The International Journal of Flexible Manufacturing Systems, 5 (1993): 95–128

Real-Time Software Methodologies: Are They Suitable for Developing Manufacturing Control Software?

JARIR K. CHAAR
IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598

DANIEL TEICHROEW
IOE Department, The University of Michigan, Ann Arbor, MI 48109-2117

RICHARD A. VOLZ
CS Department, Texas A&M University, College Station, TX 77843-3112

Abstract. Computer-Integrated Manufacturing (CIM) systems may be classified as real-time systems. Hence, the applicability of methodologies that are developed for specifying, designing, implementing, testing, and evolving real-time software is investigated in this article.

The paper highlights the activities of the software development process. Among these activities, a great emphasis is placed on automating the software requirements specification activity, and a set of formal models and languages for specifying these requirements is presented. Moreover, a synopsis of the real-time software methodologies that have been implemented by the academic and industrial communities is presented together with a critique of the strengths and weaknesses of these methodologies.

The possible use of the real-time methodologies in developing the control software of efficient and dependable manufacturing systems is explored. In these systems, efficiency is achieved by increasing the level of concurrency of the operations of a plan, and by scheduling the execution of these operations with the intent of maximizing the utilization of the devices of their systems. On the other hand, dependability requires monitoring the operations of these systems. This monitoring activity facilitates the detection of faults that may occur when executing the scheduled operations of a plan, recovering from these faults, and, whenever feasible, resuming the original schedule of the system.

The paper concludes that the set of surveyed methodologies may be used to develop the real-time control software of efficient and dependable manufacturing systems. However, an integrated approach to planning, scheduling, and monitoring the operations of these systems will significantly enhance their utility, and no such approach is supported by any of these methodologies.

Key Words: manufacturing software, real-time software, software methodologies

1. Introduction

The first burst of activity in software design methods occurred during the late 1960s and early 1970s. It was assumed that software development began with writing a natural langauge requirements specification document and ended with programming, testing, and installation. *Design* was a well-defined activity that happened between requirements analysis and programming. Many methods and criteria were proposed for performing this decomposition (consult Peters and Tripp 1977 and Griffiths 1978 for a survey of these methods). The software development activities were grouped into a framework for managing software design projects; Royce's (1970) software life-cycle model.

The most obvious difference between the design methods of the 1970s and the design methods of the 1980s is that there is no longer consensus on the scope of the software development activities. The traditional software life-cycle was extended to include more phases (see, for example, Boehm 1986; Aoyama 1987; Hekmatpour 1987; Mantei and Teorey 1988; Wolff 1989) due to the increased complexity of the systems being automated, and the increased hardware and software capabilities offered by computer systems. Techniques for improving the outcome of the requirements specification, design, implementation, system testing, and system evolution phases of the traditional software life-cycle were developed. The software development process was improved by automating many steps in the process in order to reduce the cumulative effect on later phases of an error committed in any phase of the process. System prototyping and simulation techniques were introduced to help assess the performance of the resulting software system. A coherent set of these approaches was labeled a methodology, and a set of software tools designed and implemented with the intent of partially or fully automating the application of these techniques was referred to as a software engineering environment. A number of attempts to survey some of these methodologies and environments have been reported (see, for example, Peters and Tripp 1977; Griffiths 1978; Teichroew 1982; Freeman and Wasserman 1983; Du Plessis 1986; Chaar 1987; Chin 1991; Houghton and Wallace 1987; Kelly 1987; Taylor 1989; White 1987; Webster 1988).

This paper presents a synopsis of the methodologies used to specify, design, implement, test, and evolve the software of real-time embedded systems. An adequate model of such systems should encompass both data and control structures, and should model both the structure and behavior of a 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, possibly distributed, real-time embedded system (Dasarathy 1985; Stankovic 1988; Kenny and Lin 1991).

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.

The traditional considerations of hierarchy, information hiding (Parnas 1972), 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.

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.

The vast majority of computer-integrated manufacturing systems that have been built in the 1980s can be classified as real-time process control systems. For these systems, designing and implementing their required software has proven to be a daunting problem; the current practices for such software are archaic and often result in a high cost, an extended development period, and extremely inflexible systems (Naylor and Volz 1987; Ben Hadj Alouane, Chaar and Naylor 1990). Hence, the development of this software can benefit from the methodologies used in developing the software of the more general class of real-time embedded systems.

A major goal of manufacturing in the 1990s is to increase the levels of automation in computer-integrated manufactuirng systems and the efficiency and dependability of such systems. This goal has been partially achieved through the technological advances in both manufacturing and computer hardware. Similar advances in software technology can be achieved by rigorously applying and extending current real-time software methodologies. Hence, the suitability of such methodologies to developing the control software of efficient and dependable manufacturing systems is explored in this article.

Traditionally, control software has been assigned the tasks of monitoring manufacturing systems by coordinating the operations of their devices, and by handling the faults that may occur while these operations are being performed. Planning the sequence of operations of these devices, and scheduling the execution of this sequence have been carried out independently, prior to developing the control software of these systems. In contrast with current practice, the goal of designing efficient and dependable manufacturing systems requires an integrated approach to planning the operations of these systems, scheduling these operations, and monitoring their execution (Chaar, Volz and Davidson 1991).

In efficient and dependable manufacturing systems, efficiency is achieved by increasing the level of concurrency of the operations of a plan, and by scheduling the execution of these operations with the intent of maximizing the utilization of the devices of their systems. Furthermore, the most efficient use of such systems may be achieved by shifting the times of occurrence of some operations of the plans of their jobs while preserving the sequence of dependent operations in these plans (Chaar and Davidson 1990). Consequently, planning and scheduling efficient manufacturing systems is an iterative process that can benefit from integrating their planning and scheduling activities.

Dependability in manufacturing systems requires monitoring their operations. This monitoring activity facilitates the detection of any faults that may occur when executing the scheduled operations of a plan, recovering from these faults, and, whenever feasible, resuming the original schedule of the system. In order to be able to resume an original schedule automatically, scheduling and monitoring must be integrated. Furthermore, whenever recovery to the original schedule is not feasible, an alternative plan and a new schedule for processing the remaining set of jobs have to be created. This latter activity can be performed by activating an integrated planning and scheduling procedure when recovery is needed. Consequently, the dependability of manufacturing systems can be achieved by integrating the planning, scheduling, and monitoring activities of these systems.

In this paper, the activities of the software development process are discussed in section 2. Among these activities, a great emphasis is placed on automating the software requirements specification activity. Hence, a set of formal models and languages for specifying these requirements is presented in section 3. A list of some of the formal models and specification languages has been compiled by Berztiss (1987), and an attempt to classify them has been reported by Davis (1988). Sections 4 through 8 survey the major features of some real-time software methodologies and reflect on their applicability in developing the real-time control software of efficient and dependable manufacturing systems. This is followed, in section 9, by a critique of the strengths and the weaknesses of these methodologies.

2. The Software Development Process

The software development process encompasses all the activities required in defining, developing, testing, delivering, operating, and evolving a software project. This traditional process is captured by the software life-cycle, the waterfall model (Royce 1970). This model segments the software life-cycle into a series of successive activities. Each activity requires well-defined input information, utilizes well-defined processes, and results in well-defined products. Resources are required to complete the processes in each activity, and each activity is accomplished through the application of explicit methods, tools, and techniques. In this model, the activities of analysis, software design, implementation, system testing, and system evolution are performed iteratively.

- *Analysis* consist of *planning* and *requirements specification*:
  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 specification* and a *project plan*. The system specification 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 specification* 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, input and output data, 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 the debugging, documentation, and unit testing of the source code. Modern programming languages, e.g., Ada (The United States Department of Defense 1983), 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. The presence of efficient compilers, syntax-directed editors, and symbolic debuggers can also enhance the productivity of the software implementor. For the Ada programming language, these tools are grouped in a Programming Support Environment (APSE) (Ada Joint Program Office 1982; Taylor and Sandish 1985; Marcus et al. 1986; Dawson 1987) to support the development of Ada-based distributed real-time embedded software (Privitera 1982; Cherry 1985, 1987).
- *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 documents.
- *System deployment* activities include installing the implemented software on a single computer system or on multiple computer systems, training the users of this software, and marketing this software product if commercially available.
- *System evolution* activities include enhancement of capabilities, adaptation of the software to new processing environments, and correction of the resulting software bugs.

Software development seldom proceeds in a smooth progression of activities as indicated in the waterfall model. Nevertheless, the waterfall model of the software life-cycle may be an appropriate 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 can be improved by establishing milestones, review points, standardized documents (e.g., DoD's MIL-2167A) and management signoffs, and by incorporating the concept of prototyping.

Prototyping helps in exploring, experimenting, and confirming an appropriate set of software requirements for a system. Prototyping techniques can be broadly classified into three categories (Hekmatpour 1987); namely *throw-it-away* prototyping, *incremental* prototyping, and *evolutionary* prototyping.

In throw-it-away prototyping, a prototype is discarded once the correct set of requirements is identified. Because of its short life span, quality factors such as efficiency, structure, maintainability, full error handling and documentation can be ignored in designing the prototype. The prototype may even be implemented in an environment other than the one intended for the final product.

Incremental prototyping is based on an overall initial design of the system. The system is then implemented one section at a time, and incrementally converges to a final implementation. Hence, only the implementation and evaluation of a software system can be iteratively performed in incremental prototyping.

In contrast, evolutionary prototyping gradually builds a system by iteratively carrying out the design, implementation, and evaluation processes of software development. Initially, enough development is carried out to enable the user to carry out one or more tasks completely. Once more is known about these tasks and how they may affect other tasks, more parts of the system are designed, implemented, and integrated with the rest of the system.

A variant of the traditional software life-cycle model, the spiral software life-cycle model adopts a risk-driven approach to software development (Boehm 1987; Wolff 1989). The software project is partitioned into a set of cycles or rounds, the objectives of each cycle are determined, and alternatives for meeting these objectives are considered. The implementation of each phase of the software development process is preceded by a risk analysis, and prototyping is then used to minimize the risk associated with each phase. An evolutionary prototype is then built in order to identify and solve the areas of uncertainty of a cycle and reduce the risk of propagating an error in this cycle to any subsequent cycles.

Both traditional and spiral software life-cycle models are general frameworks for managing large software projects. Hence, they are suitable for developing manufacturing control software. If properly applied, these models help reduce both the cost and the extended development period of manufacturing control software.

3. Requirements Specification Models

The consistency and completeness of the software requirements specification document can be verified 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 of a software system, and provide a basis for verifying the resulting software product.

Both *relational* and *state-oriented* notations are used to specify the functional characteristics of software. 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 entitites, and data flow between entities. Relational notations include 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, and Petri nets.

3.1. Algebraic Axioms

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.

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. A specification is complete if all desired properties are specified, and there are no undefined entities in the specification. Consistency is achieved if there is a unified interpretation of the relationships among specifications that produces no contradictions.

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

3.2. Regular Expressions

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. Strings 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 of the two components while preserving the original ordering of the atoms within each expression; the resulting expressions are called event expressions or flow expressions (Shaw 1980).

Event expressions impose some restrictions on shuffling; only the subset of all possible shuffles that preserves the atomicity of critical sections as required by message passing and other synchronization operations is allowed. No instructions from another component may be interleaved between the instructions of a critical section of a component. The synchronization scheme in flow expressions has one explicitly defined syntax, and this is the major distinguishing difference between event expressions and flow expressions. A flow expression can be blocked for shuffling by enclosing it within a wait/signal pair, similar to binary semaphores. The enclosed expression is 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 language 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 and Shields 1980).

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 1979). 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 a set of 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 decribing systems with dynamic structure (Avrunin and Wileden 1985). 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 et al. 1986). 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.

3.3. Decision, Event, and Transition Tables

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 actions may be desired or may indicate a specification error. A Karnaugh map can be used to check a decision table for completeness and for 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 are both 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. Transition tables have also been used to develop the control software of manufacturing transfer lines (Fisher 1989).

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.

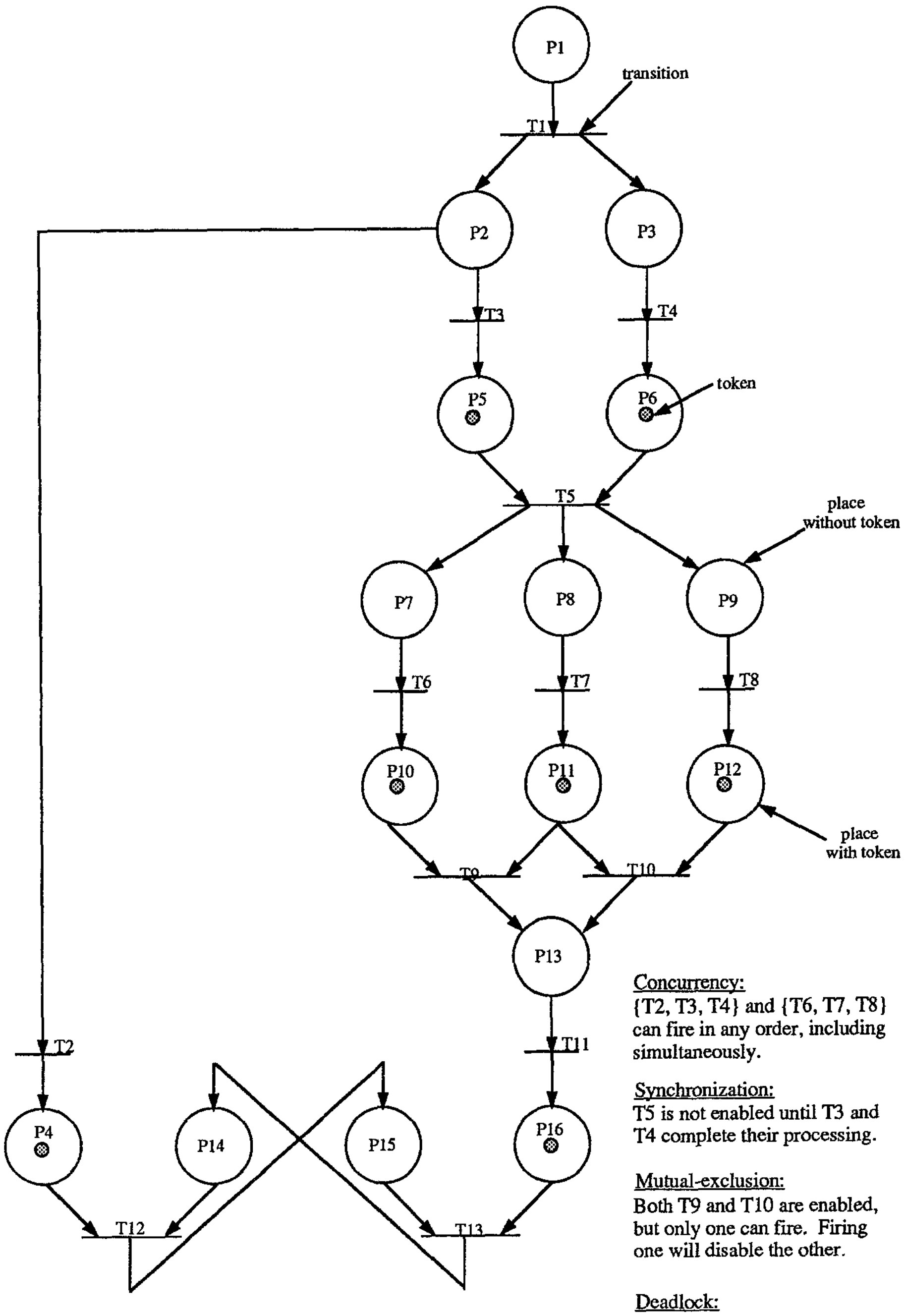
3.4. Petri Nets

A Petri net is a graphical model used to specify, analyze, simulate, and evaluate the dynamic behavior of concurrent systems, and to detect synchronization, mutual exclusion, and deadlock situations in these systems (Peterson 1981). With the introduction of suitable extensions, Petri nets can be used to model a range of hardware and software systems (Coolahan and Roussopoulos 1983; Yan and Caglayan 1983; Molloy 1985; Leveson and Stolzy 1987). The theoretical properties of Petri nets have been studied extensively in the literature (see, for example, Chong Yi 1985a, 1985b; Goltz and Chong Yi 1985; Genrich 1986; Murata and Komoda 1987; Watson 1987). Interest in Petri nets has also generated a long list of software tools (Feldbrugge 1985) for describing their mathematical structure, analyzing their properties (see, for example, Murata et al. 1989; Stotts 1988), simulating their dynamic behavior (see, for example, Ghezzi et al. 1987; Nelson et al. 1983), and generating their layouts (see, for example, Berztiss 1987).

The following sections present the structure and the properties of the original Petri net model, the place-transition net. Some of the extensions that enhance the modeling power of place-transition nets are also introduced. These Petri nets, as implied in a companion paper (Chaar, Teichroew and Volz 1993), are extensively used in developing manufacturing control software.

3.4.1. Place-Transition Nets. The simplest Petri net model, the place-transition net (figure 1), is represented as a bipartite directed graph (Peterson 1981). The two types of nodes in a place-transition 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 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 place-transition net is characterized by an initial marking of places and a firing rule. 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 none of its output places would exceed its capacity after the firing. When a transition fires, the number of tokens at 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



<u>Concurrency:</u>
{T2, T3, T4} and {T6, T7, T8} can fire in any order, including simultaneously.

<u>Synchronization:</u>
T5 is not enabled until T3 and T4 complete their processing.

<u>Mutual-exclusion:</u>
Both T9 and T10 are enabled, but only one can fire. Firing one will disable the other.

<u>Deadlock:</u>
Both T12 and T13 are waiting for the other to fire and neither can proceed.

Figure 1. A place-transition Petri net model.

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 place-transition net can 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 place-transition net serves to relate the models to the specified operating rules to form an event-based simulation (Agerwala 1979).

For a place-transition net that is to model a real hardware device, one of the more important properties is safeness. A place of this net is safe when it has a capacity of 1. 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 place-transition net is safe if all places in the net are safe.

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 $(k + 1)$ up-down counter. A place is bounded if it is k-safe for some k; a place-transition net is bounded if all its places are bounded. A bounded net can be realized in hardware, while a place-transition net with an unbounded place cannot in general be implemented in hardware.

Place-transition 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 place-transition 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 place-transition nets. A deadlock in a place-transition net is a transition (or set of transitions) that cannot fire. A transition is live if it is not deadlocked. Thus, it is always possible to manuever the place-transition net from its current marking to a marking which would allow the transition to fire. Liveness of the place-transition net marks the absence of deadlock in the real system.

In a place-transition 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 place-transition nets are equivalent to finite-state automata. In a finite-state automaton representation of a place-transition net, the state of the net is given by the marking configuration (i.e., a tally of how many tokens are at each place), and transitions leading to next states change the markings to reflect changes in the next state. Despite the above equivalence, place-transition nets provide a more convenient notation for specifying and analyzing concurrent systems; unbounded and nonconservative place-transition nets provide more powerful mechanisms than do finite-state automata.

Place-transition nets are used by Yau and Shatz (1982) to specify the communication modules of distributed software systems. The paper derives place-transition net models for the asynchronous communication mechanism, the synchronous communication mechanism, and the remote procedure call mechanism. The specifications of some standard communication modules are also provided in order to illustrate the use of place-transition nets in analyzing the liveness and boundedness properties of the design specifications of these communication modules.

3.4.2. Modified Petri Nets. A modified form of place-transition nets has been used by Yau and Caglayan (1983) to represent and analyze the design of distributed software systems. 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 place-transition net. Software components correspond to the nonprimitive transitions of a place-transition net, where a nonprimitive transition has a place-transition 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 nonprimitive transition. The nonprimitive transition firing rule of place-transition nets is generalized in modified Petri nets by associating with each software component a control transfer specification, which gives the control flow(s) through the component, and a data transfer specification, which represents the data flow(s) through the component.

3.4.3. Augmented Petri Nets. Place-transition nets have been extended by Coolahan and Roussopoulos (1983) to incorporate the notion of time, as required to capture the timing requirements of embedded real-time systems. In this augmented Petri net, the instantaneous firing of an enabled transition marks the start of execution of a process, and places a token in the output place, p, of this transition that represents the process. This process executes for the nonnegative duration of time, T, associated with place p. An output transition is enabled when the required number of tokens in each of its input places has been at that place for at least T units of time, where the value of T is defined for each place. If only one output transition of p becomes enabled after T, it is fired immediately. Otherwise, a single enabled output transition is selected based on a predetermined firing frequency of this transition. The selected transition is then fired immediately disabling all the other enabled output transitions of place p.

Instead of coupling the transitions of a place-transition net with firing frequencies, Molloy (1985) assigns conditional probabilities to these transitions. The resulting net is called a discrete time stochastic Petri net, and specifies the probability that a transition of the net, once enabled, would instantaneously fire. Generalized stochastic Petri nets are used to model, analyze and evaluate the performance of automated manufacturing systems by Al-Jaar and Desrochers (1990). In another extension to both the augmented Petri net and the discrete time stochastic Petri net, Holliday, and Vernon (1987) define a probability distribution over the possible next states of the net. This distribution is based on the firing frequencies of the transitions of this net. The resulting net is labeled a generalized time Petri net, and is used, as are augmented and discrete time stochastic nets, to evaluate the performance of concurrent and distributed systems.

3.4.4. Timed Petri Nets. Timed Petri nets represent a dual strategy for modeling the timing requirements of real-time embedded systems, and are used by Leveson and Stolzy (1987) to analyze the safety, recoverability, and fault-tolerance aspects of these systems. A timed Petri net is composed of a set of places, a set of transitions, an input function, an output function, and an initial marking along with an 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 reached, the transition must fire immediately. The interval limits are absolute time units specified relative to the software system initialization time.

Added complexity over the untimed place-transition net arises because of the continuous nature of time. The time Petri net is equivalent to a standard place-transition net if the minimum firing time function is set to 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 can be used to enforce sequential timing constraints on enabled transitions by setting the maximum firing time of the transition with higher priority to be less than the minimum firing time of the transition with the lower priority.

The timed Petri net of a system can be analyzed to identify the states of this net that can lead to the occurrence of a set of unplanned events (the hazardous states of the system). The system designer can then ensure 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 a hazardous state.

Once the design is determined to be safe, run-time faults and failures must be considered. A failure is defined as an event, whereas a fault is a state. A failure may result in a fault and is called a fault-starting event. The type of fault that results from the failure must be included in the model in order to analyze the consequences of failures on the system. Hence, a failure transition, which is denoted by a double bar, and a fault condition, which is denoted by a double circle, are introduced. To make analysis practical, a place that 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 that the transition (failure) can fire. Timing information is used to set a limit on the delay needed to resume a normal course of action by the system. Graceful degradation can also be achieved by assigning a lower priority to an alternative path in the system, and by specifying a time-out condition on the execution of this path in case the normal path of execution is not functional.

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 timed 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 this place.

3.4.5. Predicate-Transition Nets. In a predicate-transition net, the tokens are no longer anonymous, but can be structured objects carrying values, and transition firing can be controlled by imposing conditions on these token values. Predicate-transition nets are used by Giordana and Saitta (1985) to model production rules in rule-based systems. They are also used by Murata and Zhang (1986, 1988) to detect parallelism in interpreting Horn clause-based logic programs in order to improve the efficiency of the control component of these programs. Moreover, Peterka and Murata (1989) present a proof-procedure and answer-extraction scheme that corresponds to a firing sequence that fires the goal transition of the predicate-transition model of a logic program.

In a predicate-transition net, places correspond to predicates with variable extensions, and transitions represent classes of elementary changes of extensions. The arcs of the net are labeled with sums of tuples of variables; the length of each tuple is the number of instances of arguments that appear in the predicate connected to the arc. The symbol ¢ denotes the zero-tuple, i.e., a no-argument predicate (a place in place-transition nets). The capacity of a place defines the number of copies of the same token that this place can carry simultaneously.

The tokens of a predicate-transition net are tuples of typed objects. The object types of the net are each coupled with a set of operations and relations that apply to all the tokens of the type. These operations and relations are used as inscriptions inside some transitions of the net. The presence of a token in a place denotes the fact that the predicate associated with this place is true for the particular instantiation of the tuple of arguments contained in the token. A transition of the predicate-transition net is enabled whenever:

1. Each input place of the transition contains at least as many tokens as specified by the label of the arc that connects this place to the transition.
2. The tokens occurring in the input places have values satisfying the inscribed formula, if any, in the transition.
3. The capacity of each output place is not exceeded by firing the transition.

When a transition is enabled, it can be fired by removing from each input place of the transition a number of tokens specified by the label on the arc connecting this input place to the transition, and by adding to each output place of the transition a number of tokens specified by the label on the arc connecting the transition to this output place. The lack of any formula inscribed in a transition means that the firing depends only upon the existence of tokens in the input places and upon the capacity of the output places.

4. SREM

The Software Requirements Engineering Methodology (SREM), developed by Alford (1977), describes a sequence of steps for specifying the functional and performance requirements of real-time software systems. These requirements are expressed in the Requirements State Language (RSL), and analyzed with the tools of the Requirements Engineering Validation System (REVS) (Bell et al. 1977; Davis and Vick 1977). The tools are used to perform automated consistency and completeness analysis, produce documentation, and generate simulators for the specification requirements of a system (Alford 1985).

RSL and its corresponding graphical notation, the Requirements networks (R-nets) (Bell et al. 1977), represent an extension to conventional finite-state machiens. Each R-net (figure 2) specifies the transformation of a single input message plus the current state into some number of ouptut messages plus an updated state. Only a single R-net may be active at a time.

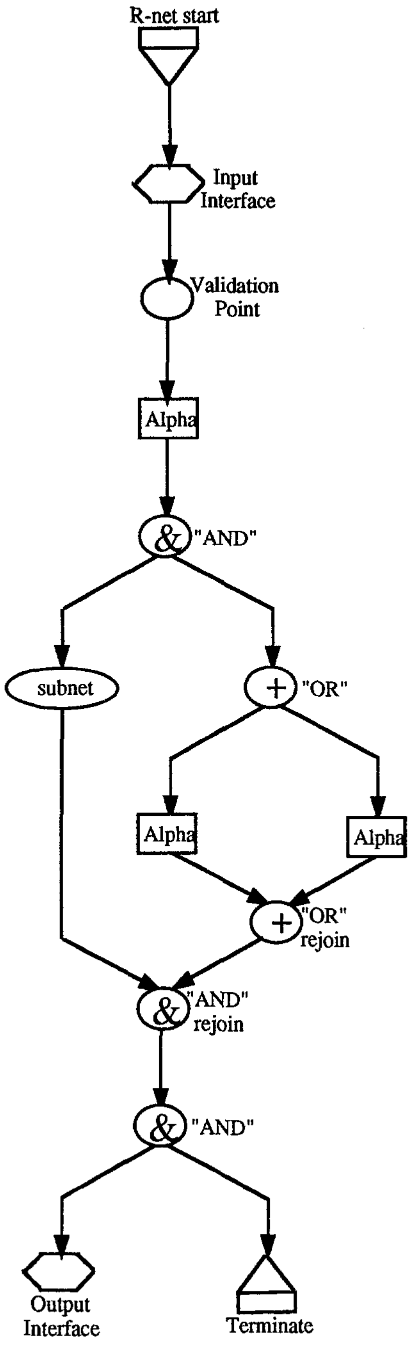


Figure 2. The requirements network (R-net) of SREM.

The R-net includes, at most, one input interface that receives a single input message, some number of output interfaces, some number of ALPHAs, and some number of subnets. The ALPHA, denoted by a rectangle, is used to specify a task of the system. A subnet, represented as an oval, is defined as a processing subgraph with one entrance and one exit point; the subnet is used to show common processing requirements under different conditions and to shorten the length of an R-net.

R-nets can be used to specify parallel (order-independent) operations using the AND node (&). Fan-in at an AND node is a synchronization point; all parallel operations must be completed before any subsequent operations are initiated. The OR node (+) has a condition associated with each operation and 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. Ramamoorty et al. (1985) outline the sequence of steps for transforming an R-net into an equivalent place-transition Petri net.

Accuracy and response time performance requirements are expressed by means of the concept of a path through an R-net. A path of processing through an R-net and its associated subnets is specified by inserting named nodes, called validation points, in the R-net graph and then defining the path in terms of a sequence of these validation points. A path can include a set of AND nodes and a set of OR nodes, can traverse multiple R-nets by using the EVENT node (which enables another R-net) as a connector.

The Requiremens Engineering and Validation System (REVS) operates on RSL statements. REVS consists of threee major components:

1. A translator for the Requirement Statement Language (RSL).
2. A centralized relational 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 evaluate the performance characteristics of the specified systems.

Automated tools for processing information in the ASSM include an interactive graphics package to aid in specifying flow paths, static checkers that check for consistency and completeness of the R-net models of a system, and an automated simulation package that generates and executes simulation models of this system.

SREM describes the sequence of steps for defining requirements, expressing them in RSL, and using the tools to check consistency and completeness. The first step of the methodology identifies all the I/O messages, R-nets, and ALPHAs of a system. This is followed by generating the plots of the R-nets, defining the input and output data of the ALPHAs, and checking the consistency and completeness of the system requirements. The validation points of the R-nets are defined next, and the performance requirements of the system are verified by prototyping and testing the algorithms enclosed in the ALPHAs of the nets, and by checking the response time of all the paths of these nets.

A recent extension of the SREM concepts sought to define the requirements of distributed real-time software systems (Alford 1985). The result was called the Distributed Design Methodology (DDM). DDM uses the System Specification Language (SSL) to specify the functional and performance requirements of a system in terms of time functions and their decompositions. A time function is specified in terms of its input data, output data, completion criteria and performance. The R-nets of RSL are used consequently to capture the software requirements of each function. The ALPHAs of these R-nets are mapped onto modules and data structures; this mapping is expressed in the Module Design Language (MDL). The Distributed Design Language (DDL) is then used to assign the above modules to their appropriate computing nodes subject to the required performance of the time functions of the system and the computing power of these nodes.

When all the functional and performance requirements of a system have been specified and verified with the tools of the REVS, these requirements are translated by the Task Specification Language (TSL) into a design to be coded in a high-level programming language.

While SREM is concerned with verifying the consistency and correctness of the specification requirements of a real-time system, DDM can be viewed as a direct extension of the capabilities offered by SREM; DDM is used to verify the consitency and correctness of the specification requirements of a distributed real-time system. Because of the use of the RSL, SSL, DDL, MDL, and TSL languages in the software design and implementation phases of DDM, applying the concepts of this methodology may not be a straightforward task. This limitation can be overcome by using a specification language such as Anna (Luckham et al. 1985, 1987) that has the same syntax as its implementation language, Ada, and automatically translates these requirements into an implementation. The use of Anna can extend SREM to cover all the phases of the traditional software life-cycle, and can provide a basis for supporting both the incremental prototyping and the evolutionary prototyping paradigms.

Developing the control software of manufacturing systems can benefit from the use of SREM. In particular, the functional and timing requirements of a system can be coded in SSL, and the plan of a job of this system can be expressed as an R-net. REVS can be used to verify the consistency and completeness of this R-net, and a simulation model of the manufacturing system can be automatically generated and executed in order to evaluate the performance of this system when processing a batch of these jobs. However, the optimality of the resulting schedule is not guaranteed. Moreover, recovery actions from probable faults may be added to the R-net and verified by REVS. When the performance of a manufacturing system is satisfactory, TSL can be used to create a high-level language implementation of its control software.

5. SARA

The System ARchitects' Apprentice (SARA) is a computer-aided design environment that supports a structured, multi-level methodology for the design of hardware and software systems (Estrin et al. 1986). The functional and performance requirements of a system and its operating environment are specified prior to the use of SARA in designing this system. SARA permits the separation of structure and behavior in models, and supports top-down decomposition of a complex system into manageable pieces as well as bottom-up composition of reusable building blocks (Campos and Estrin 1977). SARA emphasizes interfaces as places where inconsistencies are revealed and where information hiding in enforced.

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.

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 (figure 3) primitives include *nodes* (circles) which represent processing activities, 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 place-transition Petri net model, and supports similar reachability analysis methods (Estrin et al. 1986).

  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 is initiated. 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 ouptuts 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 processor (hexagons), datasets (rectangles), 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

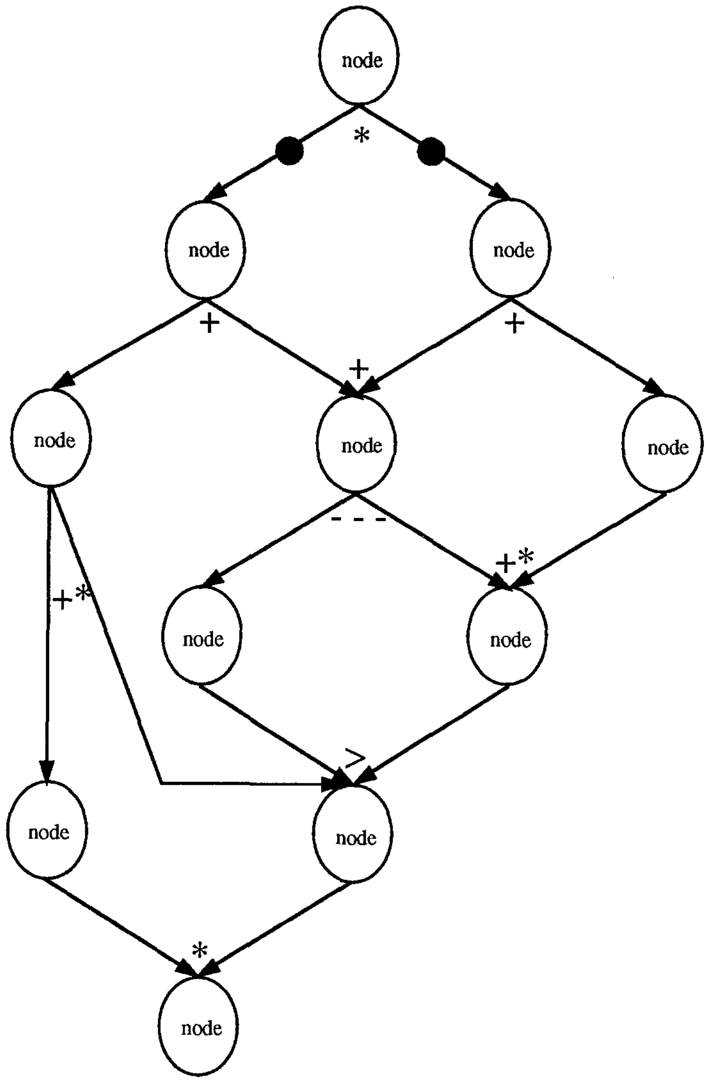


Figure 3. The GMB control graph of SARA.

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) tool 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), and implements a strong reduction algorithm that reduces the size of the state-space which must be explored, yet allows most of the analysis properties to be determined. Moreover, an interactive simulator allows designers to perform experiments on the GMB model of the system. Consequently, both performance analysis and correctness analysis of the same model can be performed in SARA.

SARA does not offer any support for implementing the software of a system following the verification of both functional and performance requirements of this software system. The Ada-based specification language Anna (Luckham et al. 1985, 1987) can replace the SL and MID languages of SARA in specifying the above requiremens. This language allows the specification and implementation of a system to be carried out concurrently, and can extend the capabilities of SARA to cover the implementation phase of the traditional software life-cycle, and to support incremental and evolutionary prototyping.

SARA can be used to verify both the functional and performance requirements of manufacturing control software. A job plans can be expressed as a GMB that can also capture the recovery actions from probable faults that may occur while processing a batch of these jobs. Processing a batch of jobs is simulated by repeatedly executing the GMB of the system. However, the optimality of the resulting schedule cannot be guaranteed.

6. MASCOT

The acronym MASCOT stands for Modular Approach to Software Construction Operation and Test. One of the main purposes of MASCOT is to aid the reliable design of concurrent systems. The design phase is provided for by the activity-channel-pool (ACP) diagram which allows the representation of the modular breakdown, without explicitly including the concurrency requirements in the diagram itself. MASCOT is a language-independent and machine-independent design tool, supported by a programming system (Jackson and Simpson 1975; Simpson and Jackson 1979; Simpson 1982). It can provide support through the design, implementation, and testing stages of the traditional software life-cycle.

MASCOT software is modeled as a set of subsystems that 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 (figure 4) 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.

A MASCOT subsystem consists of one or more activies, 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 that are required when the root procedure is used to support an activity.

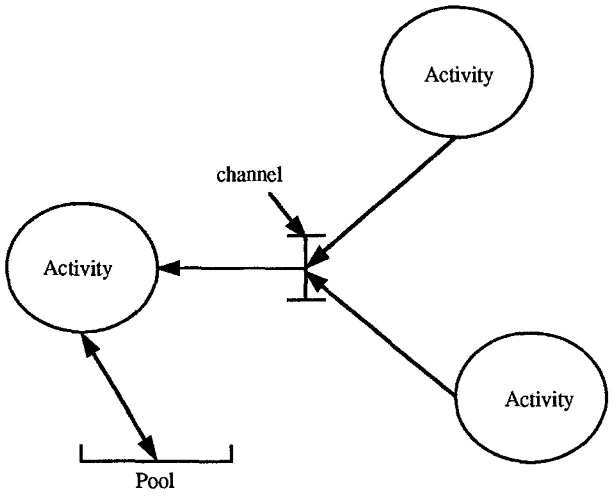


Figure 4. The activity-channel-pool (ACP) diagram of MASCOT.

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 bidirectional interface. Pools and channels are generally accessed by means of access procedures.

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 its execution, clearing the STIM. Otherwise, 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 allows MASCOT to support the 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.

In MASCOT, the requirements specification and software design phases of the traditional software life-cycle are combined, and evolutionary prototyping is supported by the *form* facility. Furthermore, the ACP diagram of a system can be translated into a high-level modular language implementation of this system such as Ada (Dibble 1982) or Modula-2 (Budgen 1985).

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, MASCOT 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.

MASCOT can be used to develop the control software of efficient and dependable manufacturing systems. The devices of these systems can be encapsulated into MASCOT subsystems and their operations can be modeled as MASCOT tasks. The ACP diagrams of a system are then translated into high-level language modules. These modules implement the control software of the system and may be used to simulate the operations of this system. Planning, scheduling, and fault detection and recovery subsystems must be implemented in MASCOT as part of the control software of a manufacturing system.

7. ESML

The Extended Systems Modeling Language (ESML) is a requirements specification language, based on the Hatley (1984) and Ward-Mellor (1986) extensions to data flow diagrams, that captures control and timing information of real-time systems (Bruyn 1988; Ward 1988). The flow diagram extension used in ESML is called the transformation schema (figure 5.)

In this schema, terminators are depicted as rectangles, and represent an entity in the external environment of the system whose details are of no interest within the schema. Transformations are depicted as rectangles with rounded corners, and include flow transformations and control transformations. A solid rectangle with rounded corners is a flow transformation that describes a function performed on the data of a system. A dashed rectangle with rounded corners is a control transformation that determines the activation time of other transformations of the schema, and the time interval during which these transformations are enabled. Each transformation within a schema carries a label describing the function or control performed. The line across the upper part of the transformation isolates an identification field.

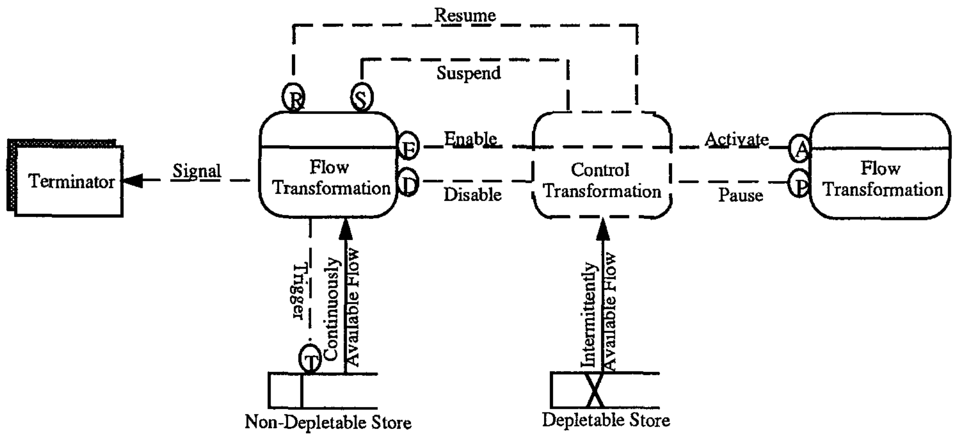


Figure 5. The transformation schema of ESML.

Value Bearing Flows are represented by directed solid line segments, and represent the input and output data of enabled flow transformations. A flow with a single arrowhead is called a continuously available flow that can be used by the flow transformations of a schema at every point in time. In contrast, an intermittently available flow has a double arrowhead, and can be used by the flow transformation of the schema at some discrete points in time.

Nonvalue Bearing Flows are represented by directed dashed line segments. A flow with a double arrowhead is called a signal, and represents the recognition or reporting of the occurrence of an event in a transformation or a terminator of the schema. Prompts are represented by directed dashed line segments with small circles at their ends. Prompts represent control imposed by one control transformation on another transformation.

There are seven distinct prompts distinguished by a letter placed in the small circle at the end of the line segment. A trigger causes a flow transformation to perform a time-discrete action such as producing an intermittently available flow or storing the instantaneous value of a continuously available flow. The enable and disable prompts initiate and terminate the activity of a transformation. The suspend and resume prompts are similar to enable and disable except that a suspended transformation resumes action from the point it was stopped. In addition to the single action prompts, *activate* is a combination of enable and disable, and *pause* is a combination of suspend and resume. In contrast with prompts, the continuously available flows, intermittently available flows, and signals of a scheme carry unique labels.

Stores are represented by a pair of parallel lines closed on the left side. A nondepletable store is drawn with a vertical solid line to isolate an identification field. It represents information that persists within the schema and is accessible to transformations at discrete points in time. A depletable store is drawn with an "X" inside the store to isolate an identification field. It represents a repository for flows that are consumed as they are used.

A complete ESML must contain a specification of each value-bearing flow and store. This specification defines the structure and attributes (e.g., update rate and default value) of a flow, and the structure of the content of a store and its capacity. A depletable store must be specified as either a stack or a queue.

The specification of a flow transformation must describe either a lower-level transformation schema or the function performed by a primitive flow transformation. Given the transformation inputs, the specification of a primitive flow transformation should unambiguously determine the corresponding outputs of this transformation. On the other hand, the specification of a control transformation must define the control logic of this transformation in the form of a Mealy-type or a Moore-type finite state automaton.

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, and permits the verification of both functional and performance requirements of a software system. Generally speaking, the presence of a token indicates actual or potential activity, and a set of execution rules governs their displacement in the schema. Tokens may be placed on transformations, flows and depletable stores; nondepletable stores do not enter directly into the execution rules, and tokens are not placed on them. An interactive graphical environment for generating the transformation schema of a system, and simulating the interaction of the transformations of this schema has been implemented by Blumofe and Hecht (1988).

The transformation schema of ESML provides for an adequate decoupling of the data flow and control flow aspects of a system. The behavior of the system is more adequately described than its structure. The guidelines provided by the Design Approach for Real-Time Systems (DARTS) (Gomaa 1984, 1986, 1988) can be used in structuring a system specified in ESML into a set of subsystems and in structuring each subsystem into a set of concurrent tasks. Coupling ESML and DARTS can help cover all the phases of the traditional software life-cycle of a system.

In DARTS, the structuring of an application into subsystems is based on functional decomposition; a set of flow transformations that perform closely related functions may be grouped into a subsystem. Flow transformations that access a common depletable or nondepletable store may also be grouped into a subsystem. The following criteria can then be used to decompose these subsystems into concurrent tasks:

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

ESML and DART can be used in developing the control software of manufacturing systems. The operations and recovery actions of the components of such systems are modeled as flow transformations. On the other hand, the job plans and schedules of these systems are modeled as control transformations. The operation of a system is simulated by executing its transformation schema. However, the optimality of both the plans and the schedules of this system is not guaranteed by the use of ESML and DART.

8. STATEMATE

The STATEMATE system is a graphical environment intended for the specification, analysis, design, and documentation of large and complex reactive systems such as real-time embedded systems, control and communication systems, and interactive software (Harel et al. 1988). Three closely related views of a system can be specified and analyzed in STATEMATE; the structural view, the functional view, and the behavioral view.

A hierarchical decomposition of the system into its components (called modules) is provided in the structural view of this system. The data and control flows between these components are also specified. The language of module-charts is used to specify the structural view of a system. In a module-chart (figure 6), modules are depicted as rectilinear shapes, with storage modules having dashed sides and with encapsulation capturing the submodule relationship. Environment modules appear as dashed-line rectangles external to that of the system itself. Information flow is represented by labeled arrows or hyperarrows (arrows with more than two endpoints). Various kinds of connectors can appear in these charts, both to abbreviate length arrows and to denote compound data items.

A hierarchy of the activities performed by the system, complete with the details of the data items and control signals that flow between them is provided by the functional view of this system. This view provides only the decomposition into activities and the possible flow of information in a system, but says little about how these activities and their associated

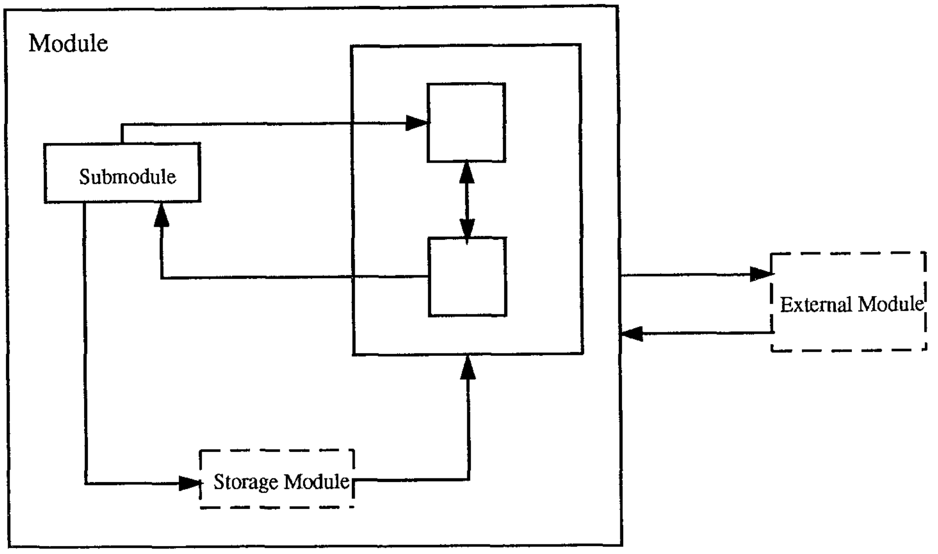


Figure 6. The module-chart of STATEMATE.

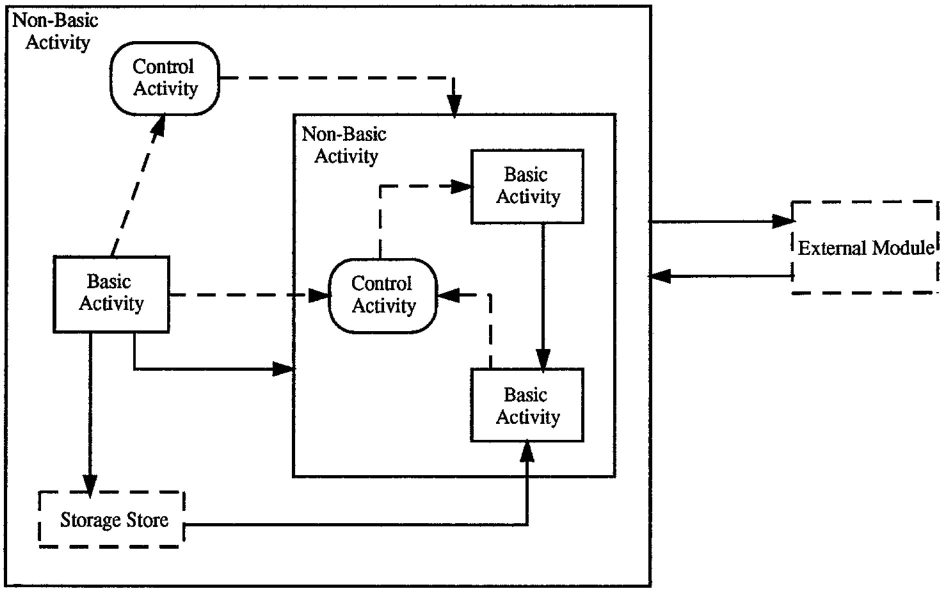


Figure 7. The activity-chart of STATEMATE.

inputs and outputs are controlled during system operation. The functional view of a system is captured by the language of activity-charts. Graphically, an activity-chart (figure 7) is very similar to a module-chart, but the rectilinear shapes of this chart stand for the activities, or the functions carried out by the system. Solid arrows represent the flow of data items and dashed arrows capture the flow of control items.

A typical activity will accept input items and produce output items during its active time-spans, its inner workings being specified by its own lower-level decomposition. Activities that are basic (i.e., cannot be decomposed further into lower-level activities) are described as simple input/output transformations. Furthermore, the module that implements a given activity is specified at this stage.

Activity-charts may also contain two additional kinds of objects: data stores and control activities. Data stores can be databases, data structures, or buffers, and typically correspond to the storage modules in the module-chart. Hence, the storage module that implements a given data store is also specified at this stage. The control activities constitute the behavioral view of the system, and appear in the activity-chart as empty round-edge boxes only, one (at most) within each nonbasic activity. In general, a control activity has the ability to control its sibling activities by essentially sensing their status and issuing commands to them. The statecharts graphical language is used to describe the contents of these control activities.

Statecharts were introduced by Harel and Pneuli (1985), Harel (1988) and Harel et al. (1988). They are extensions of conventional finite-state machines (FSMs) and their visual counterpart, state-transition diagrams. Conventional state diagrams are inappropriate for the behavioral description of complex control since they suffer from being flat and unstructured, are inherently sequential in nature, and given rise to the so-called state explosion phenomenon (i.e., small extensions of a system cause unacceptable growth in the number of its states). These problems are overcome in statecharts by supporting the decompositions of states in an AND/OR fashion, combined with an instantaneous broadcast mechanism. Furthermore, these extensions allow transitions to leave and enter states on any level of the decomposition.

In a statechart (figure 8), states that have common input and/or output transitions can be clustered into a new higher-level state (OR clustering). States U and W are examples of this clustering; the system is in state U whenever it is in either states S or T, and is in state W whenever it is in either states X or Y. The components of this OR clustering are called exclusive states. On the other hand, state Z is decomposed (AND decomposition) into states U and W; the system is in state Z whenever it is in both states U and W. The components of an AND decomposition are graphically separated by dotted lines, and are called orthogonal states because concurrent and independent state transitions can be represented in these components. The AND decomposition can be carried out on any level of states, and is therefore more convenient than allowing only single-level sets of communicating finite-state machines; orthogonality is the feature that statecharts employ to solve the state explosion problem.

The general syntax of an expression labeling a transition in a statechart is $\alpha[C]/\beta$ where α is the event that triggers the transitions, C is a Boolean condition that guards the transition from being taken unless it is true when α occurs, and β is an action that is carried out if, and precisely when, the transition is taken. Any of these can be omitted. An action can also be associated with a state of the statechart.

In STATEMATE, the consistency and completeness of the module-charts, activity-charts, and statecharts of the specification of a system can be checked, and the dynamics of the system can be simulated by executing this specification. Furthermore, the activity-chart and the statecharts of a system can be automatically translated into Ada. Code can also be added by the user to emulate the environment and/or implement the bottom-level basic activities. This translation results in an executable prototype of the system because the Ada code produced by STATEMATE will not necessarily be as efficient or as fine-tuned as production code.

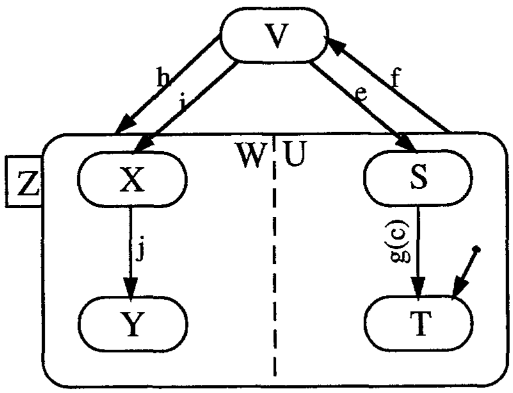


Figure 8. The statechart model of STATEMATE.

STATEMATE can be used in developing the control software of manufacturing systems. The manufacturing components of such systems are encapsulated in module-charts. The operations and fault recovery actions of each such component are captured as activity-charts. Job plans and schedules are expressed as statecharts. An implementation of the control software of these systems can be automatically derived in a high-level language. This implementation is also used to simulate the behavior of these systems. However, the optimality of both the job plans and the schedules of such systems is not guaranteed.

The Modechart, a variant of the statechart visual language that provides a more adequate treatment of the stringent timing constraints of hard-real-time systems, has been proposed by Jahanian, Lee and Mok (1988). Modecharts make use of the concept of modes; modes are partitions of the state space of a system, and are an effective way for the modular specification of large state machines. The formal semantics of Modecharts are expressed in terms of Real Time Logic (RTL). RTL is a first-order predicate logic invented primarily for reasoning about timing properties of real-time systems (Jahanian and Mok 1986). RTL provides a uniform way for the specification of both the relative and the absolute timing of events.

9. Conclusion and Critique

Table 1 is a summary of some of the features of the approaches to developing real-time software presented in this paper. These approaches stress the importance of formal models and specification languages in capturing the functional and timing requirements of real-time embedded systems, and the role of prototyping in refining these requirements. Furthermore, the semantics of these models and languages can capture the behavior of their specified systems.

Table 1. Approaches to developing real-time software.

Acronym	Domain	Basis of Approach	Specification Language	Environment/ Tools	Developer/ Marketer	Status
SREM/DDM	Ballistic missiles	R_nets	RSL, SSL, MDL, TSL	REVS	TRW	Developed for US Army
SARA	Hardware design	GMB	SL, MID	Multix/CFA	UCLA	Academic research
MASCOT	Military hardware	ACP		Military computers	UK's DD-RE	Developed for UK's DD
ESML	Real-time sofware	SART	Extended DFD	Many	Many	Commercial product
STATEMATE	Real-time software	statechart	Graphical	Many	i-logix	Commercial product

Various teams have designed and implemented software tools that enhance the use of the above formal models and specification languages. These tools are targeted towards verifying the consistency and completeness of the specification of a real-time embedded system. System behavior can also be simulated to detect synchronization, mutual exclusion, or deadlock situations in this system, and to verify the timing requirements of the system.

Hierarchical decomposition of real-time embedded systems is also supported by the above models and languages. This decomposition is primarily used to manage the complexity of the specification diagrams and modules of these systems. On the other hand, the reuse of these diagrams and modules is highly dependent on the strategy followed by the software designers of a system. In particular, the methodologies of this paper can support the reuse of the objects, the processes and the functions of a real-time embedded system specification. SREM's R-nets and ESML's transforms permit the reuse of both objects and functions. SARA's modules, MASCOT's subsystems, and STATEMENT's modules permit the reuse of objects.

By following an object-oriented, process-oriented, or function-oriented approach, the above methodologies cover the design phase of the software life-cycle. Moreover, the generation of an executable prototype of a real-time embedded system provides a platform for incrementally refining this software into a production quality product, and for thoroughly testing this product.

Consequently, the methodologies of this paper may be used to develop the real-time control software of efficient and dependable manufacturing systems. Integrating the planning, scheduling and monitoring activities of this control software is essential to achieving efficiency and dependability for these systems. However, none of the above methodologies supports this integrated approach. In particular, the formal models and specification languages of these methodologies assume that a fixed control sequence is repeatedly performed by their specified real-time embedded systems, and do not provide any capabilities for optimizing the use of the resources of these systems. Furthermore, it is assumed that the behavior of the system under both normal and faulty conditions is fully simulated before the software product controls the actual system.

Planning the control sequence of efficient and dependable manufacturing systems is required because these sequences are job-specific. The execution of these control sequences requires the use of the devices of these systems. Hence, scheduling attempts to maximize the usage of these expensive devices. On the other hand, monitoring permits the detection and subsequent correction of any faults that may occur to these devices during the execution of the above control sequences.

To cure the deficiences of real-time software methodologies when used in developing the control software of efficient and dependable manufacturing systems, a methodology for developing this software and a set of software tools that enhance the applicability of this methodology have been implemented (Chaar 1990; Chaar, Volz and Davidson 1991). The tools aid in planning the set of operations of a manufacturing job, generating a cyclic schedule for processing a batch of jobs, and monitoring the operations of the system while this batch is being processed.

In this methodology, a component-oriented rule-based language is used to specify the formal models of manufacturing systems. A model captures the state of a component of the system in a set of first-order logic predicates, and it captures the semantics of the operations performed by this component in a set of rules that determine the preconditions and postconditions of an operation. These models are used in planning the sequence of operations of each class of jobs to be manufactured by these systems.

To achieve efficiency, the reservation table technique is used to create optimum cyclic job-shop schedules for processing a batch of identical jobs or a mix of jobs from several classes on these systems. A reservation table is derived from the plan of a job. This table is then used to determine the theoretical maximum job initiation rate and the set of all possible initiation strategies for the batch. In some cases, this theoretical maximum rate is achieved by increasing the flow time of the job. The above technique inherently allows multiple devices to be reserved concurrently, it can deal with transport time explicitly, and it achieves higher initiation rates by including cycles that involve multiple job initiations.

To achieve dependability, a plan-oriented fault detection and correction strategy is proposed. This strategy can automatically handle any combination of faults that may occur when monitoring the operations of manufacturing systems. A fault-tree is consulted prior to executing the scheduled operations of a plan, and the faults that affect the execution of these operations are handled subsequently. Resuming the original cyclic schedule is attempted, whenever feasible.

References

Ada Joint Program Office, "Ada Methodologies: Concepts and Requirements," Technical report, The United States Department of Defense (November 1982).

Agerwala, T., "Special Feature: Putting Petri Nets to Work," *Computer*, pp. 85–94 (December 1979).

Al Jaar, R.Y. and Desrochers, A.A., "Performance Evaluation of Automated Manufacturing Systems Using Generalized Stochastic Petri Nets," *IEEE Transactions on Robotics and Automation*, Vol. 6, No. 6, pp. 621–639 (December 1990).

Alford, M.W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, pp. 60–69 (January 1977).

Alford, M.W., "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, pp. 36–46 (April 1985).

Aoyama, M., "Concurrent Development of Software Systems: A New Development Paradigm," *ACM SIGSOFT Software Engineering Notes*, Vol. 12, No. 3, pp. 20-24 (July 1987).

Avrunin, G.S. and Wileden, J.C., "Describinig and Analyzing Distributed Software System Design," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 380-403 (July 1985).

Avrunin, G.S., Dillon, L.K., Wileden, J.C. and Riddle, W.E., "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, pp. 278-292 (February 1986).

Bell, T.E., Bixler, D.C. and Dyer, M.E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 49*-60 (January 1977)

Ben Hadj-Alouane, N., Chaar, J.K. and Naylor, A.W., "The Design and Implementation of the Control and Integration Software of a Flexible Manufacturing System," *The Proceedings of The First International Conference on Systems Integration*, New Jersey, pp 494-502 (April 1990).

Berztiss, A.T., "Specification of Visual Representation of Petri Nets," *The Proceedings of the 1987 Workshop on Visual Languages* (August 1987).

Berztiss, A.T., "Survey of Formal Specification Methods," Technical report, Software Engineering Institute, Carnegie-Mellon University (1987).

Blumofe, R. and Hecht, A., "Executing Real-Time Structured Analysis Specification," *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 3, pp. 32–40 (July 1988).

Boehm, B.W., "A Spiral Model of Software Development and Enhancement," *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 4, pp. 14–24 (August 1986).

Boehm, B.W., "Improving Software Productivity," *Computer*, pp. 43–57 (September 1987).

Bruyn, W., Jensen, R., Keskar, D. and Ward, P.T., "ESML: An Extended Systems Modeling Language Based on the Data Flow Diagram," *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 1, pp 58–67 (January 1988).

Budgen, D., "Combining MASCOT with Modula-2 to aid the Engineering of Real-Time Systems," *Software-Practice and Experience*, Vol. 15, pp. 767–793 (August 1985).

Campos, I.M. and Estrin, G., "Concurrent Software System Design Supported by SARA at the Age of One," *The Proceedings of the 3rd International Conference on Software Engineering*, Atlanta, Georgia, pp. 230–242 (1977).

Chaar, J.K., "Software Design Methodologies: A Survey," Technical Report RSD-TR-20-87, Robot Systems Division, The University of Michigan (Ocotber 1987).

Chaar, J.K. and Davidson, E.S., "Cyclic Job Shop Scheduling Using Reservation Tables," *The Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp 2128–2135 (May 1990).

Chaar, J.K., Teichroew, D. and Volz, R.A., "Developing Manufacturing Control Software: A Survey and Critique," *International Journal of Flexible Manufacturing Systems*, Vol. 5, No. 1, pp. 53–88 (1993).

Chaar, J.K., Volz, R.A. and Davidson, E.S., "An Integrated Approach to Developing Manufacturing Control Software," *The Proceedings of the 1991 IEE International Conference on Robotics and Automation*, Sacramento, California, pp. 1979–1984 (April 1991).

Cherry, G.W. and Crawford, B.S., "The PAMELA Methodology," Technical report, Thought**Tools (November 1985).

Cherry, G.W., "PAMELA 2: An Ada-Based Object-Oriented Design Method," Technical report, Thought**Tools (1987).

Chin, R.S. and Chanson, S.T., "Distributed Object-Based Programming Systems," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 91–124 (March 1991).

Chong Yi, Y., "Synchronic Distances in C/E Systems," *Advances in Petri Nets*, pp. 101–121 (1985a).

Chong Yi, Y., "Process Periods and System Reconstruction," *Advances in Petri Nets*, pp. 122–141 (1985b).

Coolahan Jr., J.E. and Roussopoulos, N., "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp. 603–616 (September 1983).

Dasarathy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, pp. 80–86 (January 1985).

Davis, A.M., "A Comparison of Techniques for The Specification of External System Behavior," *Communications of the ACM*, pp. 1098–1115 (September 1988).

Davis, C.G. and Vick, C.R., "The Software Development System," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, pp. 69–84 (January 1977).

Dibble, R., "Software Design and Development Using MASCOT," *Software for Avionics, AGARD Conference Reprint nbr. 330*, (1982).

Dowson, M., "ISTAR—An Integrated Project Support Environment," *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 27–33 (January 1987).

Du Plessis, A.L., "*A Software Engineering Environment for Real-Time Systems*," PhD thesis, University of South Africa (June 1986).

Estrin, G., Fenchel, R.S., Razouk, R.R. and Vernon, M.K., "SARA (Systems Architects Apprentice): Modeling, Analysis and Simulation Support for Design of Concurrent Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, pp. 293–311 (February 1986).

Feldbrugge, F., "Petri Net Tools," *Advances in Petri Nets*, pp. 203–223 (1985).

Fisher, J.P., "Zone Logic—Increased Machine Productivity through Artificial Intelligence," *The Proceedings of the 18th Annual International Programmable Controllers (IPC) Conference* (April 1989).

Freeman, P. and Wasserman, A.I., *Tutorial on Software Design Techniques*. (eds.) IEEE Computer Society, Washington, D.C. 4th ed. (1983).

Genrich, H.J., "Net Theory and Application," Kugler, H.-J., editor, *Information Processing (IFIP)* 86, pp. 823–831 (1986).

Ghezzi, C., Mandrioli, D. and Pezzé, M., "Petri Nets as a Support to Symbolic Execution of Concurrent Ada Programs," Technical Report 87-007, Politecnico di Milano-Dipartimento di Elettronica (1987).

Giordana, A. and Saitta, L., "Modeling Production Rules by Means of Predicate Transition Networks," *Information Sciences*, Vol. 35, pp. 1–41 (1985).

Goltz, U. and Chong Yi, Y., "Synchronic Structure," *Advances in Petri Nets*, pp. 233–252 (1985).

Gomaa, H., "A Software Design Method for Real-Time Systems," *Communications of the ACM*, Vol. 27, No. 9, pp. 938–949 (September 1984).

Gomaa, H., "Software Development of Real-Time Systems," *Communications of the ACM*, Vol. 29, No. 7, pp. 657-668 (July 1986).

Gomaa, H., "Extending the DARTS Software Design Method to Distributed Real Time Applications," *The Proceedings of the 2lth Annual Hawaii International Conference on System Sciences (HICSS-21)*, Hawaii, Vol. II: Software, pp. 252–261 (January 1988).

Griffiths, S.N., "Design Methodologies—A Comparison," *Structured Analysis and Design*, Vol. 11, pp. 133–166 (1978).

Harel, D. and Penuli, A., "On the Development of Reactive Systems," Apt, K.R., (ed.), *Logics and Models of Concurrent Systems*, Vol. F13, pp. 477–498. NATO ASI Series (January 1985).

Harel, D., Lachover, H., Naamad, A., Pneuli, A., Politi, M., Sherman, R. and Shtul Trauring, A., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *Proceedings of the 10th IEEE International Conference on Software Engineering*, pp. 396–406, Singapore (April 1988).

Harel, D., "On Visual Formalisms," *Communications of the ACM*, pp. 514–530 (May 1988).

Hartley, D., "The Use of Structured Methods in the Development of Large Software-Based Avionics Systems," *Proceedings of AIAA/IEEE 6th Digital Avionics Conference*, Baltimore, Maryland (1984).

Hekmatpour, S., "Experience with Evolutionary Prototyping in a Large Software System," *ACM SIGSOFT Software Engineering Notes*, Vol. 12, No. 1, pp 38–41 (January 1987).

Holliday, M.A. and Vernon, M.K., "A Generalized Timed Petri Net Model for Performance Analysis," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, pp. 1297–1310 (December 1987).

Houghton, Jr., R.C. and Wallace, D.R., "Characteristics and Functions of Software Engineering Environments: An Overview," *ACM SIGSOFT Software Engineering Notes*, Vol. 12, No. 1, pp 68–84 (January 1987).

Jackson, K. and Simpson, H.R., "MASCOT—A Modular Approach to Software Construction Operation and Test," Technical Report, Royal Radar Establishment (RRE) Technical Note 778 (October 1975).

Jahanian, F. and Mok, A.K., "Safety analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, pp 890–904 (September 1986).

Jahanian, F., Lee, R. and Mok, A.K., "Semantics of Modechart in Real Time Logic," *The Proceedings of the 2lth Annual Hawaii International Conference on System Sciences (HICSS-21)*, Hawaii, Vol. II: Software, pp. 252–261 (January 1988).

Kelly, J.C., "A Comparison on Four Design Methods for Real-Time Systems," *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California, pp. 238–252 (May 1987).

Kenny, K.B. and Lin, K.-J., "Building Flexible Real-Time Systems Using the Flex Language," *Computer*, pp. 70–78 (May 1991).

Lauer, P.E. and Shields, M.W., "COSY: An Environment for Development and Analysis of Concurrent and Distributed Systems," *The Proceedings of the Symposium on Software Engineering Environments*, Lahnstein, Germany (June 1980).

Leveson, N.G. and Stolzy, J.L., "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, pp. 386–397 (March 1987).

Luckham, D.C. and von Henke, F.W., "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, pp. 9–22 (March 1985).

Luckham, D.C., Neff, R. and Rosenblum, D.S., "An Environment for Ada Software Development Based on Formal Specification: Status and Development Plan," *Ada Letters*, Vol. VII, No. 3, pp. 94–106 (May/June 1987).

Mantei, M.M. and Teorey, T.J., "Cost/Benefit Analysis for Incorporative Human Factors in The Software Lifecycle," *Communications of the ACM*, Vol. 31, No. 4, pp. 428–439 (April 1988).

Marcus, M., Sattley, K., Schaffner, S.C. and Albert, E., "DAPSE: A Distributed Ada Programming Support Environment," *The Proceedings of the IEEE 2nd International Conference on Ada Applications and Environments*, pp. 115-125 (1986).

Molloy, M.K., "Discrete Time Stochastic Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 417-423 (April 1985).

Murata, T. and Zhang, D., "A High-Level Petri Net Model for Parallel Interpretation of Logic Programs," *The Proceedings of the 1986 Conference on Computer Languages*, pp. 123-132 (October 1986).

Murata, T. and Komoda, N., "Liveness Analysis of Sequence Control Specifications Described in Capacity Designated Petri Net Using Reduction," *The Proceedings of the 1987 IEEE International Conference on Robotics & Automation*, Raleigh, North carolina, pp. 1960-1965 (March 1987).

Murata, T. and Zhang, D., "A Predicate-Transition Net Model for Parallel Interpretation of Logic Programs," *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, pp. 481-497 (April 1988).

Murata, T., Shenker, B. and Shatz, S.M., "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Transactions on Software Engineering*, pp. 314-326 (March 1989).

Naylor, A.W. and Volz, R.A., "Design of Integrated Manufacturing System Control Software," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 6, pp. 881-897 (November/December 1987).

Nelson, R.A., Haibt, L.M. and Sheridan, P.B., "Casting Petri Nets into Programs," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp. 590-602 (September 1983).

Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058 (December 1972).

Peterka, G. and Murata, T., "Proof Procedure and Answer Extraction in Petri Net Model of Logic Programs," *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 2, pp. 209-217 (February 1989).

Peters, L.J. and Tripp, L.L., "Comparing Software Design Methodologies," *Datamation*, Vol. 23, No. 11, pp. 89-94 (November 1977).

Peterson, J.L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ (1981).

Privitera, J.P., "Ada Design Language for the structured Design Methodology," *Proceedings of the AdaTEC Conference*, pp. 76-90 (October 1982).

Ramamoorthy, C.V., Tsai, W., Yamaura, T. and Bhide, A., "Metrics Guided Methodology," *Proceedings of COMPSAC'85: 9th Conference on Software Applications*, Chicago, Illinois, pp. 111-120 (October 1985).

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

Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of the 9th International Conference on Software Engineering (a Reprint of 1970 article)*, Monterey, California, pp. 328-338 (March 1987).

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

Simpson, H.R. and Jackson, K., "Process Synchronization in MASCOT," *The Computer Journal*, Vol. 22, No. 4, pp. 332-345 (1979).

Simpson, H.R., "Act Parallel: Use MASCOT," *Computer Bulletin*, pp. 6-9 (March 1982).

Stankovic, J.A., "A Serious Problem for Next-Generation Systems," *Computer*, pp. 10-19 (October 1988).

Stotts, P.D., "The PFG Environment: Parallel Programming with Petri Net Semantics," *The Proceedings of the 21th Annual Hawaii International Conference on System Sciences (HICSS-21)*, Hawaii, Vol. II: Software, pp. 630-638 (January 1988).

Taylor, R.N., Belz, F.C., Clarke, L.A., Osterweil, L., Selby, R.W., Wileden, J.C., Wolf, A.L. and Young, M., "Foundations for The Arcadia Environment Architecture," *SIGPLAN Notices*, Vol. 24, No. 2, pp. 1-13 (February 1989).

Taylor, R.N. and Standish, T.A., "Steps to an Advanced Ada Programming Environment," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 302-310 (March 1985).

Teichroew, D., "The Development of Software Support Environments," *Proceedings Canadian Information Processing Society (CIPS), 1982 National Conference*, Saskatchewan (1982).

The United States Department of Defense, *Ada Programming Language (ANSI/MIL-STD-1815A)* (February 1983).

Ward, P.T., "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, pp. 198-210 (February 1986).

Ward, P.T., "Embedded Behavior Pattern Languages: A Contribution to a Taxonomy of CASE Languages," *The Proceedings of the 2lth Annual Hawaii International Conference on System Sciences (HICSS-21)*, Hawaii, Vol. II: Software, pp. 273–284 (January 1988).

Watson, A.G., "Petri Net Topologies for a Specification Language," Master's thesis, University of Witwatersrand (1987).

Webster, D.E., "Mapping the Design Information Representation Terrain," Technical Report MCC-STP-367-88, Microelectronics and Computer Technology Corporation (November 1987).

White, S.M., *A Pragmatic Formal Method for Computer System Definition*, PhD thesis, Polytechnic University (June 1987).

Wolff, J.G., "The Management of Risk in System Development: 'Project S' and the 'New Spiral Model'," *Software Engineering Journal*, pp. 134–142 (May 1989).

Yau, S.S. and Shatz, S.M., "On Communication in The Design of Software Components of Distributed Computer Systems," *Proceedings of the 3rd Conference on Distributed Computing Systems*, pp. 280–287 (1982).

Yau, S.S. and Caglayan, M.U., "Distributed Software System Design Representation Using Modified Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, pp. 733–745 (November 1983).