Real-Time Systems Design Methodologies: An Introduction and a Survey

Krishna M. Kavi and Seung-Min Yang
The University of Texas at Arlington, Computer Science Engineering Department, Arlington, Texas

In this article we describe examples of real-time systems in an attempt to characterize such systems. We address the issues as they relate to real-time embedded software systems, and issues that distinguish them from other software systems. The key feature of real-time systems is the timely response requirement. This often implies concurrent processing. The critical nature of many real-time applications necessitate fault-tolerant implementations. Future real-time systems will be more complex and would require distributed implementations. Although several design methods and tools have been proposed, they fall short in meeting all the requirements. We include a "wish-list" of desirable features of future software tools.

1. INTRODUCTION

As the environments in which real-time software systems are embedded become more complex, specifying such systems precisely, describing the interactions among the tasks in such systems and with their environment, and designing and verifying the correctness of the code become more difficult. Traditional real-time systems have been designed using ad hoc techniques, and have often been hand coded in assembly languages. Frequently, these designs relied on the primitives for concurrent operations provided by the run-time executives. Modern concurrent languages such as Ada incorporate the necessary concurrent constructs, thus obviating the reliance on run-time executives but necessitating a new tactic for dealing with concurrent activities of real-time systems.

A number of software design methodologies have been advocated for the identification of parallel activities and their implementation in Ada (and other languages). These methods are based on functional decompositions, data flow diagrams, and the identification of I/O dependencies, time-criticalities, and periodic nature of tasks. Some approaches advocate decompositions based on real-life entities (that is, clearly recognizable units such as hardware components). However, most of these methodologies are based on heuristics, in that a thorough knowledge of the intended system, choice of programming language, and execution characteristics of the underlying architecture play key roles in the decomposition of the system into concurrent tasks. Formal verification of the resulting systems is often ignored: "When it comes to the implementation of specifications formally, one does not do it by writing programs and then trying to prove that they meet the specifications. Instead, one constructs correct programs in small steps—each step taking the specification and producing something that is a bit closer to the final program" [1]. Formal specification and verification approaches to real-time systems are covered by Ostroff elsewhere in this issue.

At the other extreme, some formal specification and verification methods strive for fool-proof or error-free designs. A proof is only a demonstration that one formal statement follows from another, and the validity of a statement depends on the validity of the statement from which it is derived. Thus, a balance between the rigorous nature of formal verifications and the informal nature of program implementations is desired.

In this survey, we will describe some simple examples of real-time systems in an attempt to characterize such systems. We will address the issues as they relate to real-time embedded software systems, and issues that distinguish them from other software systems. Next we will summarize a few of the design methodologies and three design tools. Finally, we will list major research issues along with a "wish list" of desirable features for future software tools.

Address correspondence to Professor Krishna M. Kavi, The University of Texas, Computer Science Engineering Department, Arlington, TX 76019.

0164-1212/92/$5.00

2. REAL-TIME SYSTEMS AND DESIGN ISSUES

Real-time systems have several features that distinguish them from other computerized systems. In an attempt to bring out these features and elucidate the problem domain, we describe some examples of real-time applications. Detailed specifications for these examples are readily available, and several researchers have used these examples to illustrate their design methodologies. We will refer to these examples whenever necessary. We will then characterize real-time systems and identify important design issues.

2.1 Examples of Real-Time Systems

The term real-time system covers many applications, from factory automation to nuclear power plants, from automobile engine control to space shuttle and aircraft avionics, and from robotics to command-and-control systems. One simple example is a system to control a conveyer belt, as shown in Figure 1 [2]. The goal of the control system is to maintain a constant motor speed and to protect it against dangerous load changes. The system has two interface components, one to acquire the current speed of the conveyer belt from the speed counter, and the other to accelerate or decelerate the motor. The decision (viz., adjustment of speed or no change), is made by the control component.

In this subsection we will describe six other examples of real-time systems that are frequently used by researchers to illustrate their methodologies.

2.1.1 Robot controller system. A robot controller system [3] (at General Electric's Industrial Electronics Development Laboratory) controls up to six axes of motion and interacts with digital I/O sensors. Control of axes and I/O is effected by a program initiated from a control panel consisting of a number of push buttons and a selector switch for program selection (Figure 2). The state transition diagram for the controller is shown in Figure 3. Error conditions are ignored in this description.

When the POWER ON button is pressed, the system enters the POWERING UP state. On successful completion of the power up sequence, the system enters MANUAL state. The operator may then select a program using the program select rotary switch to indicate the desired program number. Pressing RUN initiates execution of the program selected, and the system transitions into RUNNING state. Execution of the program may be suspended by pressing STOP, at which time the system enters the SUSPENDED state. The operator may then resume program execution by pressing RUN, returning the system to RUNNING state, or terminate the program by pressing END, in which case the program enters the TERMINATING state. When the program finally terminates, the system returns to the MANUAL state.

2.1.2 Bottle-filling system. The bottle-filling system [4] has several bottling lines attached to a single vat that dispenses liquid. Each line may fill a different-sized bottle. Figure 4 shows one of the bottling lines. The major control tasks manage the liquid level in the vat and the liquid's hydrogen-ion content (pH level), the movement of bottles on each line, and interface with human operators.

The liquid level in the vat is controlled by monitoring the output of a level sensor and opening or closing the liquid input valve to keep the level within certain

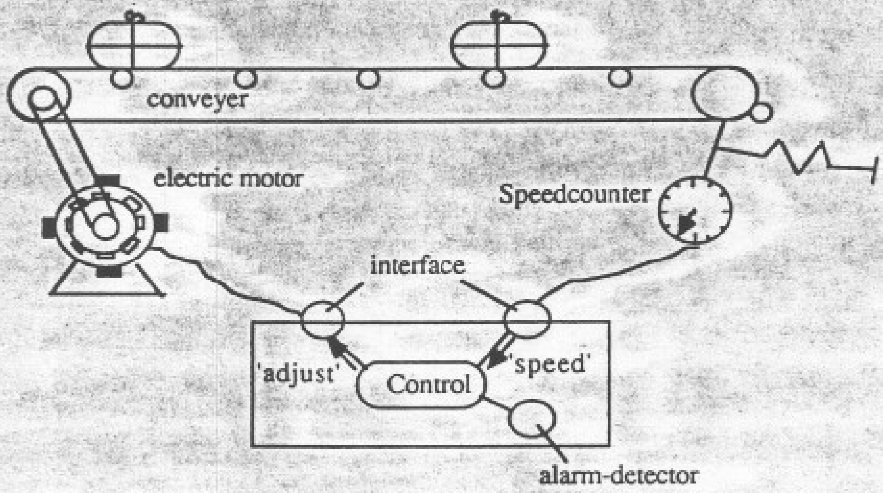

Figure 1. Conveyor control system.

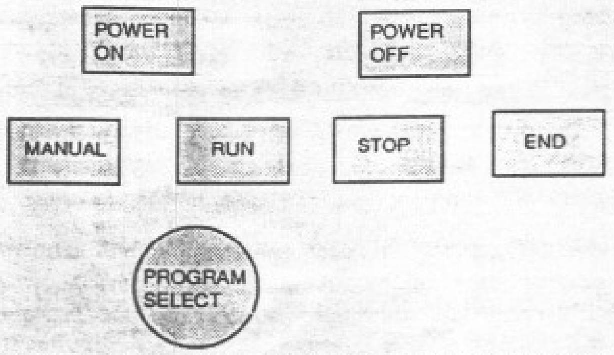

Figure 2. Control panel of robot controller system (adapted from [3]).

limits. The liquid's pH content is monitored by a pH sensor; whenever the pH creeps above a certain threshold, a pH control valve releases small quantities of a neutralizing liquid to correct the pH level.

On each bottling line, a bottle is released down a chute onto a scale platform that activates a bottle-contact sensor. The bottle-filling valve opens, releasing a mass of liquid proportional to the bottle size; the system monitors the bottle's weight through the scale to determine when the bottle-filling valve should be shut off. The filled bottle is labeled and capped automatically and the line operator removes the bottle from the scale. Removing the bottle resets the bottle-contact sensor and the weight on the scale, and lets the next bottle be released.

The operator of each bottling line can manually signal the individual line to start or stop and can change the bottle size. A line can only start operation from stopped status when the overall system area is enabled, the bottle-contact sensor is off, and the scale is set to zero. Each operator requires a display of the the on-offline status and current bottle size. The area supervisor can enable or disable the overall system area, including all bottling lines, and change the pH setpoint. If the measured pH cannot be kept automatically within a threshold, the entire area is automatically disabled. The area supervisor must stabilize the pH manually and then reenable the system operation. The area supervisor requires a display of pH level, vat level, and status of individual lines.

2.1.3 Remote temperature sensor problem. A software-driven, remote temperature sensor [5, 6] obtains temperature readings from a number of furnaces via a digital thermometer and reports the temperature values to a host computer. The host sends control packets (CP) to the remote sensor. Each CP includes a furnace number (0 . . . 15) and a reading interval (10 . . . 99 seconds). The remote sensor acknowledges each CP (CP-Ack or CP-Nak), and the sensor reads the temperature of that furnace periodically at the prescribed interval. The sensor manages up to 16 such simultaneous reading series for different furnaces. The sensor sends every temperature reading to the host in a data packet (DP), it then waits for either a DP-Ack or DP-Nak from the host. A DP-Nak signifies that the DP has been received incorrectly by the host and causes the sensor to resend it. If neither a DP-Ack nor a DP-Nak arrives within two seconds, it is assumed that a transmission error has occurred, and the DP is resent.

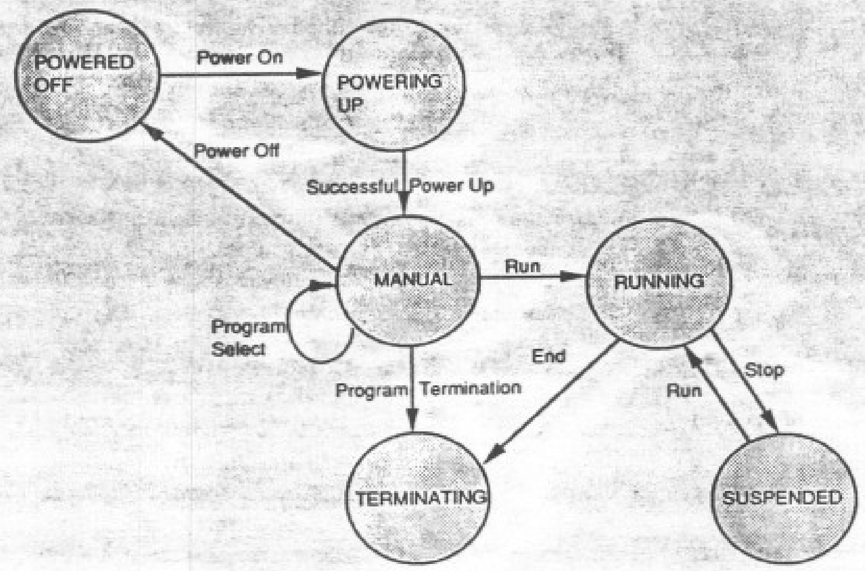

Figure 3. State transition diagram of the robot controller system (adapted from [3]).

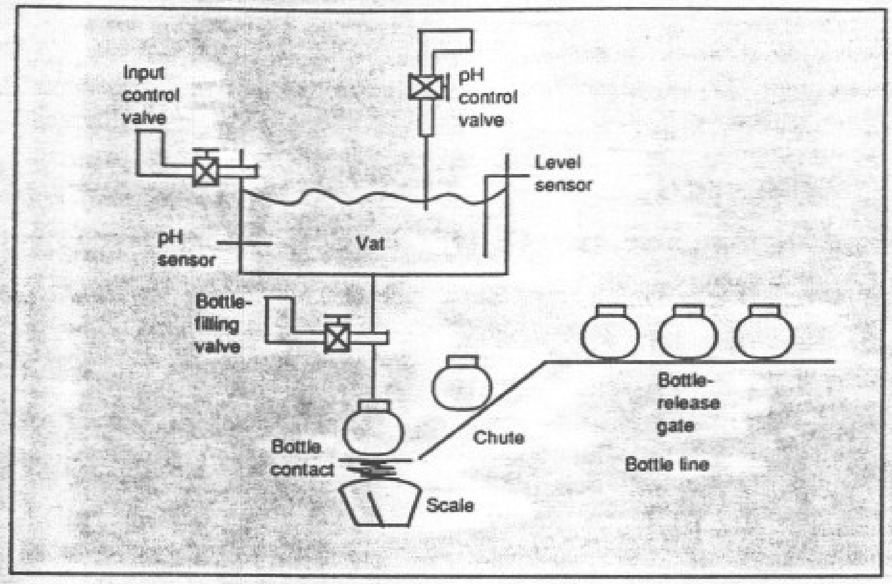

Figure 4. The production like process in the bottle-filling system (adapted from [4]).

The sensor is equipped with one thermometer, which, upon request from the sensor software, stores the temperature of a specified furnace in a designated hardware buffer and then creates an interrupt. Only one thermometer request can be handled at a time.

2.1.4 Cruise control. The cruise control function [7, 8] takes over the task of maintaining a constant speed when so commanded by the driver. The driver must be able to enter several commands, including ACTIVATE, DEACTIVATE, START ACCELERATING, STOP ACCELERATING, and RESUME. The cruise control function can be operated any time the engine is running and the transmission is in top gear. When the driver presses ACTIVATE, the system selects the current speed, but only if it is at least 30 miles per hour, and holds the car at that speed. DEACTIVATE returns control to the driver regardless of any other commands. START ACCELERATING causes the system to accelerate the car at a comfortable rate until STOP ACCELERATING occurs and the system holds the car at this new speed. RESUME returns the car to the speed selected before braking or gear shifting.

The driver must be able to increase the speed at any time by depressing the accelerator pedal or reduce the speed by depressing the braking pedal. Thus, the driver may go faster than the cruise control setting simply by depressing the accelerator pedal far enough. When the pedal is released, the system will regain control. Any time the brake pedal is depressed or the transmission shifts out of top gear, the system must be inactive. When the brake is released, the transmission is back in top gear, and RESUME is pressed, the system returns the car to the previously selected speed. However, if a DEACTIVATE has occurred in the intervening time, RESUME does nothing.

The speed measurement, which is based on the size and count of rotations of the tire, can be calibrated using START MEASURED MILE and STOP MEASURED MILE. They are only effective when cruise control is inactive.

2.1.5 Simplified unmanned vehicle system. The simplified unmanned vehicle system (SUVS) [9] controls a vehicle with no assistance from a human driver. It periodically receives data from sensors such as the speedometer, temperature sensor, and direction sensor, and controls the vehicle by generating appropriate signals for the actuator devices such as the accelerator, brake, and steering wheel. It changes the direction of the car when curves are encountered, reduces speed when an obstacle appears, increases speed after the obstacle is past, stops the car within a specified time to avoid a collision, and so on. Decisions are made based on the current inputs from the sensors and the current status of the road and the vehicle. Decisions (or responses) must be made within a specified time.

2.1.6 Ballistic missile defense system. The goal of a ballistic missile defense (BMD) system [10] is to either prevent the enemy from damaging a defended region, or extract an unacceptably high price for defeat of the system. A BMD system typically consists of sensors to observe and collect data about a threat, a kill mechanism to render the threat ineffective, and a computational element designed to ensure a cooperative effect from the component parts. A typical BMD scenario is shown in Figure 5.

2.2 Characteristics of Real-Time Systems

As can be seen from the above examples, real-time systems have clearly identifiable features. They consist of a set of sensors that acquire data from the environment. The data is processed and the output is sent to a set of actuators, which produce the desired changes in the system. The real-time software system consists of a set of concurrent threads for processing inputs (sensor data) and effecting the output (actuators). These threads can be characterized as one input-one output, one input-multiple outputs, multiple inputs-one output, and multiple inputs-multiple outputs. In some applications the concurrent processing threads operate asynchronously, while in other applications they must follow strict synchronization rules. The different timing requirements of real-time systems may place different constraints on the processing:

1. Response time deadline: the system must respond to the environment within a specified time after input (or stimuli) is recognized.
2. Validity of data: in some cases, the validity of the input (or output) is a function of time. That is, some stimulus (and the corresponding response) become obsolete with time, and the interval for which data is valid must be accounted for in processing requirements.
3. Periodic execution: in many control systems, sensors collect data at predetermined time intervals, and the real-time system must process the data and output needed responses.
4. Coordinating inputs and outputs: in some applications (e.g., unmanned vehicle system), input data from various sensors need to be synchronized. Otherwise, decisions would be made based on inconsistent information. Output data to actuators also need to be well synchronized.

2.3 Design Issues

The issues in the design of real-time systems are different from other systems primarily because of the presence of concurrent events and the temporal requirements that exist between inputs and outputs, as described in the previous section. Design issues as they relate to real-time systems have been thoroughly investigated and reported. We describe three of the important issues here.

2.3.1 Decomposition. Since real-time systems operate in the presence of concurrent events, it is only

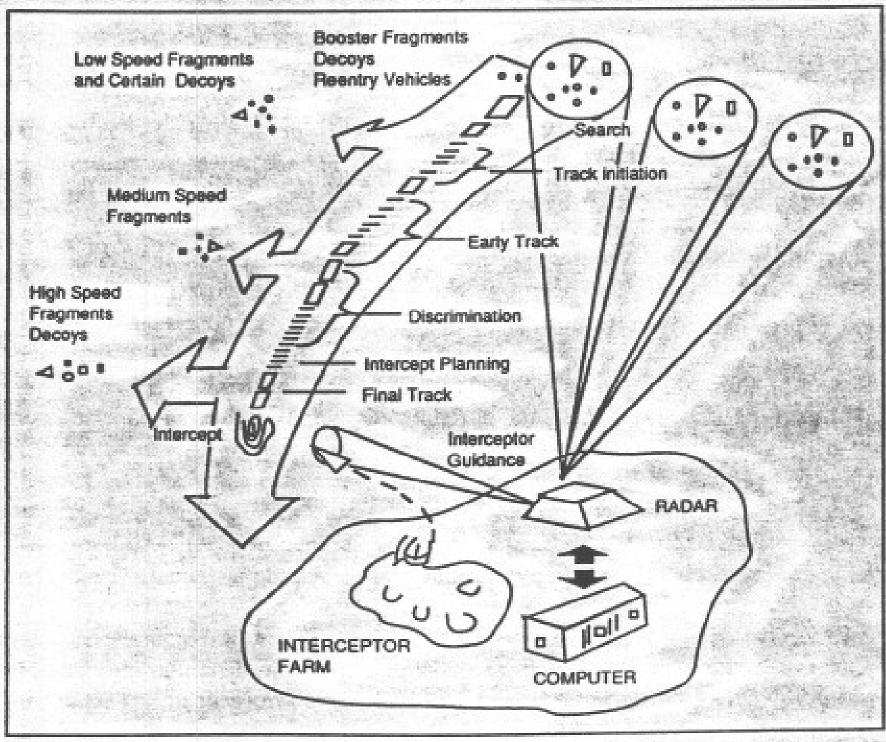

Figure 5. Typical BMD mission (adapted from [10]).

natural to design the software as a set of concurrent processes. The proper decomposition not only helps in coping with system complexity, but also in increasing concurrent activity. Decomposition may also aid in the predictability of the timing requirements. Some decomposition approaches are discussed in section 3 below.

2.3.2 Interface. The interface between the components of the system (including sensors and actuators) is in the form of communication and synchronization. Communication can be achieved either through shared data or message passing, and the choice depends on the implementation language and processor architecture. Tight synchronization among the components may be desirable even if the application does not require such synchronization. Synchronization leads to more predictability and easier detection of error conditions. On the other hand, delays caused by synchronization lead to performance degradations.

2.3.3 Timing. A primary design concern of real-time systems is how to reflect the timing constraints. The system decomposition should include both functional and temporal requirements. Furthermore, the temporal behavior of the components should be analyzed for completeness (i.e., whether all requirements are reflected in design), correctness (i.e., whether there is any conflict or inconsistency in design), and feasibility (i.e., whether it is feasible to achieve the requirements).

Timeout (and timer interrupt) is a basic form of monitoring the temporal behavior of the system. Although often ignored, the selection of proper timeout signals is important. The choice of the timeout interval plays a significant role in the synchronization of concurrent activities and timely adjustments to compensate for deviations.

2.3.4 Other issues. Other important issues, although not specific to real-time systems, include:

1. Automated support for the design from requirement specification to implementation.
2. Means for reflecting the processor architecture during the software development life-cycle. The reliability and timing analyses become meaningful only if the characteristics of the processor are considered.
3. Task allocation and scheduling. Issues regarding the selection of scheduling discipline to meet timing requirements are not covered here.
4. Fault tolerance. Issues regarding the amount and type of redundancy, error detection, location, and recovery techniques are not covered here.
5. Design support tools. The design of any complex software system can benefit greatly from interactive (preferably graphical) debugging and monitoring tools.

3. DESIGN METHODOLOGIES

The fundamental aim of all real-time designers is to produce a "good" software system. By good we mean a system that not only satisfies the functional and performance requirements, but that is reliable and easy to maintain. Another implied requirement of software systems in general, and real-time systems in particular, is that the system be produced on time at a reasonable cost. Finally, the system should be adaptable to changing (enhanced) requirements and new hardware units. However, good designs often require experience, good engineering practices, and good tools to aid the design process.

As with any design, a real-time system must start with the specification of requirements. From these requirements, basic system parameters can be extracted. The parameters may either directly or indirectly influence the software system. It is sometimes necessary to simulate the desired system to verify performance requirements. Based on these analyses, initial decisions on the choice of hardware (for example, whether to use a dedicated hardware unit to cope with the stringent processing requirements) and software can be made. In terms of software, the initial decisions relate to the choice of the operating environment (for example, distributed or shared multiprocessing), run-time system, programming language, and software development environment.

Although often neglected, the software designer should be involved during the choice of the hardware configuration. Once frozen, the hardware configuration (the processor, memory management, I/O processing), significantly affects the software design. For example, the reliability of the system may require the inclusion of software-level exception handling, voting, or reconfiguration of hardware units. The performance may dictate the granularity of the concurrent tasks.

3.1 System Decomposition Methods

Real-time software is typically designed as a set of communicating concurrent processes. In many cases, these processes execute asynchronously in order to accommodate devices and sensors with varying speeds. The processes communicate either to exchange data or control information at synchronization points. Thus, the decomposition of the real-time system into component processes is a major issue. System decomposition with respect to time, space, size, control flow, data flow, subfunctions, objects, and data structures has been thoroughly investigated by software engineers over the past three decades. In the next several sections we will study the applicability of some of these approaches to structuring real-time software systems.

3.1.1 Structured designs and functional decompositions. These approaches (see, for example, Cameron [11], Jackson [12]) are similar to traditional structured approaches. Often the designer starts with an analysis of system functionality to obtain a data flow diagram of the system. The system is then hierarchically decomposed into subsystems. Entity-relationship diagrams are also used to identify the relationship between the (data) entities. A data dictionary is often created to capture the data flows and data entities.

When designing software for a real-time system containing multiple processors, decomposition may require two phases: subsystems and tasks. A subsystem consists of one of more tasks, and these tasks usually run on a single processor. Structuring of the application into subsystems is often based on functional decomposition to achieve high cohesion within a subsystem and weak coupling between the subsystems. Functional cohesion refers to the similarity of the functions performed. Functionally similar activities may require extensive data communication, and they should be grouped together into a single subsystem (or even into a single task) to limit communication overheads. Activities may be related in other ways. For example, several activities may be triggered by the same event (temporal cohesion), or the activities must be executed sequentially in a prescribed order (sequential cohesion). It may be desirable to group together activities with such dependencies. The degree of coupling among the modules (and tasks within a module) determines the amount of concurrency and delays due to synchronization. It is desirable to decrease the coupling to increase the asynchronous concurrency in the system.

The subsystems are then decomposed into concurrent tasks. The data flow diagram of the system is studied to identify the transformations that can be executed concurrently and those that must be executed sequentially. A number of guidelines have been suggested for determining if a transformation should be a separate task or grouped with other transformations into one task. For example, Gomaa [13-15] suggests the following:

1. I/O dependency: since transformations that depend on input or output must run at the speed of the I/O device, separate tasks must be used for such transformations. It may be desirable to have a different task for each asynchronous I/O device (at least for each device type).
2. User interface dependency: like with transformations that depend on I/O, transformations that depend on user interactions should be structured into separate asynchronous tasks.
3. Periodic execution: transformations that are executed regularly at predetermined intervals should be designed as separate tasks to facilitate scheduling. Also, independent tasks are used for periodic I/O activities.
4. Time-critical functions: activities that have stringent response-time requirements should be separated into tasks to permit priority scheduling. Likewise, non-time-critical but computationally intensive computations should be assigned low priority tasks.

To illustrate this approach, consider the remote temperature sensor problem (section 2.1.3). According to the guidelines, a separate process should be allocated for each external device. This leads to two processes, one to interface with the digital thermometer (which reads the temperature of a furnace) and the other to interface with the host computer. The periodic nature of temperature sensing may indicate that a separate task be allocated for recording the temperature of each furnace.

In the case of the cruise control problem (section 2.1.4), the functional decomposition may lead to the creation of a separate task to process each of the inputs from the accelerator (its current state), the wheel (to compute the current speed), the brake (to revert to manual control), and so on, and a separate process for the output (to control the throttle). The computational part of the system is decomposed based on functional cohesion. For example, we may use the following processes: *Obtain-Desired-Speed*, *Get-Current-Speed*, *Calculate-Throttle-Value*. It may be necessary to further decompose some of these tasks into concurrent threads to minimize coupling among the activities.

3.1.2 Object-oriented methodology. Object-oriented methodology [7] differs from functional approaches primarily in the manner the modules/subsystems or tasks are identified. In functional methods, each module represents a major transformation. In object-oriented methods, each module is responsible for managing a major object of the system. In other words, in a functional approach, several modules may operate on an object (e.g., a data item), performing different transformations. In an object-oriented approach, a single module is allowed to operate on the object, performing all the transformations.

Note that there is no agreement on what object orientation really means. Object-oriented language designers firmly believe that inheritance, polymorphism, and dynamic binding are essential. According to Meyer [16], the following properties distinguish object-oriented languages from procedural and functional languages:

1. Object-based modular structure: systems are modularized on the basis of their data structures.
2. Data abstraction: objects should be described as implementations of abstract data types.
3. Automatic memory management: unused objects should be deallocated by the underlying language system, without programmer intervention.
4. Classes: every nonsimple type is module and every high-level module is a type.
5. Inheritance: a class may be defined as an extension or restriction of another.
6. Polymorphism and dynamic binding: program entities should be permitted to refer to objects of more than one class and operations should be permitted to have different realizations in different classes.
7. Multiple and repeated inheritance: it should be possible to declare a class as heir to more than one class and more than once to the same class.

Languages that meet the first four criteria are regarded as object based. Languages must meet all seven characteristics to be regarded as object oriented.

The two primary properties of the object-oriented approach used by advocates of this method are abstraction and information hiding. An abstraction is a simplified description of a system emphasizing certain aspects of the system while hiding other details. The information-hiding principle requires that design decisions about a subsystem be local and invisible to other subsystems. Proponents claim that these two properties together lead to the following advantages: the objects match closely to the real system, and the effects of changes are localized. In reality (based on Meyer's [16] criteria), these approaches should be considered object-based methodologies.

Regardless, let us examine the steps in object-oriented development. The first step is to identify the objects and their attributes. Typically, this implies the identification of the major actors, agents, and servers in the problem space and the definition of their roles in the system. Next, the transformations that must be performed on the objects and the services (if any) offered by the objects must be identified. To satisfy the information-hiding principle, it is necessary to define the interfaces to the object that are visible. Finally, the services and transformations associated with each object are implemented. The actual implementation details should not be visible to other objects.

As may be seen from the above discussion, during the initial phases of the design one can use transformations and services common to objects in defining object classes and class hierarchies. In turn, this can lead to the use of inheritance and polymorphism properties for defining subclasses, and transformations on different data types.

For example, consider the bottle-filling problem (section 2.1.2). The behavior of the different lines (for different bottle sizes) can be modeled using inheritance. Thus, the operation of filling bottles can be captured into a class hierarchy, and each instance of the class is instantiated for a specific bottle size. The inheritance then automatically instantiates the appropriate values for controlling the process of filling the bottle, moving the bottles, releasing a new (empty) bottle, and so on.

As another example to illustrate the differences between object-oriented and structured approaches, consider the cruise control problem (section 2.1.4). The decomposition resulting from the structured approach was discussed earlier. In the object-oriented approach, a separate module/object/task is allocated to deal with a clearly identifiable entity (or object). For example, a separate task will be assigned to control each of the devices—the accelerator, throttle, wheel, etc. These tasks are responsible for all computations (or transformations) required to control that device.

The object-oriented approach seems natural for real-time systems, with sensor and actuator objects and control objects in between these. Moreover, reusability and maintainability are, among others, the positive attributes of the approach, because of their capability of abstraction, information hiding, and inheritance. However, the approach has some drawbacks. First, the system often results in too many objects. For example, in telephone switching systems we may have thousands of subscriber and link objects. This requires clustering of objects to avoid the overhead, which is a design burden. Second, and more importantly, it is usually harder to predict the timing behavior of the system with the object-oriented approach than with the functional approach. This is because transformation of the system (with which the timing constraints are associated) may consist of multiple threads among multiple objects. Although some work (e.g., [17]) has been done on object-oriented design and timing, especially predictability, this is a major area for future research.

3.1.3 Virtual machines. Hierarchically layered virtual machines can be used to deal with the design of complex real-time systems. Typically, this approach involves the creation of a set of layered abstractions that simplify the design process by deferring implementation details to lower level machines. If properly used, this approach promotes a high degree of maintainability and portability, since it is possible to replace any layer with a new one [18]. In many practical systems, this may not be easy to accomplish because of the dependencies between the abstractions at various layers.

At the higher levels of the abstractions, the virtual machine representation of a real-time system usually consists of processes, channels, buffers, shared pools, synchronization events, and interrupts. Processes represent agents that implement activities or functions. Processes communicate with each other using channels, buffers, and pools, and synchronization is achieved via events. Channels usually connect two processes (one to one), and they are either uni- or bidirectional. Buffers with channels permit asynchronous communication. Shared pools are used to represent communication among several processes (one to many and many to many). *Wait*, *Signal*, and *Interrupt* are among the common types of synchronization events. The decomposition of the system into processes can be based either on functional or object-oriented approaches.

As opposed to hierarchically layered virtual machines, some methods advocate vertical layers in dealing with complex systems [19]. In a vertical partition, each subsystem (or partition) encapsulates all the hierarchical layers of a traditional virtual machine. Each layer within a partition can be optimized for the partition; in a traditional layered approach, the generality forces less-than-optimum implementations.

3.1.4 Other methods. Sanden [20, 21] advocates an approach to the identification of tasks (particularly when Ada is the implementation language for real-time embedded systems) known as entity-life model. In this approach, the designer starts looking for complex yet purely sequential behavior patterns in the problem space. The objective is to capture the functionality of the problem in as few subsystems as possible. Each functionality can be as complex as required (describing sequential processing). However, it is difficult to define exactly what entity-life modeling means. It can only be vaguely related to the guidelines suggested by Sanden [20, 21] that we structure (define concurrent processes, packages, tasks, communication, and synchronization) the software based on "real-life" entities and activities. This approach advocates decomposition based on the constructs provided by the implementation language. For example, the synchronous nature of Ada rendezvous and the use of Ada tasks must be taken into account during the decomposition. It is our observation that the entity-life approach relies on both functional and object-oriented approaches. In some applications, the entity-life approach derives more parallelism than the object-oriented approach and less parallelism than functional decomposition. Sanden contends that the functional approaches produce more parallelism (finer grain) than can be used in realistic implementations.

Consider the temperature-sensing example (section 2.1.3). Entity-life modeling leads to separate task for each of the furnaces. Each task records the furnace temperature, sends the value read, and idles until the next temperature-reading interval. Compare this with the structured approach, which advocates a single task to read the temperature of each furnace at specified intervals. When using Ada, the use of separate task for each furnace requires a complex "intermediary" task to accept the temperature readings form each of the furnaces at specified intervals.

As can be seen from the methodologies discussed, there are no formal or strict rules governing the decompositions, and in most cases, a good decomposition results from a clear understanding of the requirements, the high-level language, and the processor architecture chosen for the implementation. It is important to use some formal specification and verification methods, even if these formalisms do not guarantee fool-proof or error-free implementation. They often will bring out shortcomings of the implementations.

Strict adherence to any one decomposition methodology may lead to designs that are either inefficient or unnatural. Any good software engineering design environment (and associated tools) should permit multiple methodologies. The entity-life model encourages the use of shared data to minimize the number of tasks. This is primarily because of the use of packages in Ada to encapsulate tasks and the shared data.

3.2 Process Abstractions

Once subsystems (objects, tasks, packages) are identified, the activities (transformations, functions) to be performed by the subsystems and the interfaces among the subsystems must be defined. Concurrent tasks need to communicate with each other to exchange data (data flows), control information (event flows), and share data (data stores). It is common to use graphic abstractions to represent the operations of subsystems. In this section we will discuss some such abstractions available to the real-time designer.

3.2.1 Finite state machines. A finite state machine (FSM) is commonly used to describe the behavior of a process by listing all the possible states of the process, the inputs (either control or data) that move the process from one state to another, and the outputs produced by the process in each state. The "next state" of the process depends on the input and the current state. Finite state machines are usually represented graphically as state transition diagrams. Each possible state of the process is drawn as a circle (the name of the state is usually written inside the circle), and the directed arcs between states represent state transitions. Arcs are often labeled with the inputs that cause the transitions. In some cases, outputs are also listed on arcs. When using FSMs as an abstraction, the states represent completion of an activity. The inputs represent events or conditions. The functions (transformations) performed by the process are abstracted into the states, implying that the process performs operations of transformations while residing in a state. The process accepts inputs and transitions to the next state upon completion of the activities associated with the current state.

3.2.2 Extended data flow diagrams. A data flow diagram for a process consists of three elements [22, 23]:

1. Data transformations: the functions carried out by the process. These nodes are usually represented as circles or boxes.
2. Flows: directed arcs between the nodes represent data flows between transformations.
3. Data stores: usually represented by two bars; act as repositories of data.

Although data flow diagrams have been used extensively, they are not adequate for representing real-time processes. Specifically, data flow diagrams cannot describe external stimuli or the dependency of transformations on time. To overcome these limitations, Ward [24] proposed some extensions. The extended data flow diagrams contain the following elements:

1. Transformations: there are two types of transformations—data and control. Data transformations (represented as solid circles) describe the functions of a process and are identical to transformations in standard data flow diagrams. Control transformations (represented as dotted circles) define how the data transformations are activated and deactivated. The control transformations must be associated with a state machine to represent the sequence of actions to be performed by the process.
2. Flows: as in standard data flow diagrams, there are arcs (represented as solid arcs) to represent flow of data among data transformations. The flow can indicate either discrete data (solid, single-arrow arcs) or continuous data (solid, double-arrow arcs). A continuous flow is an abstraction or a real-world quantity such as a temperature of pressure monitored by the system. In addition to data flows, control flows connect the control transformations to data transformations. A dotted, single-arrow arc represents a signal. A signal is used to indicate the occurrence of an event. A dotted arc with double arrows represents activations. A dotted arc with reversed arrowheads represents deactivation.
3. Stores: data stores are represented by a pair of parallel straight lines. Buffers are represented by a pair of dotted parallel lines.
4. State machine: as stated earlier, a state machine is used to describe the control transformations of the system.

Although extended data flow diagrams attempt to address the representation of control flow, the use of state machines results in a nonuniform model.

3.2.3 Petri nets. Petri nets have gained considerable popularity over the past decade because of their graphic nature and their ability to represent asynchronous concurrency. Petri nets consist of two types of nodes called places (represented as circles) and transitions (represented as bars). Transitions represent activities, functions, or transformations, while places represent events such as signals and completions of activities. The occurrence of an event is indicated by placing a token in the appropriate place. Transitions are enabled (or start executing their operation) only when all its input places contain tokens. (See the article by Ostroff in this issue for a more detailed treatment of Petri nets and how they can be used for formal specification of real-time systems.)

3.2.4 Other process abstractions. Data flow graphs [25–28] are similar to Petri nets and can be used as executable process abstractions. The data flow graph models have their roots in data flow languages (such as VAL [29] and SISAL [30]) and data flow computers [31, 32]. Computations (or data transformation) are represented using actors (nodes in the graph). The execution of the data flow graphs is data driven—an actor is enabled only when all the necessary inputs are available. Data flow graphs represent asynchronous concurrency, since computations with no dependencies can be executed in parallel. In addition to data transformation and data flows, data flow graph models permit the representation of control using special actors (for example, disjunctive input and selective output) with control inputs. These actors select a subset of the inputs or produce a subset of outputs based on the value on the control input. For a more detailed description of the model, see references [25–27]. The data flow graphs, unlike the extended data flow diagrams, represent both the data and control transformations in a unified manner.

Hierarchical multistate machines (HMS) [33] are proposed to model multiple active states. The goal is to preserve "concurrency" even when dealing with higher-level abstractions. HMS permit the definition of a collective transition of a set of states. This mechanism is similar to Petri nets, where the concurrency is preserved by defining markings. HMS attempt to reduce the exponential growth in the number of possible states (and possible combinations of multistate transitions) by defining "objects" with sets. Objects represent data (or control) values. This should be contrasted with abstractions that define separate states for each possible data value.

Process abstractions are either synchronous or asynchronous in nature. Finite state machine-based models are synchronous. In most cases, real-time specifications can only be achieved by treating time as a state variable rather than as a function of the state transition. Petri nets and data flow graphs are asynchronous process models. Events that trigger actions are more natural in such asynchronous abstractions.

3.3 Implementation Methodologies

The graphic representation of tasks also identifies the interfaces to the tasks. Concurrent tasks need to communicate with each other to exchange data (data flows), share data (data stores) and control information (event flows). During the final stages of the real-time system design, the abstract representations of the processes and their interfaces are translated into programs in a chosen high-level language. This requires the design of data structures, control structures, task structures, and communication and synchronization structures among tasks. Obviously, the choices are constrained by the implementation language chosen.

When the language does not provide mechanisms for the definition and control of concurrent tasks, the designer is forced to rely on the primitives provided by a real-time executive. However, some modern languages (for example, Ada) have incorporated most of the primitives required for a concurrent system. It is necessary to understand the responsibility of a task (and hence the programmer who writes it) and the responsibility of the system (compiler, run-time system) in achieving correct and predictable execution of the system. It is important to understand the model of concurrency and communication offered by the language. If the language provides only synchronous communication, it may be necessary to create structures to simulate asynchronous execution (for example, by creating intermediary tasks). It is also necessary to understand if the communication is one to one or one to many. In Ada, for example, the calling task must know the name of the called task (and its entry points), but not vice versa.

The mechanisms for activating tasks are also different in different languages. Tasks may be active at all times, they may become active as soon as the procedures containing them are invoked (as in Ada), or they may be activated on demand. The activation of tasks at specified times is supported in some languages. However, in most cases, the programmer has to fine tune the time specification for the activation of tasks in meeting the requirements of the application. The run-time overhead increases with the delay in binding (activation) of tasks; the flexibility also increases with dynamic binding. Some languages permit the definition of exception handling. Unusual (and unexpected) events can be handled using these capabilities.

The information-hiding principle can be implemented easily if the language provides abstract data types, modules, or packages. Such data abstractions permit the definition of visible interfaces while hiding the implementations. Object-oriented languages also facilitate information hiding. Note, however, that the concept of inheritance may violate the information-hiding principle.

Often, embedded systems must deal with low-level I/O from sensors and instruments. Such I/O is necessarily machine dependent, and earlier languages tended to ignore I/O, relying instead on the support provided by the operating systems. More recent languages exhibit features specifically designed to deal with low-level I/O. A survey of languages and language constructs for real-time systems by Stoyenko can also be found elsewhere in this issue.

4. DESIGN TOOLS

Real-time design tools should permit the representation of concurrency, communication, synchronization, and timing constraints. Additionally, the tools should facilitate the analysis of the temporal characteristics for feasibility, predictability, and correctness. Although various computer-aided software engineering (CASE) tools are available for many applications, tools for design of real-time systems are still scarce. In this section we briefly review three existing tools: STATEMATE from i-Logix, Inc., computer-aided real-time design tools (CARDtools) from Ready Systems, and Software through Picture (StP) by Interactive Development Environments (IDE). A general review of CASE tools can be found in [34].

4.1 STATEMATE

STATEMATE is a set of graphically oriented tools 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 or hardware [35]. STATEMATE provides three different graphic or diagrammatic languages. Module charts represent the structural view of the system, activity charts represent functionality, and state charts represent behavior of the system. STATEMATE makes it possible to execute, debug, and analyze a system as specified using the graphic representations. Figure 6 illustrates the overall structure of STATEMATE.

4.2 CARDtools

CARDtools is an integrated set of tools designed for the development of complex real-time embedded systems [36]. The front-end design tools of CARDtools are designed with real-time systems in mind. They enable engineers to model real-time software and detect design errors through simulation of the software architecture.

As illustrated in Figure 7, CARDtools provides four levels of software support:

1. conventional CASE
2. real-time extensions
3. multitasking design with target knowledge
4. Ada/government

The primary real-time facility is the TaskTimer, which is implemented using multipath simulation techniques. It allows software developers to evaluate the performance of their software architecture early in the design phase, before generating code. Real-time developers can visualize how paths preempt one another and how concurrent tasks compete for resources. TaskTimer simulates the timing behavior and operating system overhead of multiple paths in the software design.

The software behavior is simulated according to the mechanisms provided by VRTX/ARTX real-time executives. VRTX is a development and runtime environment that makes full use of the Sun Unix network, and is integrated into Sun's extensible Window System. ARTX stands for Ada real-time executive. Issues concerning concurrency, task priorities, scheduling, synchronization, and communication are managed by TaskTimer. In addition, TaskTimer deals with both synchronous and asynchronous interrupts, which can be activated at appropriate times during the simulation. Since TaskTimer incorporates characteristics of the operating system and the target architecture, costly design and performance errors can be detected and eliminated during the development stages.

4.3 StP

StP is used for embedded and commercial systems development [37]. StP provides a collection of graphic editors that use a different structured design methodol-

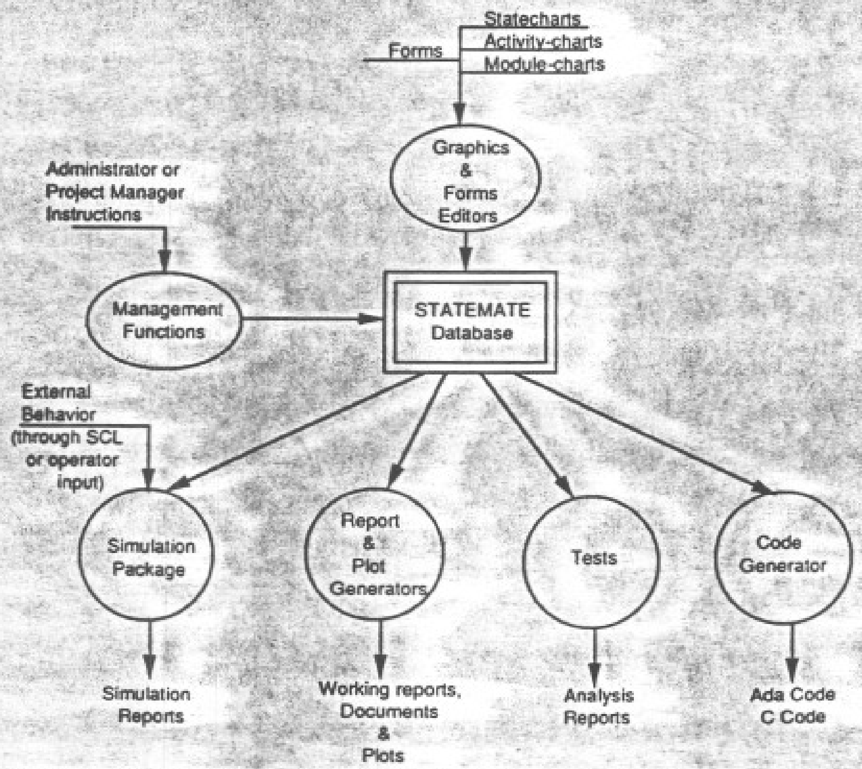

Figure 6. Overall structure of STATEMENT (adapted from [35]).

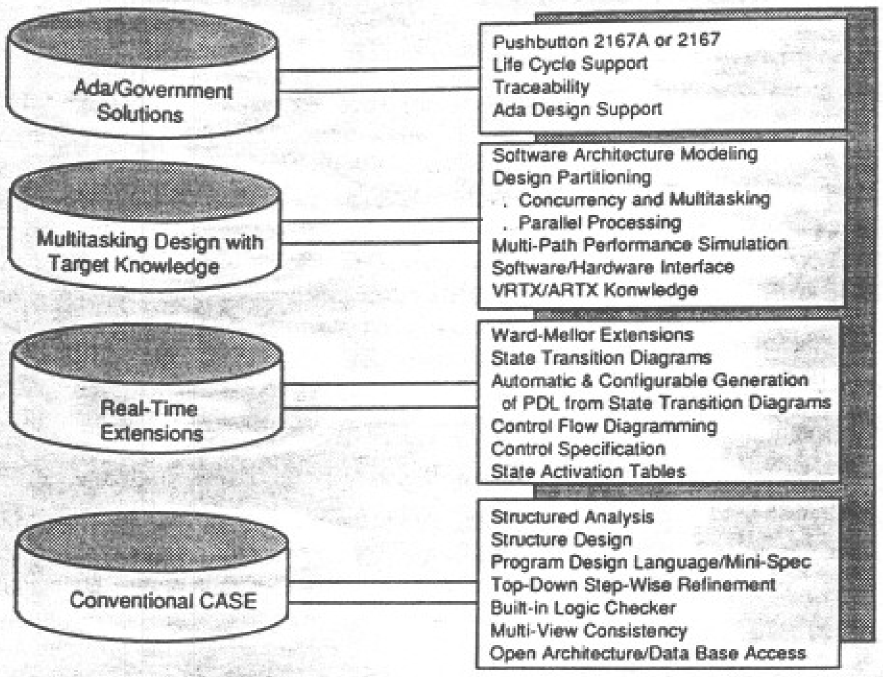

Figure 7. CARDtools offers four levels of software support (adapted from [36]).

ogy and are linked by a relational data base management system, as shown in Figure 8. Among others, StP supports the object-oriented structured design (OOSD) approach [38], not only through the drawing of the OOSD chart but also consistency checking, code generation, design reuse, and comprehensibility. A graphic editor for OOSD is configured for a particular programming language by associating a set of language-specific templates with the editor. In this way, the appropriate set of detailed design symbols can be included, and code can be produced for the given language. For example, a language-specific editor, such as the OOSD/Ada design editor, linked with a structured editor and an Ada programming environment (compiler, linker, debugger, etc.) produces an OOSD development environment for Ada.

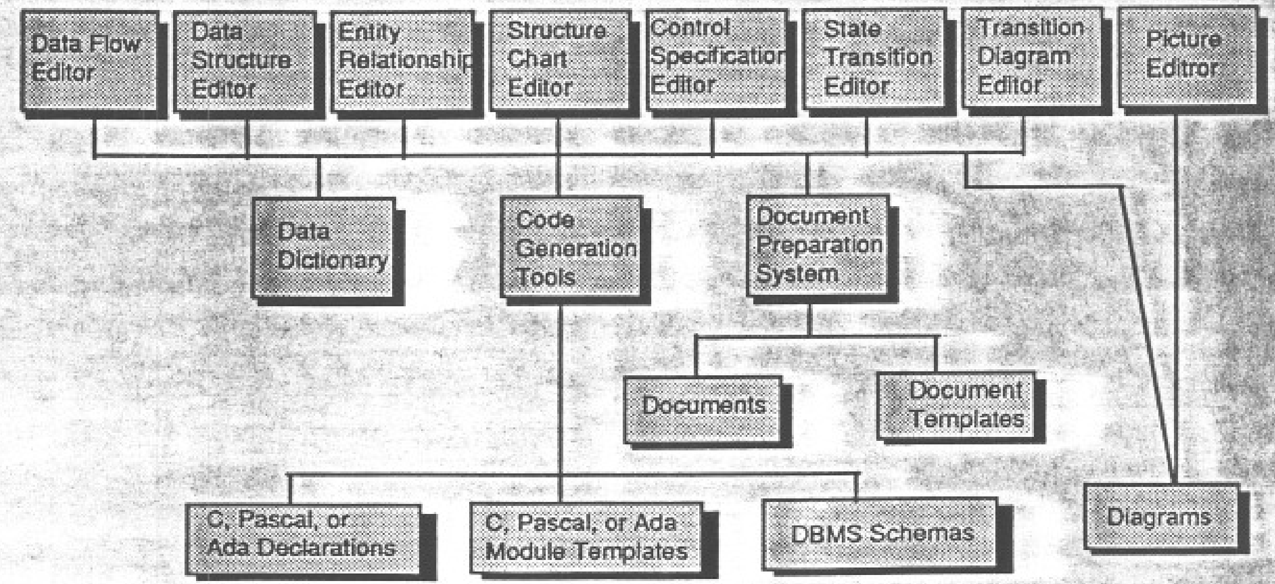

Figure 8. StP integrated environment (adapted from [39]).

5. SUMMARY AND RESEARCH ISSUES

In this survey we described examples of real-time systems, identified key issues in the design of such systems, examined design methodologies, and reviewed three commercial design tools. Real-time systems can be characterized by their real-time response and inherent concurrency. The critical nature of many real-time applications requires the systems to be ultrareliable. Future real-time systems will be more complex and require distributed implementations. Although several design methods and tools have been proposed, they fall short in meeting all the requirements. Based on our studies, we list the following as the major research issues that must be addressed.

Not all real-time systems are the same. Although all real-time systems share common aspects, they differ in their scale, criticality, complexity of communication requirements, and execution environments. Some are only a few thousand lines of code, while others are millions of lines of code. Some require formal verification of logic and timing correctness. Some applications rely on tightly coupled communication, while others operate in loosely coupled environments. Some systems must be ultrareliable. Thus, it is highly desirable to categorize real-time systems such that appropriate design methods, architectures, and fault-tolerance strategies can be identified.

Specification and analysis of time. The analysis of timing requirements of real-time system must be carried out at several phases of development, including specification, design, and implementation, for their completeness, consistency, and correctness. Existing analysis techniques are either too complex or too limited in their scope to be useful in all phases of development. Distributed or parallel computing environments in which timing behavior depends on many factors, including task allocation, scheduling, and communication, complicate the analysis of timing requirements. Thus, approaches to cope with the analysis of time during all phases of real-time system development are needed. The analysis of the time for error detection, recovery, and reconfiguration must be an integral part of the analysis.

Integration of fault-tolerance and reliability analyses. Fault tolerance is paramount in all life-critical systems. Although fault-tolerance techniques have been investigated thoroughly, problems in the practical application of these techniques remain. One of the more serious problems is the cost of designing and implementing multiple software versions. An efficient and systematic way of designing multiple versions must be investigated. Design of effective error-detection mechanisms, i.e., voting and acceptability check routines with multple software versions is another challenging area. Specification of timing requirements in light of fault tolerance needs investigation. Although reliability models have been proposed for hardware components and fault-tolerant software, models for the study of system-level reliability are needed. The future reliability model must include not only hardware and software components, but also communication subsystems.

Reuseable software components. It is generally believed that software reuse can reduce development cost while reducing the possibility of errors. However, the reuse technology is far from useful in complex systems. It is necessary to develop new methodologies for the specification, customization, cataloguing and retrieving of software components. It is also necessary for methodologies to optmize the software after integration of reusable components to meet timing and reliability requirements.

Automated implementation and CASE tools. Although many CASE tools are available, tools for complex distributed real-time systems design and implementation are scarce. We believe the next generation tools should strive:

1. to assist the system designers in identifying and defining the target applications. They should support systematic decomposition of the system and the definition of interfaces among the components;
2. to support principles of abstraction and reuse. They should support the automatic cataloguing, storage, retrieval, and update of components;
3. to support the analysis of the temporal behavior of systems for completeness, correctness, feasibility, and predictability. They should aid in the selection of proper timeout intervals and other synchronization points;
4. to assist in the incorporation of redundancy, error-detection, and recovery mechanisms. They should also facilitate the analysis of the system for reliability and performance.

Finally, to cope with future distributed real-time systems, the development environment and tools must permit the use of a variety of formalisms, design methodologies, languages, and target architectuers.

REFERENCES

1. A. Hall, Seven Myths of Formal Methods, *IEEE Software* 6:11–19 (1990).
2. H. Kopetz, W. Merker, and G. Pauthner, *The Architec-*

on User Operations, *IEEE Software* 4:39–50 (1988).

5. K. W. Nielsen and K. Shumate, Designing Large Real-Time Systems with ADA, *Commun. ACM* 30, 695–715 (1987).
6. B. Sanden, Entity Life Modeling and Structured Analysis in Real-Time Software Design—A Comparison, *Commun. ACM* 32:1458–1466 (1989).
7. G. Booch, Object-Oriented Development, *IEEE Trans. Software Eng*. 12:211–221 (1986).
8. D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1988.
9. S. M. Yang, K. M. Kavi, A. Agarwalla, M. Reddy, and S. Anam, SUVS: A Distributed Real-Time System Testbed for Fault-Tolerant Computing, *Proceedings of 1992 Symp. on Applied Computing*, pp. 782–791, ACM. Kansas City, 1992.
10. C. G. Davis, Ballistic Missile Defense: A Supercomputer Challenge, *IEEE Computer* 30, 37–46 (1980).
11. J. R. Cameron, An overview of JSD, *IEEE Trans. Software Eng*. 12:222–240 (1986).
12. M. A. Jackson. *System Development*, Prentice-Hall Englewood Cliffs, New Jersey, 1983.
13. H. Gomaa, Software Development of Real-Time-Systems, *Commun. ACM* 27:657–668 (1986).
14. H. Gomaa, Structuring Criteria for Real-Time System Design, *Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, Pennsylvania, May 15–18, ACM SIGSOFT* 23:290–301 (1989).
15. H. Gomaa, A Software Design Method for Distributed Real-Time Applications, *J. Systems Software* 9, 81–94 (1989).
16. B. Meyer, *Object-oriented Software Construction*, Prentice Hall International, Englewood Cliffs, NJ, 1988.
17. T. Bihari, P. Gopinath, and K. Schwan, Object-Oriented Design of Real-Time Software, *Proceedings of the Real-Time Systems Symposium*, 1989, pp. 194–201.
18. S. T. Allworth and R. N. Zobel, *Introduction to Real-Time Software Design*, 2nd. ed., Springer-Verlag, New York, 1987.
19. S. P. Hufnagel and J. C. Browne, Performance Properties of Vertically Partitioned Object-Oriented Systems, *IEEE Trans. Software Eng*. 15:935–946 (1989).
20. B. Sanden, An Entity-Life Modeling Approach to the

[illegible]

24. P. T. Ward, The Transformation Schema: An Extension to Data Flow Diagrams to Represent Control and Timing, *IEEE Trans. Software Eng*. 198–210 (1986).
25. A. K. Deshpande and K. M. Kavi, A Review of Specification and Verification Methods for Parallel Programs, Including the Dataflow Approach, *Proc. IEEE* 1816–1828 (1989).
26. A. K. Deshpande, A Framework for Network of Processes Using Dataflow Graph Model, Ph.D. Thesis, UT-Arlington, Arlington, Texas, 1990.
27. K. M. Kavi, B. P. Buckles, and U. N. Bhat, Isomorphisms between Petri Nets and Dataflow Graphs, *IEEE Trans. Software Eng*. 13:1127–1134 (1987).
28. K. M. Kavi and A. K. Deshpande, A Model and a Proof System for Parallel and Distributed Processes, in *Proceedings of the 23rd Hawaii International Conference on System Sciences* (*HICSS-23*), 1990, IEEE, Hawaii, pp. II.386–II.392.
29. W. B. Ackermann, Dataflow Languages *IEEE Computer* 11:15–25 (1982).
30. J. R. McGraw, SISAL: Streams and Iterations in a Single Assignment Language—Reference Manual, *Lawrence Livermore National Labs*, 1985.
31. J. B. Dennis, Dataflow Supercomputers, *IEEE Computer* 13:48–56 (1980).
32. V. K. Arvind, and K. Pingali, A Dataflow Architecture with Tagged Tokens, Technical Memo #174, LCS, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1980.
33. A. Garbrielian and M. K. Franklin, State-Based Specification of Complex Real-Time Systems, in *Proceedings of the 1988 Real-Time Systems Symposium*, 1988, IEEE, Huntsville, AL, pp. 2–11.
34. A. Topper, Evaluating CASE Tools, *Emb. Syst. Progr*. 4:55–77 (1991).
35. D. Harel, et al. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. Software Eng*. 403–414 (1990).
36. *CARDtools Product Brief*, Ready System, 1991.
37. *Software through Pictures: Products & Services Overview*, Interactive Development Environment, 1991.
38. A. I. Wasserman, P. A. Pircher, and R. J. Muller, The Object-Oriented Structured Design Notation for Software Design Presentation, *IEEE Comp*. 21:50–63 (1990).