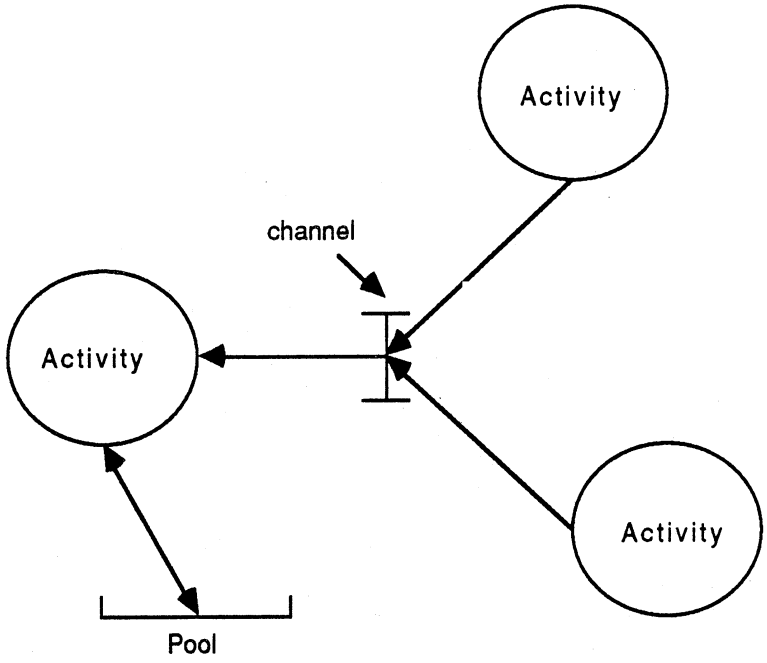


Figure 5: The MASCOT ACP model

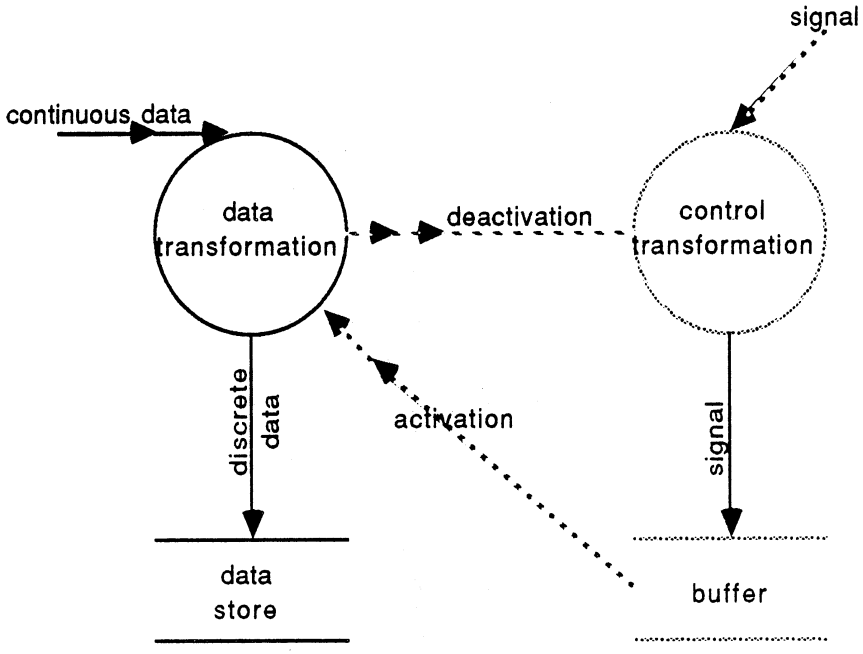


Figure 6: The transformation schema model

- *Data stores* act as a repository for data that is subject to storage delay, that is, data whose values are modified at discrete points in time and remembered over the intervening time intervals. A data store is an abstraction of a file.

- *Buffers* form a special type of store in which flows produced by one or more transformations are subject to a storage delay before being consumed by one or more transformations. The buffer is an abstraction of a stack or queue.

- *State Machines* form a representation for a Mealy-type finite automaton with output. The automaton describes the logic of a control transformation, and summarizes the dynamics of a portion of the schema.

A transformation schema is built from the above elements using a set of transformation rules. Hierarchies are used in the transformation schema to avoid any explosion in the number of transformations in the diagram. Execution of the schema is visualized in terms of the placement of *tokens*. Generally speaking, the presence of a token indicates actual or potential activity. A set of execution rules governs the displacement of tokens in a transformation schema.

Tokens may be placed on transformations, flows, and buffers. Data stores are used within a schema to indicate storage delays among transformations; data stores do not enter directly into the execution rules and tokens are not placed on them.

The transformation schema provides for an adequate decoupling of the data flow aspect from the control flow aspect in a system. The behavior of the system is more adequately described than its structure.

3.8 PAISLey

PAISLey (Process-oriented, Applicative, and Interpretable Specification Language) is a specification language for describing digital systems. It was formed by merging two successful models of digital computation, asynchronous processes and functional programming [Zave 86].

A system is described in PAISLey as a set of asynchronous processes, where each process has a state and goes through a never-ending sequence of discrete state changes. The set of possible states of a process can be described using *set expressions*. Set expressions are formed using set names and enumerations of values, plus the union and Cartesian-product operators. The union operator provides for *generalization hierarchies* among sets; the resulting set is composed of all the members of the individual sets. The Cartesian-product operator provides for *component* or *aggregation hierarchies* among data items; each member of the resulting set is an ordered n-tuple of elements from the constituent sets.

In a process, the interval of time between successive state changes is called a *process step*. Computations occurring during a process step are specified using a simple functional notation. Functional notations view a computation as a mapping from an input value (or argument) to an output value. The

values can be structures of arbitrary complexity, and mappings are defined using expressions without variables or assignment statements.

In PAISLey, mapping expressions are formed using the names of other mappings, formal-parameter names, values, and three *combining forms*. The combining forms are *composition, conditional selection* and *tuple-formation*.

Each process is defined by a mapping called its *successor mapping*. Each step of the process consists of one evaluation of the successor mapping; the current state is the argument given to the successor mapping and the next state is the value produced by the successor mapping (at the end of the process step, the current state is replaced by the next state).

Exchange functions provide a mechanism for interprocess interactions. They complete the merger between asynchronous processes and functional programming. Exchange functions can perform any kind of interprocess interaction and yet behave locally like mapping evaluations.

Each exchange function has *type* and *channel* attributes. Once the evaluation of an exchange function has been initiated, if there is another exchange-function evaluation pending with the same channel attribute and a compatible type attribute, then the two evaluations interact by *exchanging* their arguments. Each evaluation terminates, returning as its value the argument of the other. From the view point of the process in which an exchange function is evaluated, it behaves like any other mapping. From the global viewpoint, however, exchange functions specify point-to-point, two-way, mutually synchronized interprocess communication.

A *timing constraint* can be attached to any mapping in a PAISLey specification. The evaluation time of the mapping is viewed as a random variable; a combination of an upper bound, lower bound or distribution may be given. Both functional properties and timing properties of the specification are checked for consistency.

The PAISLey language is supported by a set of software tools forming an environment for analyzing and executing specifications. It should be noted that specifications in PAISLey are *operational* (they are implementation-independent models of *how* to solve a problem), rather than *nonconstructive* (statements of *what* properties a solution should have).

3.9 DAPSE

Ada is a high level programming language developed at the initiative of the United States Department of defense [DoD 83] for use in the embedded systems application area. Ada is not merely a programming language; it is a vehicle for new software practices and methods for specification, program structuring, development and maintenance [DSB 87]. The Ada language coupled with its Programming Support Environment *(APSE)* support the development of software methodologies for embedded software systems [DoD 82]. Three levels of programming support environment for Ada were defined:

- *KAPSE.* A kernel APSE supporting basic functions of operating systems, database, and communication support. This kernel can provide tool portability from one computer system to another.

- *MAPSE.* A Minimal APSE providing a methodology-independent set of essential tools to support the Ada programmer. These tools include language editors, translators, configuration management, loading, linking, and static and dynamic analysis tools.

- *APSE.* Fuller support for development of Ada programs, including tools for requirements, specification, design and management. An APSE offers a coordinated and complete set of tools which is applicable at all stages of the system life-cycle [Dowson 86].

A Distributed Ada Programming Support Environment (DAPSE) is in its initial development stages [Marcus 86]. The DAPSE project has two main goals: First, to build an environment and an initial set of tools to support the programming of distributed Ada applications, and second, to build this environment and its tools with a particular design methodology. This methodology consists of first describing large parts of the environment, as well as the tools in that environment, using a high-level specification language, and then utilizing generators to process these specifications and automatically produce components of the environment and tools.

4 Towards a real-time design methodology

Real-time systems can be classified as *time-relevant* (time sharing, general computing), *soft real-time* (missing deadline is not fatal) and *hard real-time* (missing deadline is fatal).

Design methodologies for time-relevant and soft real-time systems have been proposed and evaluated based on the principles of software engineering. Distributed real-time embedded systems are classified as hard real-time systems. Recent attempts to deal with the various aspects of design and analysis of these systems were reported. However, a complete methodology targeted for designing distributed real-time embedded systems is needed.

A distributed real-time embedded software system is a system for monitoring and controlling a set of time dependent physical processes. For these systems, it is not sufficient for the software to be logically correct, i.e., implementing the intended algorithms. The system should satisfy a set of requirements that can be classified into *concurrency, timing, reliability* and *hardware* requirements. Separate attempts to model some of these requirements have been reported. These attempts are all based on modified Petri net models, and, are detailed below. A methodology for designing distributed real-time embedded systems could extend these models to cover all the unique aspects of the above systems, and, use the expressiveness offered by distributed languages in implementing the resulting model.

4.1 Concurrency Requirements

The loosely-coupled architecture of distributed real-time embedded systems allows concurrency in the execution of their control software. The representation of these systems can be based on a modified form of Petri nets [Yau 83] designed to represent both their control and data flows.

The modified Petri net consists of a set of control state variables, a set of abstract data types, a set of data objects, and a set of software components which are connected to each other through the control state variables and through the data objects. Software components are externally described in terms of their input and output control states, associated data types and data objects, and a set of control and data transfer specifications. Interconnection of software components is defined through shared control states and through shared data objects. A system component can be viewed internally as a collection of partially ordered subcomponents, local control states, local data types, and local data objects.

The control state variables correspond to the places of a Petri net. Software components correspond to the non-primitive transitions of a Petri net, where a non-primitive transition has a Petri net subgraph as its inner structure and does not fire instantaneously. The execution of a software component can be regarded as the firing of a non-primitive transition. The non-primitive transition firing rule, which is fixed in ordinary Petri nets, is generalized in modified Petri nets by associating with each software component a control transfer specification, which gives the control flow(s) through that component. Each component has associated with it a data transfer specification which represents the data flow(s) through that component.

4.2 Timing Requirements

The timing requirements of a real-time system can broadly be classified under the major categories: *deadline, throughput,* and *periodic processing* requirements.

- *Deadline* requirements dictate that a system be in a given state before some real timing event occurs. This requires some form of synchronization with the given event.
- *Throughput* requirements dictate that the overall system make progress at some given *rate*. This state is usually measured and specified in various different ways: number of outputs generated per unit time, the ratio of the number of outputs to the number of inputs if the number of inputs determine a well defined measure of time.
- *Periodic* requirements dictate that a part or whole of system processing be repeated either indefinitely or a certain number of times at some specified rate.

Timing requirements are modeled using *timed Petri nets*.. Timed Petri net modeling involves using a time extended version of Petri net theory to represent a software system. They allow the incorporation of statistical and deterministic timing information into real-time embedded systems analysis [Leveson 87].

A timed Petri net is a Petri net composed of a set of places, a set of transitions, an input function, an output function, and an initial marking along with the added *minimum firing time* and *maximum firing time* functions associated with the transitions in the net. The firing time functions specify the conditions under which a transition may fire. Once enabled, a transition may fire within the time interval specified by the above functions. If the maximum firing time is exceeded, the transition must fire immediately. The interval limits are absolute time units specified in reference with the software system initialization time.

Added complexity over the untimed Petri net arises because of the continuous nature of time. The timed Petri net is equivalent to a standard Petri net if the minimum firing time function is 0 for the transitions in the net and the maximum firing time function is set to ∞ for all the transitions in the same net.

Priorities can be associated with transitions in a timed Petri net using the minimum and maximum firing time functions. These functions will enforce timing constraints on enabled transitions by setting the maximum firing time of the transition with higher priority less than the minimum firing time of the transition with the lower priority.

4.3 Reliability Requirements

Control usually can not be abandoned abruptly in an embedded system. Therefore, response to hardware failures, software faults, human errors, and undesired and perhaps unexpected environmental conditions must be built into the system. These responses can take three basic forms:

- a *fault-tolerant* system continues to provide full performance and functional capabilities in the presence of operational faults.
- a *fail-soft* system continues operation but provides only a degraded performance or reduced functional capabilities until the fault is removed.
- a *fail-safe* system attempts to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

In general, from a safety standpoint, the first priority of the response to a safety-critical situation is reduction of risk rather than attainment of mission.

While software itself cannot be unsafe, it can issue commands to a system it controls which place the system in an unsafe state. Furthermore, the controlling software should be able to detect when factors beyond the control of the computer (e.g. environmental conditions) place the system in a hazardous state and to take steps to eliminate the hazard or, if that is not possible, initiate procedures to minimize the hazard; a *hazard* is a set of conditions within a state from which there is a path to an unplanned event (a hazardous state).

The timed Petri net of a system can be analyzed to determine the set of hazardous states the system may reach where a state corresponds to a net marking [Leveson 87]. Next, the system designer ensures

that these states will never be reached by assigning a higher priority to the transition leading to a safe state, and inhibiting the firing of the transition leading to the hazardous state. Another way to ensure that the proper transition is always fired when both transitions are enabled is to enforce timing constraints or timing conditions in the designed system: the maximum time that it may take for the desired transition to fire must be less than the minimum time for the other transition to become enabled and to fire. Each of these time quantities must be the total time that the enabling conditions have been met, not just the individual transition time limit.

Once the design is determined to be safe, run-time faults and failures must be considered. A failure is defined as an event while a fault is a state. A failure always results in a fault and is called a fault-starting event. The type of fault which results from the failure must be included in the model in order to analyze the consequences of failures on the system. Thus, a new type of transitions is introduced; a *failure transition* which is denoted by a double bar and a *fault condition* which is denoted by a double circle. To make analysis practical, a place which acts as a counter can be added to the failure transition. The number of tokens initially contained in this place controls the maximum number of times the transition (failure) can fire. Adding these features coupled with Petri net analysis techniques with respect to the definitions of fault tolerant, recoverable, and fail-safe design will aid the designer in adding appropriate failure detection and recovery techniques to the system. Timing information is used to set a limit on the delay needed to resume normal course of action by the system. On the other hand, graceful degradation can be achieved by assigning a lower priority to an alternative path in the system, and specifying a time-out condition on the execution of this path in case the normal path of execution is not functional [Leveson 87].

Fault conditions and failure transitions can be used to model exception-handling in the system. On the other hand, partial fault-tolerance can be achieved by duplicating the critical places of a Petri net, while full fault-tolerance can be achieved by duplicating all the places in the net. The degree of duplication of a place is stored in a counter associated with the place.

4.4 Hardware Requirements

The hardware constraints in a distributed real-time embedded system are stressed because the host computer used in developing the software system is usually different from the target computer where the software should eventually be installed and running. The topology of both the host and the target computer networks and the processing limits of each node in the network are essential in designing an allocation scheme for both code and data in the system.

The hardware requirements of a distributed real-time embedded system involves a description of the topology of the target sytem; the node interconnections together with the communication bandwidth of their data transmission links, as well as, the computational speed and storage capacity of each node are provided. This knowledge is used by a set of partitioning and allocation algorithms to derive a distribution scheme for the modules of the control software that is both functionally correct and satisfying the timing and storage constraints imposed on the operation of the system. The distribution scheme need not be unique and will be affected by the structure of fault-tolerant systems: the number

of duplicated modules in these systems as well as the degree of duplication of these modules. A Petri net model could be used to represent these alternatives, the scheme adopted by the current system configuration, and, any change in the system configuration dictated by hardware faults and exceptions.

5 Conclusion

This report presents a detailed discussion of the major software design methods in the field of Software Engineering. The need for these methods was dictated by the continuously growing complexity of computer systems and their application domains. These software design methods are managed by a standard life-cycle model, the waterfall model, composed of the requirements specification, design, implementation, system testing and system evolution phases respectively. The report stresses the need for an improved life-cycle for managing the design of distributed real-time embedded systems. This life-cycle will be based on system prototyping and simulation; prototyping will help refine the requirements specification of a system into a more consistent and complete version, while simulation will help verify the performance of a designed system.

Currently, a large set of techniques for designing general-purpose centralized and distributed systems is available. The design philosophy and the specific steps followed by each method are application-dependent. The database-oriented techniques model the statics of a system, while the real-time methods model its dynamics. Both aspects should be modeled in distributed real-time embedded systems since these systems control their environment through a continuous flow of control and data between the system and the environment. A methodology for designing the above class of systems should model both their structural and behavioral aspects.

Distributed real-time embedded systems are characterized by a set of concurrency, timing, reliability and hardware requirements. The Ada language was designed with the intent of dealing with these special requirements, and should be adapted for implementing the systems designed by the above methodology. Ada, on the other hand, improves the readability and usability of software components by providing a set of constructs that encourage the practice of the Software Engineering principles of program modularity and information hiding.

References

[AFCET 77] Group de travail Systèmes logiques de l'AFCET, "Pour une représentation normalisée du cahier des charges d'un automatisme logique," Automatique et Informatique Industrielles, nº 61/62, Nov./Dec. 1977.

[Agerwala 79] T. Agerwala, "Special Feature: Putting Petri Nets to Work," Computer, Dec. 1979, pp. 85–94.

[Alford 77] M. W. Alford, "A Requirements Engineering Methodology for Real–Time Processing Requirements," IEEE Transactions on Software Engineering, vol. SE–3, nbr. 1, Jan. 1977, pp. 60–69.

[Alford 79] M. W. Alford, "Requirements for Distributed Data Processing Design," Proceedings of the First International Conference on Distributed Computing Systems, Oct. 1979, pp. 1–14.

[Alford 85] M. W. Alford, "SREM at the Age of Eight; The Distributed Computing Design System," Computer, April 1985, pp. 36–46.

[Aschim 82] F. Aschim, and B. M. Mostue, "IFIP WG 8.1 Case Solved Using SYSDOC and SYSTEMATOR," Information Systems Design Methodologies: A Comparative Review, T. W. Olle, H. G. Sol, and A. A. Verrijn–Stuart (eds.), IFIP, 1982, pp. 15–40.

[Auernheimer 86] B. Auernheimer, and R. A. Kemmerer, "RT–ASLAN: A Specification Language for Real–Time Systems," IEEE Transactions on Software Engineering, vol SE–12, nbr. 9, Sept. 1986, pp. 879–889.

[Avrunin 85] G. S. Avrunin, and J. C. Wileden, "Describing and Analyzing Distributed Software System Design," ACM Transactions on Programming Languages and Systems, vol. 7, nbr. 3, July 1985, pp. 380–403.

[Avrunin 86] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle, "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 278–292.

[Bell 77] T. E. Bell, D. C. Bixler, and M. E. Dyer,"An Extendable Approach to Computer– Aided Software Requirements Engineering," IEEE Transactions on Software Engineering, vol. SE–3, nbr. 1, Jan. 1977, pp. 49–60.

[Bergland 81] G. D. Bergland, and R. D. Gordon, Eds., *Tutorial: Software Design Strategies,* second edition, IEEE Computer Society, Washington D.C. 1981.

[Booch 86] G. Booch, "Object-Oriented Development," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 211–221.

[Boucher 84] D. Boucher, M. Poise, and A. Puissochet, "GRAFCET as a Description and Simulation Tool at the Functional Level in CAD System," 1984 IEEE International Symposium on Circuits and Systems, ISCAS '84, pp. 324–327.

[Bruno 86a] G. Bruno, and G. Marchetto, "Process–Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 346–357.

[Bruno 86b] G. Bruno, and A. Balsamo, "Petri Net–Based Object–Oriented Modeling of Distributed Systems," OOPSLA'86: Object–Oriented Programming Systems, Languages and Applications Conference Proceedings, Sept. 1986, pp. 284–293.

[Bruno 87] G. Bruno, and M. Morisio, "Petri-Net Based Simulation of Manufacturing Cells," 1987 IEEE International Conference on Robotics and Automation, March 1987, pp. 1174–1179.

[Budgen 85] D. Budgen, "Combining MASCOT with Modula–2 to aid the Engineering of Real–time Systems," Software-Practice and Experience, vol. 15, August 1985, pp. 767–793.

[Cameron 82] J. R. Cameron, "Two pairs of Examples in the Jackson Approach to System Development," Proceedings of the 15^{th} Hawaii International Conference on System Sciences, Jan. 1982.

[Cameron 86] J. R. Cameron, "An Overview of JSD," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 222–240.

[Campos 77] I. M. Campos, and G. Estrin, "Concurrent Software System Design Supported by SARA at the Age of One," Proceedings of the 3^{rd} International Conference on Software Engineering, 1977, pp. 230–242.

[Carrière 83] B. Carrière, C. Cazalot, J. M. Dumas, P. M. Grojean, P. Leroy and F. Prunet, "A CAD System for Process Control Based Upon a Standard," Information Processing 83, R. E. A. Mason (ed.), IFIP, 1983, pp. 507–512.

[Dasarathy 85] B. Dasarathy, "Timing Constraints of Real–Time Systems: Constructs for Expressing Them, Methods of Validating Them," IEEE Transactions on Software Engineering, vol. SE–11, nbr. 1, Jan. 1985, pp. 80–86.

[Davis 77] C. G. Davis, and C. R. Vick, "The Software Development System," IEEE Transactions on Software Engineering, vol. SE–3, nbr. 1, Jan. 1977, pp. 69–84.

[Davis 82] A. M. Davis, "The Design of a Family of Application–Oriented Requirements Languages," Computer, May 1982, pp. 21–28.

[Dibble 82] R. Dibble, "Software Design and Development using MASCOT," Software for Avionics, AGARD Conference Reprint nbr. 330, 1982, pp. 19_1–19_15.

[DoD 82] "Ada Methodologies: Concepts and requirements," Ada Joint Program Office, The United States Department of Defense, Nov. 1982.

[DoD 83] "Reference Manual for the Ada Programming Language," ANSI/MIL–STD–1815A–1983, The United States Department of Defense, Feb. 1983.

[Dowson 86] M. Dowson, "ISTAR – An Integrated Project Support Environment," Proceedings of the 2^{nd} SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, Dec. 1986, pp. 27–33.

[DSB 87] "Report of the Defense Science Board Task Force on Military Software," The United States Department of Defense, Sept. 1987.

[Du Plessis 86] A. L. Du Plessis, "A Software Engineering Environment for Real– Time Systems," Ph.D. Dissertation, University of South Africa, June 1986.

[Dubois 86] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, and A. Rifaut, "A Knowledge Representation Language for Requirements Engineering," Proceedings of the IEEE, vol 74, nbr. 10, Oct. 1986, pp. 1431–1444.

[Estrin 86] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 293–311.

[Fairley 85] R. E. Fairley, *Software Engineering Concepts,* New York: McGraw–Hill, 1985.

[Freeman 83] P. Freeman, and A. I. Wasserman, Eds., *Tutorial on Software Design Techniques,* fourth edition, IEEE Computer Society, Washington D.C. 1983.

[Freeman 87] P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," IEEE Transactions on Software Engineering, vol. SE–13, nbr. 7, July 1987, pp. 830–844.

[Gane 80] C. P. Gane, "Data Design in Structured Systems Analysis," Infotech State of the Art Report on Data Design, 1980.

[Gomaa 84] H. Gomaa, "A Software Design Method for Real–Time Systems," Communications of the ACM, vol. 27, nbr. 9, Sept. 1984, pp. 938–949.

[Greenspan 82] S. J. Greenspan, J. Mylopoulos, and A. Borgida, "Capturing More World Knowledge in the Requirements Specification," Proceedings of the Sixth International Conference on Software Engineering, 1982, pp. 225–234.

[Greenspan 86] S. J. Greenspan, A. Borgida, and J. Mylopoulos, "A Requirements Modeling Language and its Logic," Information Systems, vol. 11, nbr. 11, 1986, pp. 9–23.

[IEEE 83] *IEEE Standard Glossary of Software Engineering Terminology,* IEEE standard 729-1983.

[Jackson 76] M. A. Jackson, "Constructive Methods of Program Design," Proceedings of the First European Cooperation in Informatics, vol. 44, 1976, pp. 236–262.

[Kelly 87] J. C. Kelly, "A Comparison of Four Design Methods for Real–Time Systems," Proceedings of the 9^{th} International Conference on Software Engineering, May 1987, pp. 238–252.

[Knuth 82] E. Knuth, F. Halász, and P. Radó, "SDLA: System Descriptor and Logical Analyzer," Information Systems Design Methodologies: A Comparative Review, T. W. Olle, H. G. Sol, and A. A. Verrijn–Stuart (eds.), IFIP, 1982, pp. 143–171.

[Lauer 80] P. E. Lauer, and M. W. Shields, "COSY: An Environment for Development and Analysis of Concurrent and Distributed Systems," Proceedings of the Symposium on Software Engineering Environment, June 1980.

[Leveson 87] N. G. Leveson, and J. L. Stolzy, "Safety Analysis Using Petri Nets," IEEE Transactions on Software Engineering, vol. SE–13, nbr. 3, March 1987, pp. 386–397.

[Marcus 86] M. Marcus, K. Sattley, S. C. Schaffner, and E. Albert, "DAPSE: A Distributed Ada Programming Support Environment," Proceedings of the IEEE 2^{nd} International Conference on Ada Applications and Environments, 1986, pp. 115–125.

[Masiero 86] P. C. Masiero, T. Chase, S. Padmanabhan, and D. Teichroew, "Implementation of Structured Methodologies for Real–Time Systems using the System Encyclopedia Manager System," Presented at the National Conference and Workshop on Methodologies and Tools for Real–Time Systems, Washington, D.C., March 10–11, 1986.

[Matsuzaki 85] K. Matsuzaki, S. Hata, J. Hamano, Y. Kurashima, and M. Torii, "Petri-Net Structured Sequence-Control Language with GRAFCET-Like Graphical Expression for Programmable Controllers," 1985 International Conference on Industrial Electronics, Control and Instrumentation, IECON '85, pp. 433–438.

[Olive 83] V. Olive and D. Rouquier, "A Systematic Method for the Synthesis of Control Parts Defined by GRAFCET," Information Processing 83, R. E. A. Mason (ed.), IFIP, 1983, pp. 35–40.

[Orr 78] K. T. Orr, "Introducing Structured Systems Design," Structured Analysis and Design, vol. *II*, 1978, pp. 3–23.

[Peterson 81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice–Hall, 1981.

[Ramamoorthy 84] C. V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda, "Software Engineering: Problems and Perspectives," Computer, Oct. 1984, pp. 191–209.

[Ramamoorthy 85] C. V. Ramamoorthy, W. Tsai, T. Yamaura, and A. Bhide, "Metrics Guided Methodology," Proceedings of COMPSAC'85: 9^{th} Conference on Software Applications, Oct. 1985, pp. 111–120.

[Riddle 79] W. E. Riddle, "An Event–Based Design Methodology Supported by DREAM," Formal Models and Practical Tools for Information Systems Design, N.-J. Schneider (ed.), IFIP, 1979.

[Ross 77] D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, vol. SE–3, nbr. 1, Jan. 1977, pp. 16–34.

[Shaw 80] A. C. Shaw, "Software Specification Languages Based on Regular Expressions," Software Development Tools, W. E. Riddle, and R. E. Fairley (eds.), Springer–Verlag, 1980, pp.148–175.

[Simpson 79] H. R. Simpson and K. Jackson, "Process synchronization in MASCOT," The Computer Journal, vol. 22, nbr. 4, 1979, pp. 332–345.

[Simpson 82] H. R. Simpson, "Act parallel: use MASCOT," Computer Bulletin, March 1982, pp. 6–9.

[Stevens 74] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, vol. 13, nbr. 2, 1974, pp. 115–139.

[Teichroew 77] D. Teichroew, and E. A. Hershey, *III*, "PSL/PSA: A Computer–Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, vol. SE–3, nbr. 1, Jan. 1977, pp. 41–48.

[Teichroew 80] D. Teichroew, P. Macasovic, E. A. Hershey *III*, and Y. Yamamoto, "Application of the Entity–Relationship Approach to Information Processing Systems Modeling," Entity–Relationship Approach to Systems Analysis and Design, P. P. Chen (ed.), IFIP, 1980, pp. 15–36.

[Ward 85] P. T. Ward, and S. J. Mellor, "Structured Development for Real–Time Systems," vol. 1, 2 and 3, Yourdon Press, N. Y. 1985.

[Ward 86] P. T. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 198–210.

[Wasserman 79] A. I. Wasserman, and S. K. Stinson, "A Specification Method for Interactive Information Systems," Proceedings of the Symposium on Specifications of Reliable Software, April 1979, pp. 68–79.

[Wasserman 85] A. I. Wasserman, "Extending State Transition Diagrams for the Specification of Human–Computer Interaction," IEEE Transactions on Software Engineering, vol. SE–11, nbr. 8, August 1985, pp. 699–713.

[Wasserman 86] A. I. Wasserman, P. A. Pircher, D. T. Shewmake, and M. L. Kersten, "Developing Interactive Information Systems with the User Software Engineering Methodology," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 326–345.

[White 87] S. M. White, "A Pragmatic Formal Method for Computer System Definition," Ph.D. Dissertation, Polytechnic University, June 1987.

[Winters 81] E. Winters, "Practical Experience with the Problem Statement Language," Presented at the Conference on the Use of Computer Aids in Systems Development, London, England, June 25–26, 1981.

[Yau 83] S. S. Yau, and M. U. Caglayan, "Distributed Software System Design Representation Using Modified Petri Nets," IEEE Transactions on Software Engineering, vol. SE–9, nbr. 6, Nov. 1983, pp. 733–745.

[Yau 86] S. S. Yau, and J. P. Tsai, "A Survey of Software Design Techniques," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 6, June 1986, pp. 713–721.

[Zave 86] P. Zave, and W. Schell, "Salient Features of an Executable Specification Language and its Environment," IEEE Transactions on Software Engineering, vol. SE–12, nbr. 2, Feb. 1986, pp. 312–325.

UNIVERSITY OF MICHIGAN
3 9015 02527 8055