Analysis and Applications of Receptive Safety Properties in Concurrent Systems[1]

Gilberto Matos

[1]This research is Supported by the *Office of Naval Research* under contract ONR N000149410320

ABSTRACT:

Formal verification for complex concurrent systems is a computationally intensive and, in some cases, intractable process. The complexity is an inherent part of the verification process due to the system complexity that is an exponential function of the sizes of its components. However, some properties can be enforced by automatically synchronizing the components, thus eliminating the need for verification. Moreover, the complexity of the analysis required to enforce the properties grows incrementally with addition of new components and properties that make the system complexity grow exponentially. The properties in question are the receptive safety properties, a subset of safety properties that can only be violated by component actions. The receptive safety properties represent the realizable subset of the general safety properties because a system that satisfies any non–receptive safety properties must satisfy related receptive safety properties. This implies that any system with realizable safety requirements can be described as a set of components and receptive safety properties that specify the component interaction that satisfies the requirements. We have developed a method that automatically synchronizes complex concurrent systems to enforce their receptive safety properties. Many non–safety and non–receptive properties can be represented using receptive safety properties, and automated synchronization can be used to enforce them.

Contents

Chapter 1

Introduction

Modern software systems must address diverse requirements including user profiles, heterogeneous hardware, distributed execution, reusability and maintainability, in addition to the classic set of requirements like correctness, performance, reliability, fault tolerance and security. The complexity of the combined requirements can be overwhelming and designers usually try to reduce it using decomposition. Concurrent systems include multiple autonomous processes that interact with each other and the environment. These processes are considered to be *components* of the system. This thesis explores how the interaction between components can be specified using *receptive safety properties*, a class of system properties that can be enforced automatically at the system level. We introduce a toolset, *GenEx*, that integrates the components with automatically generated synchronization mechanisms that enforce the specified receptive safety properties while preserving the component functionality.

1.0.1 Components, Interaction and Synchronization

Components in a concurrent system can interact in a variety of ways, from shared memory and message passing methods to explicit synchronization mechanisms like semaphores. In systems that interact with the environment, components may also interact indirectly through environment reactions. Component interaction determines how the system as a whole satisfies its requirements. Consequences of incorrect interactions range from loss of performance to total and catastrophic system failures, while correct interaction is required in reliable and high performance systems. We can specify the desired interaction by a set of system safety and liveness properties, and designs that satisfy those properties are considered correct. These interaction properties are usually both conceptually and computationally simple, given that the components encapsulate the complex functionality of the system.

Despite the simplicity of individual interaction properties, the aggregate complexity of the system behavior tends to be an exponential function of the number of components. The combinatorial complexity of concurrent system behavior makes them hard to analyze and verify using formal methods, and increases the need for reliable design methods. Many interaction properties are, or can be represented by receptive safety properties. These properties specify the interactions that can guarantee the safety and reliability, or improve the performance, fault tolerance, and real–time behavior of the system. Synchronization in a concurrent system has the role of adjusting the behavior of individual components to the behavior of the system, to keep the system in a consistent and safe global state. Our research shows that receptive safety properties can be enforced by automatically synchronizing the components.

The specification of each component determines *what* the its next action is in any system state, and we assume that the resulting behavior is correct with respect to the components' functional requirements. Information about other components in the system is necessary to determine *when* that action is acceptable. System synchronization modifies the *when* decisions made by the individual components, and superimposes additional delays on some of those actions as required by the synchronization mechanism. When the *what* and *when* aspects of the system's behavior can be separated, i.e. when the functional and interaction requirements can be expressed independently, then receptive safety rules can be enforced by *implicit synchronization*, using automatically generated code stubs that modify the component behavior to satisfy the interaction properties. When the interaction between components is specified and enforced at the system level, the components' design is simplified because they must satisfy only their functional requirements. The use of simpler components and automated synchronization simplifies the software integration process.

1.0.2 Enforcement vs. Verification

System correctness is defined as consistency with a set of system requirements or specifications. To declare a system correct, all of its behaviors must satisfy the relevant system requirements. Consistency of a system with its requirements can be verified manually or by using automated model checking or proof techniques. Model checking tools [McM93] can verify that a system implementation satisfies all defined requirements by analyzing a symbolic representation of its state space. Proof techniques generally support the user in the construction of proofs that the system implementation implies that its requirements are satisfied. These formal verification methods are widespread in the research environments, but their acceptance in the industrial and development organizations is lagging.

One of the biggest problems with the formal verification approach is their complexity and the stringent training requirements for their users. The most accessible and accepted formal methods are based on the finite state machine formalism, being the best understood and most often used in programming. Even when componentsare finite state machines, the complexity of the system behavior is an exponential function of the number of components. Many verification tools attempt to reduce the complexity by various methods, but the verification in the general case requires a level of complexity proportional to the complexity of the system behavior.

Formal verification is gaining acceptance in practical design environments wherever the system reliability is paramount. Some factors contributing to the wider use of formal verification are increased complexity and criticality of software systems, regulatory requirements, and improvements in the computer performance that makes more complex systems verifiable in acceptable time.

In some systems, it is easier to modify the components to enforce some system property than it is to verify whether the system satisfies the properties. The problem with formal verification, even for finite state based systems is that the system behavior is related to the data processing in the components. Verification must take into account the influence of the computed data values on component behavior, and the possibly infinite or erroneous computations that may interfere with the system.

Receptive safety properties define the correct and incorrect sequences of system actions in a time–independent form. Their time–independent nature means that every incorrect execution trace contains a specific action that causes a property violation. Preconditions of the property violations can be determined from the property specifications, as can the violating actions and the components that cause them. Enforcement of the receptive safety properties is both conceptually and technically simple, requiring a delay of the violating components as long as the violation preconditions are satisfied by the system state. Only the actions that cause the safety violations need to be delayed, while all other actions can remain unchanged and execute when enabled without delay. The complexity of the analysis needed for safety enforcement is therefore limited to the number of possible safety violations, and is not necessarily proportional to the state space of the system or its parts.

1.0.3 Scope of the Thesis

This thesis develops and implements a theory for automated software synchronization, driven by receptive safety rules. This requires developing a formal model of component synchronization, relating this model to the receptive safety rules, and constructing a program that implements the synchronization based on an analysis of component interaction with respect to system rules. Boundaries of this investigation have been chosen to concentrate it on practical issues and to limit the scope of the project.

First, only receptive safety properties are used to define the desired system behavior, and synchronize the components to enforce it. Many safety properties used in practice are non–receptive, and they can not be automatically enforced by our method. However the non–receptive properties can not be satisfied completely even by manual system design; a system that satisfies a non–receptive safety property, also satisfies a more restricted receptive safety property. Receptiveness of safety properties is an important issue in the practical implementation of reliable software.

Another restriction is that the thesis is primarily concerned with the control aspect of component and system behavior. Methods for design and analysis of control and data processing systems exhibit important differences, due to the different nature of the two aspects of computation. Data processing is primarily concerned with the relationship between complex sets of values, possibly non–discrete ones. Conversely, the control aspect of computing has to do with the sequencing of system events, a definitely discrete domain.

Our synchronization method uses finite state machines to specify both components and sequencing rules. Restriction to finite state machines may seem to be very limiting, but in practice it is rare to find safety rules more complex than regular languages. Components defined using most modern programming languages exhibit a finite state behavior as modeled by their control graph. This restriction is not of primary importance for automated synchronization, because the analysis and enforcement mechanism could handle the use of pushdown automata corresponding to components and safety rules in the context free domain, or state–based representations of even more complex languages.

Finally, the implementation of this method works with a specific FSM notation geared toward the specification of system properties. This notation is theoretically equivalent to the FSM notations used in other formal design methods, but in practical terms it makes a clear distinction between the component states and signal values, important from the standpoint of their effect on the system behavior.

1.0.4 Overview of the Thesis

The organization of this thesis reflects its orientation toward practical applications. Besides presenting the formal foundations of the method, we make heavy use of examples to show how our method and the supporting GenEx toolset can be applied in complex system development. The remainder of chapter 1 gives an overview of related work in this area.

Chapter 2 gives an extended introduction of the terms and notations used in this document, as well as the classification of system properties and their relationship to the realizability of the systems. The most important notion in this thesis is that of a receptive safety property that describes a set of system properties whose realizability is not affected by the environmental events. We also introduce the finite state notations accepted by our system and the basics of the delayed transition mechanism we use to synchronize components.

Chapter 3 describes the GenEx toolset and its main functions. We describe the details of the input notations for GenEx and how the components and safety rules are related to each other. We also describe some of the problems that may occur in designing systems using GenEx, and the ways to detect and solve them.

Chapter 4 describes the design of a reliable and starvation free dining philosophers system using the automated synchronization to produce the integrated system. This example shows how GenEx supports the use of simpler components by eliminating the need for explicit use of synchronization mechanisms. Our synchronization mechanism satisfies the starvation freedom for the philosophers, and we also show how the starvation freedom can be represented by an explicit receptive safety property and enforced.

Chapter 5 introduces another example of automated synchronization used in a complex concurrent system. The production cell controller was used as the basis for a case study of concurrent system design and verification methods [LL95]. We show how a controller for this system can be designed and automatically synchronized using GenEx, and how easy it is to modify that controller to implement a more complex fault tolerant version.

Chapter 6 includes the formal consideration of the compositional nature of finite state implementations of components and receptive safety properties. It also contains a description of several patterns for system design that guarantee the deadlock freedom of the automatically synchronized applications. Finally, this chapter also includes some examples showing how receptive safety properties can be used to enforce nonreceptive properties like real–time execution.

Chapter 7 describes the use of GenEx in the reengineering of an existing system, where the component code can be reused and automatically synchronized to satisfy the original requirements and facilitate further modifications.

1.1 Related Work

This section shows some related approaches in the design and verification of concurrent systems. Theoretical work by Abadi and Lamport [AL93] established a framework for the proof of correctness and safety of systems synchronized automatically using GenEx. They also identified the receptive safety properties and defined the relationship between receptiveness and realizability. Other approaches are practical design methods dealing in different ways with the individual and aggregate behavior of components in a system. Some of these methods attempt to automate the verification of user–designed synchronized systems. Our approach in GenEx is to automate the design of one aspect of the system components, that related to their control behavior. We modify the components to make the system enforce a given set of receptive safety properties.

1.1.1 System Control, Synchronization and Receptiveness

Clarke [EC82, CE82] has shown that properties of the control aspect for many systems are simpler than the properties of their data processing, and that the implementation of these two aspects can be separated. He discussed the generation of synchronization skeletons from temporal descriptions of system behavior. This approach was mostly theoretical because the discussed complexity of analysis exceeded exponential growth. While the method was not practically applicable to even medium size systems, it introduced the notion of enforcement for aggregate system properties. The intractability of analysis in this method stems from the unrestricted nature of temporal properties that are enforced; our method selects a restricted set of properties whose enforcement requires dramatically lower complexity of analysis.

The concept of property receptiveness, introduced by Dill [Dil88], gives an intuitive classification of properties for open systems. Open systems consist of a controller and its environment and the interaction between them determines the system behavior. The environment in open systems can not be controlled, and no restrictions can be imposed on its behavior. Dill distinguishes properties that require restrictions on the environment behavior from those that can be satisfied by a system regardless of the environment actions. Receptive properties are those whose valid behaviors are closed under concatenation of environment events.

Receptiveness was further developed by Abadi and Lamport [AL93], through the relationship between a property and a controlled agent set that can enforce it. Moreover, they isolated safety properties as a specific domain of receptive properties that can be preserved under composition. Receptiveness is also strongly related to the concepts of realizable and unrealizable properties [AW89]. Intuitively, a rule is receptive if a system can enforce that rule regardless of environment actions. In general, any rule that can be violated by a sequence of environment actions is nonreceptive because no strategy of the system can prevent the violation from happening. A safety rule that restricts only controlled components is receptive because no environment action can force the rule violation as long as the system uses a safe synchronization strategy for the components.

1.1.2 Composability of Safety Properties

The work on specification composition [AL93] proved that a composed system preserves some types of properties enforced by its components. These results make a clear distinction between the composability of safety properties, and that of more complex properties that include liveness, and fair behaviors. When a subsystem satisfies a given safety property, provided a set of assumptions in the form of safety rules is valid, the composed system satisfies the safety property unless some part of the system violates its assumptions. Composability of safety rule enforcement is rooted in the restrictive nature of their enforcement. Safety properties can be enforced by restricting component execution using synchronization mechanisms, and composition of restrictions is also a restriction.[1]

Reachability, liveness and real–time properties may require the completion of certain transitions as a way to enforce them. A composition of two such properties may require the simultaneous completion of incompatible transitions, thus making the system that satisfies the composed specifications unrealizable. This shows that composition of non–safety properties is a harder problem than that of safety, and that those two classes of properties should not be treated as equals. The specification composition method can be generalized to these types of properties if their preconditions can be specified in the form of safety properties. These safety properties can be enforced by automated synchronization, and provide a tool for enforcing the nonreceptive properties.

1.1.3 Verification of Safety Properties

Correctness verification is an essential part of the development of complex concurrent and distributed applications. Testing can provide an estimate of the system reliability and correctness, but it covers only a subset of all executions, so errors can remain undetected. Formal checking efforts in the area of concurrent systems have been concentrated in two major areas: proving temporal properties of finite system abstractions, and trying to prove that implementations satisfy the specifications. Proving correctness of abstract descriptions is of limited use because of the possible discrepancies between the implementation and the description. In general the scalability of this approach is limited by the complexity of the system. When complexity is kept low, *mcb* [Bro86, CES86] can successfully and efficiently check formulas in first order temporal logic CTL.

Proofs on real code are rarely used because their complexity is generally unacceptably high, and they are often undecidable. Some systems try to extract abstract information from the

[1] To illustrate this, consider an example: given any set of time–independent safety rules with consistent initial states, a system that stays in an acceptable initial state trivially satisfies the safety rules. This example shows the conceptual simplicity of composing systems for safety as well as a potential pitfall in automated synchronization.

source and do partial analysis. STeP [ZM+94] tries to prove the given assertions automatically and when that fails it lets the designer guide the proof by choosing the assertions that are to be proved. Analyzer [CG94, Che96] requires additional information related to the abstract component description to be inserted in the source code, and combines it with the program reachability graph to check the consistency of the program and SCR [Hen80] style specifications. Due to the undecidability of the program behavior, this analysis is either optimistic or pessimistic, and exact analysis is impossible. However, he requirement to annotate the code for analysis has a positive side–effect, it forces the designer to understand and document the relationship and the mapping between the specifications and the code. These two systems both support the idea that automatic checking is unable to deal with the data processing aspect of computation, and human involvement is required in system validation and verification. GenEx is defined in the domain of system interaction, where automated verification and synchronization is possible because it is isolated from the data processing aspect, and its complexity is inherently limited to the finite–state domain.

Compositional and symbolic model checking are two approaches that try to reduce the complexity of the state space representations. Compositional model checking [CLM89], [FG94] tries to limit the complexity by constructing abstractions that can represent system components in further analysis of the given properties. By eliminating states that are irrelevant to the property, it can achieve significant reduction in the complexity of the analysis. This approach is orthogonal to automatic synchronization, and the same abstraction and removal of irrelevant states can be used in GenEx to reduce the complexity of the reachability analysis. Symbolic model checking [BCM+90] relies on the symbolic representation of the state space, where regularities in the state space are exploited to minimize the complexity of the representation. These techniques are very powerful analysis tools, but they require the designer to correct all inconsistencies. Also the correctness of the abstraction in no way guarantees the correctness of the implementation done by hand, a fact that reduces the practical applicability of those systems.

1.2 Design Notations for Reliable Concurrent Systems

Formal verification suffers from the system complexity and state space explosion problems, as well as high requirements for user training. Many programming languages and specification notations have been developed to help produce reliable concurrent systems, where some properties or behaviors are guaranteed by design, and do not require any verification. One of the first broadly used formal notations for concurrent systems was StateCharts [Har87], a graphical notation using states and transitions to represent complex systems. They use hierarchical clustering and refinement to specify complex components from sets of simpler ones. Simple components may be combined sequentially or in parallel within a larger subsystem, and transitions can be combined to represent different forms of component interaction. The explicit graphical nature of this model makes the system structure behavior and structure very intuitive, making it easier for the users to understand the implications of their design decisions. The computational complexity of the system used for formal verification remains proportional to a product of the parallel subsystems, possibly exponential for larger systems.

Several programming languages for reliable concurrent systems are based on the tabular approach, where every component is defined by a table specifying its reactions to specific system states. Two examples of this approach are SCR [Hen80, AFB+88] and RSML [LHHR94]. The tabular approach has the advantage of being simple for the end users to understand and comment on from the domain knowledge point of view. The tabular notation makes these languages simple to automatically analyze for completeness and consistency [HL96], and refine their behavior by modifying the specification table [AG93]. We use this basic notation for the component specifications in GenEx, because of the simplicity of component behavior modifications.

Several systems have used the code generation to implement synchronized concurrent systems, LUSTRE [CRR91] and Esterel[BG92] being based on a similar model of computation as GenEx. These systems use a synchronous transitions model of computation, where every component executes one action in every system cycle. The synchronous transitions model makes them simple to analyze and generate code for. Both LUSTRE and Esterel support the verification of given system properties versus the system behavior. GenEx differs from these languages because instead of verifying that the specified system satisfies the given properties, it actually computes the necessary synchronization of the components that makes the system enforce receptive safety properties. This difference is fundamental because GenEx allows the programmer to give a partial system description, and have it automatically refined to satisfy the given set of rules; the other systems would notify the programmer if the description satisfies the rules and if not, the design would require some changes by the programmer. Apart from requiring high skill, the manual refinement might also involve sizeable effort because the physical size of the description might have to increase.

Labeled transition system(LTS) [CK95, CK96, CGK97] is another finite state notation for component specifications that uses a synchronous transition model. The assumptions in this model are even more restrictive than the other synchronous transition models because the set of components that will execute a transition in any given cycle is determined based on the system state after the previous transition is completed. A transition with the label l can be executed only if all components that use that label for any transition have a transition with that label enabled. This formalism makes it trivial to enforce certain types of safety properties, by adding a component that uses a labeled transition when it is safe, and disables it when it may cause a safety violation. The problem with the LTS approach is that the constraint is very restrictive and results in very easy occurrence of deadlock states when all labels are disabled.

1.3 Requirement Decomposition and Enforcement

The concept of product state machines, as described in [Lim93, Lim96], is conceptually very similar to GenEx. The main difference between them is the scalability. In this system, the global reachability graph is constructed and then restricted to eliminate violation states. The restricted graph is then used in the execution. The reachability graph may be too complex to be useful in practice, even using their abstraction and composition trechniques to reduce the state space. Other similar approaches exist in the hardware design area where the behavior of circuits can be completely modeled and the sequential circuit is generated as an instance of the verified model. Conceptually, GenEx does the same thing, but the emphasis is on the local analysis and synchronization of components, and the complexity is kept low because the synchronization mechanism for every safety rule is independent.

Another related concept is that of Safety Kernel [WK95] that is less formal, but involves the code generation capability and automatic safety implementation. This centralized, and more importantly sequential, paradigm makes the code generation trivial by reducing it to a simple runtime check of the desired property. The main shortcoming of this system is its centralized safety kernel, making it useful in its domain of physical safety enforcement, but not really applicable to the concurrent and potentially distributed systems. Despite this shortcoming, the system is an example of how simple methods can solve complex problems, given the right domain.

Brewer and Kuszmaul [BK93] investigated the impact of synchronizationon system performance, and found that synchronization in some cases can contribute to improving the performance of a system beyond what could be achieved by asynchronous execution. This shows that performance requirements can sometimes be reduced to safety properties which can be enforced using synchronization. We will show later how other non–safety properties can often be reduced to receptive safety properties and enforced by automated synchronization.

Aspect–oriented computing [KIL+97] is closely related to our work by its emphasis on separating different aspects of system behavior and the use of automated integration to generate system implementations. This field is very broad, and involves many types of system properties, and different types of systems and objects. The idea in aspect based computing is that the user supplements the component sources with descriptions of some global aspects of component interaction. Aspects guide the integration of the procedural components, according to the aspect "weaving" rules given with the aspect descriptions. This makes components reusable across many applications regardless of their global structure and requirements, and makes the aspects reusable in many applications with similar global properties regardless of the system components.

GenEx defined systems have a similar structure to aspect based computing, where components are the main building blocks, safety rules are the aspects the system should satisfy, and the computation and decomposition of synchronization conditions corresponds to the aspect "weaving" method. The GenEx approach is focused on one class of aspects and comes with a predefined integration method for enforcing them. By limiting the approach to a certain class of systems, we can make better tools for integration and system verification, allowing us to automatically generate correct implementations.

1.3.1 Interface Specifications, Verification and Synthesis

Garlan and others [GS93, AG94] investigated the interface specifications from the standpoint of external control, and introduced the concept of glue protocols. Behavior of individual component interfaces is a set of possible execution traces, but some traces incompatible with the glue are made unreachable to make the interface consistent with the glue. The interface and glue specifications are manually designed, and may be verified for consistency and compatibility. Another similar approach was developed by Katz [Kat93], where control constructs are superimposed on the behavior of the components to achieve specific system behaviors.

Yellin and Strom [YS97] discuss the automated synthesis of interface adaptors according to finite state protocol descriptions for the connectors being interfaced. This work is conceptually similar to the GenEx synchronization, but lacks the modular nature in the adaptor synthesis, and is limited to the message passing constraints, not the component behavior itself. Modularity of the synchronization process makes GenEx applicable for systems with complex interactions between multiple components. The adaptor synthesis approach is defined for interfaces between two components, and all constraints for a given interface are handled together, potentially leading to exponential complexity of the adaptor. While they work with interfaces and parameters of non–finite state nature, they extract a finite state specification using a dependence mapping of parameters. The mapping specifies a partial order for the messages according to the dependence between their parameters. This partial order makes a finite state structure where the synthesis by state enumeration is possible.

Park and Miller [PM97] worked on automatic synthesis of interfaces to implement a service specification involving a number of components with real–time characteristics. Their work is based on timed finite state machines, and the individual protocol synthesis concentrates on interval planning, and introduction of synchronizing actions in the machines. Since the system handles real–time requirements, it has no guarantee of success, but does guarantee to satisfy a maximal set of requirements. As in the case of Yellin and Strom, the complexity of this method is proportional to the size of the service specification, reducing its applicability for complex systems and interactions when the full specification is a composition of numerous components.

Chapter 2

Systems, Components and Interaction Properties

Concurrent systems are a very active research area, and numerous notations and supporting semantic domains have been developed. These systems are based on various divergent and even conflicting assumptions. These assumptions deal with the semantics of component execution, whether they execute asynchronously [Hen80], synchronously [HLR92, BG92], or with runtime determination of synchronization requirements [CK95]. Other important differences between concurrent system specification methods arise in the area of time, where some systems assume a continuous timeline model [GMM90], some operate with finite intervals and a timeline made of discrete events [BG92], and others work with branching time model [McM93, CLM89].

In this chapter we will define the type of concurrent systems that our system handles, as well as the semantics of their behavior and properties. We will introduce the assumptions of our semantic model and show that it is a subset of a more general semantic model used to define the relationship between system properties, components and realizability. We will also give an overview of the theory of system realizability, and a classification of system properties that identifies the properties that can be enforced automatically. We use the delayed transition mechanism, introduced in Section 2.7, to synchronize the components and enforce the properties. Section 2.8 describes some types of enforceable properties and shows the synchronization mechanisms they require.

2.1 Open Control Systems

Our synchronization method and the associated analysis tools are specifically designed to process open control systems. Open control systems are interactive systems where some events may be unpredictable, possibly controlled by a malicious environment entity. Our goal is to design a controller that will enforce the required system behaviors, assuming the environment behavior satisfies a set of assumptions. The environment comprises all parts of the system that are not

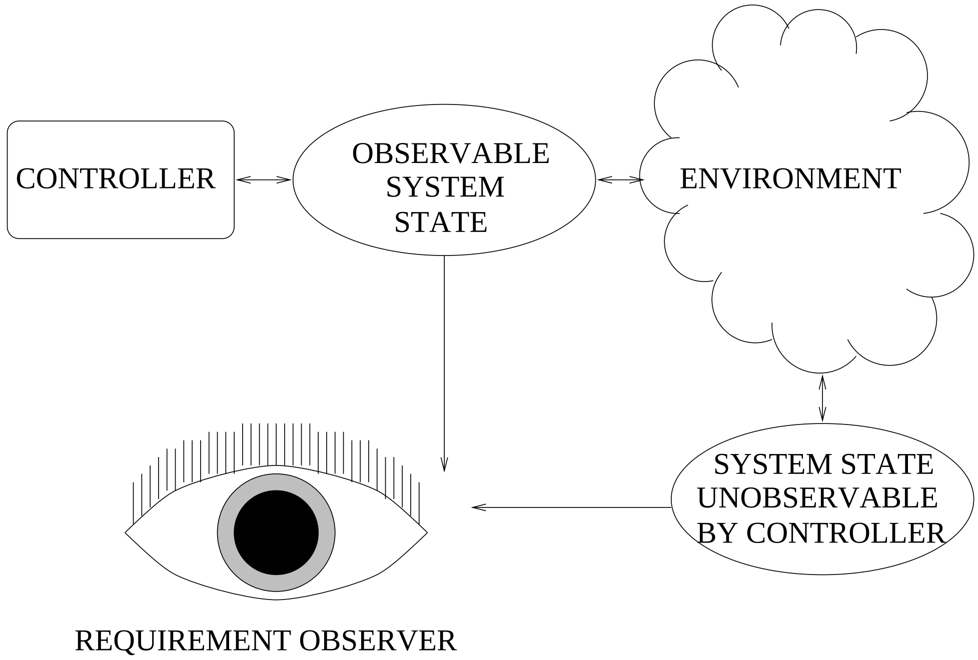


Figure 2.1: The structure of an open control system

directly controlled by the controller Some events may be completely or partially controlled by the environment, as shown in Figure 2.1. The controller monitors the observable state of the environment and uses a predefined and limited range of actions to influence the environment behavior. The environment reacts to controller actions, and the interaction between the controller and the environment produces the system behavior. User specifies the system requirements to accept or reject individual system behaviors based on the occurrence of specific events or lack thereof. Some aspects of environment behavior may be unobservable by the controller, but still contribute to the system state, and may influence the satisfaction of system requirements.

The controller essentially plays a game with the environment, where the controller wins if the system requirements are satisfied and the environment wins if some requirement can be violated. Controller tries to prevent the system behavior from violating any requirements, always assuming that the environment has the opposite goal. In this game oriented abstraction, the controller is said to implement a *strategy*, and a winning strategy is one where the environment does not have any possible action that may violate the system requirements. In practice, a program (a set of system specifications) represents a winning strategy iff it can be proved to satisfy the requirements regardless of the given inputs or the timing of the environmental events.

2.1.1 Dining Philosophers

We will illustrate these concepts using the dining philosophers example. The system consists of a set of processes (philosophers) which perform two functions, thinking and eating. The philosophers' behavior is simple, they think until they get hungry, and eat until they stop being hungry. Every philosopher can think regardless of the state of the other philosophers, but due to the lack of forks on the table there are constraints on when they can eat. There is exactly one fork between each two chairs, and every philosopher needs both adjacent forks for eating. This means that when one philosopher is using the two adjacent forks to eat, the two neighboring philosophers cannot eat.

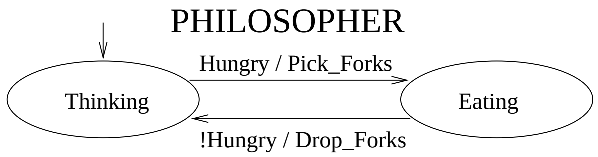
The dining philosophers can be viewed as an open control system, where the environment controls the hunger of the philosophers, while their decisions to start and stop eating are considered to be controller actions. The assumption about the environment behavior is that a philosopher that eats eventually stops being hungry. The requirements for the system behavior define the interaction between the philosophers, in this case the constraints on when they are allowed to eat. Another system requirement is that any hungry philosopher eventually gets to start eating. The game theory aspect of system design is simple in this case because the controller has total visibility of the system state, and its actions can be determined based on the current situation. This system has been extensively studied in literature, both for synchronization design [Dij, Hoa78] and for system verification [YY91]. We will use this system to illustrate different aspects of automated synchronization.

2.2 Systems as Sets of Components

The modular architecture of software systems is based on active concurrent components that communicate through passive interfaces. A component can be described by its control behavior and its data processing aspect. The control behavior of a component determines the sequence of its actions, while the data processing aspect specifies the data that is used and produced by the component during specific control actions. The data processing aspect includes potentially undecidable problems, and is hard to analyze and verify for correctness. The control behavior is generally simpler than the data processing, and often can be represented by regular languages or finite state machines (FSM in the remainder of this document). The regular language domain of control behaviors makes it possible to effectively analyze and verify their correctness using formal methodology.

Many formal methods for concurrent software design use FSMs to model the components. FSMs roughly represent the control graph of component implementation. The data processing aspect is implemented separately, but linkable to the final implementation. Systems are composed by executing the components concurrently, in some cases in lockstep with each component executing one transition in parallel, and this model is known as the synchronous transition model.

In our system we use components defined in the form of Mealy finite state machines [JEH79], whose output depends on the executed transition. Every component is defined as a tuple (S, A, I, τ) where S is a set of states, A is a set of boolean system variables the component can read or write, I is the initial state, and $\tau : (S \times 2^A) \rightarrow (S \times 2^A)$ is the transition relation mapping component state and inputs to the next state and the component outputs. The component can read and write the same set of variables and can use any combination of variables to determine the enabled transitions. This makes its alphabet the set of all possible combinations of boolean



a)

```
mode([philo],[[hungry]]).
init([philo],[thinking],[[]]).
trans([philo],[thinking],[eating],[[t]]).
trans([philo],[thinking],[thinking],[[f]]).
trans([philo],[eating],[thinking],[[f]]).
trans([philo],[eating],[eating],[[t]]).
```

b)

Figure 2.2: The control structure of a philosopher

values for the variables in A, represented above by the powerset 2^A. This definition of component behavior is similar to the tabular approaches in SCR [Hen80, AFB+88] and RSML [LHHR94].

Variables used to determine the enabled transitions for a given component are considered to be monitored, while the variables altered by its effects are controlled as defined in the SCR notation [Hen80]. Variables may be used for communication between components when they would be controlled by some components and monitored by others. From the standpoint of a controller system, variables are controlled if they are controlled by any of its components, and monitored if they are controlled by its environment. The main distinction between monitored and controlled variables is the independence of the monitored variables from system state and component transitions. The monitored variables can have arbitrary values at any point in time, while the controlled variable values are a function of the controller behavior.

The dining philosophers system consists of a set of components (philosophers) with identical behavior. The behavior of a philosopher is given in Figure 2.2a), using the finite state machine notation. The FSM notation is elementary with each transition labeled by an ordered pair **condition/effect** where the **condition** represents the enabling condition for the transition and the **effect** represents the change in system state that results from the completion of the transition. Each philosopher component monitors the respective boolean variable *hungry* and determines the enabled transitions based on its values. Figure 2.2b) shows the same specification in the form of Prolog predicates, used by our automated synchronization tools.

Predicate *mode* declares an ordered list of system variables referenced by the component. This list specifies the positions where the variable values will appear in the specifications of the component behavior. Predicate *init* specifies the initial state of the component and the initial values of its controlled variables, value in this list determines the variable whose value it sets. Every variable can be referenced in 3 possible ways: **[t]** for *true*, **[f]** for *false* or [] for undefined. Philosophers only use monitored variables that specify whether they are hungry or not, so initial values in this case are left blank. Predicate *trans* specifies a transition by giving its source and destination states, its enabling condition and the effect on the controlled variables. The enabling conditions for the transitions depend on the monitored variables, so they specify the required values. Since philosophers have no controlled variables, the transition effect list includes only bank elements.

2.3 Semantic Models of System Execution

The system behavior can be described as a game between the controller and the environment. The basic rules of this game are described by the semantic execution model that specifies when the players can make a move and what are the legal moves at any instant. The interaction between control systems and environments was studied by Abadi and Lamport [AL93], and we will use their classification of properties and their relationship to controller strategies to clarify the need for identifying receptive safety properties. This classification and theorem proofs are based on the agent set semantic model. This semantic model represents the controller actions as atomic and as members of a predefined set, and provides an elegant notation for defining system properties and reasoning about their relationship.

Our research concentrates on the composability of systems from independent components, and we selected a different but related semantic execution model. The synchronous transition model [McM93, BG92] defines controller actions as a combination of parallel component actions, and specifies strict interleaving between the controller and environment actions. This semantic model is based on the states of the individual components and the system variables, and is more oriented toward practical system design where individual components are the primary building block. The synchronous transitions semantic model represents a special case of the agent set model, and the theory of receptive safety properties developed for the agent set model also applies to the synchronous component execution model.

2.3.1 Agent Set Semantic Model

The agent set semantic model describes system behaviors using states and agents. We will give a short overview of their definitions, and more details can be found in [AL93].

Definition 2.1 *A* **state** *is an element of a nonempty set* $\mathbf{S}$ *of states. Every element of* $\mathbf{S}$ *represents the state, at some instant, of the system universe. System state represents a combination of the current controller state and environment state.*

Definition 2.2 *An* **agent** *is an element of a nonempty set* $\mathbf{A}$*. A set of agents* μ *is an* **agent set** *if it is a nonempty proper subset of* $\mathbf{A}$*.*

Agents in the agent set semantic model represent the entities that change the system state. The set of agents $\mathbf{A}$ is divided into two disjoint nonempty subsets μ and $\neg\mu = \mathbf{A} - \mu$, that represent the agents controlled by the controller and the environment respectively. The disjointness of the controller and environment agent sets implies that they always execute in some interleaving pattern and never in parallel. Both agent sets can be reduced to a single agent each, representing all disjoint controller and environment actions [AW89].

The following definitions introduce the notion of system behavior in this semantic model.

Definition 2.3 *A* **behavior prefix** *is a sequence*

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$$

where each s_i is a state and each a_i is an agent that modifies the system state s_{i-1} into s_i. A behavior prefix is either infinite or ends in a state s_m for some $m \geq 0$.

Definition 2.4 *A* **behavior** *is an infinite behavior prefix.*

2.3.2 Synchronous Transitions Semantic Model

We use synchronous transitions as the semantic model of controller behavior. This model assumes that the controller is composed from a set of components, executing in parallel. Every component selects one enabled transition in every system cycle and they all execute the transitions synchronously. The enabling conditions of component transitions are defined as a function of the system state, including both controlled and monitored system variables. Every component must select some transition in every cycle, and for every state there is an implicit default self–loop transition that is enabled in the absence of other enabled transitions. The effect of the completed transitions results in the change of component states and controlled variable values. To simplify the composition of transitions, every controlled variable should be controlled by at most one component; more complex variables can be represented as boolean functions of simple variables. Parallel component transitions represent a controller action, and changes in monitored variables represent the effects of environment actions. The controller and environment actions are strictly interleaved, so that an environment action is allowed after every controller action.

The system can be represented by an equivalent finite state machine that represents the combined behavior of the components. This equivalent FSM has a state for every combination of component states reachable in the system execution. The input alphabet of the equivalent FSM contains symbols that represent all combinations of monitored system variables. The output alphabet is defined similarly, as a set of symbols representing the combinations of controlled variable values. Given two components $C_1 = (S_1, A_1, I_1, \tau_1)$ and $C_2 = (S_2, A_2, I_2, \tau_2)$, the equivalent combined FSM is defined as $C_{eq} = (S_{eq}, A_{eq}, I_{eq}, \tau_{eq})$ where $S_{eq} = S_1 \times S_2, A_{eq} = A_1 \cup A_2, I_{eq} = I_1 \times I_2$ and the transition set is defined as $\tau_{eq} : (S_{eq} \times 2^{A_{eq}}) \rightarrow (S_{eq} \times 2^{A_{eq}})$. If component transitions include $\tau_1(s1s, a1s) = (s1d, a1d)$ and $\tau_2(s2s, a2s) = (s2d, a2d)$, and the enabling conditions of those transitions are consistent, $a1s \cap a2s \neq \emptyset$, then the combined transition set includes $\tau_{eq}((s1s, s2s), (a1s \cap a2s)) = ((s1d, s2d), (a1d \cup a2d))$. This definition of combined transitions implies that two component transitions can be combined when their enabling conditions are not contradictory. The controlled variables are controlled by at most one component, so the effects of transitions by distinct components are always consistent and can be combined. We say that a combined transition *includes* a specific individual component transition, if the combined transition represents the parallel execution of that component transition with arbitrary transitions of other components.

For any system defined using the synchronous transition model, we can construct its representation in the agent set semantic model, by representing the controller actions as agents and enforcing strict interleaving of controller and environment agents. For the system $C_{eq} = (S_{eq}, A_{eq}, I_{eq}, \tau_{eq})$ in the synchronous transition semantic model, we can construct a system $SYS_{as} = (S, \mu, \neg\mu)$

using the agent set semantic model with the identical set of possible behaviors. The set of states $S = S_{eq} \times 2^{A_{eq}} \times \{1, 2\}$ corresponds to all combinations of system component states and all values of system variables. The last part of the Cartesian product is a *phase selector* that specifies the interleaving between the controller and the environment. The environment agent set consists of a single agent $a_{\neg\mu}$, enabled when the current state of the behavior prefix contains a phase selector value of 1. The effect of the environment agent on the system state is to change the value of the phase selector from 1 to 2, and to change the values of any subset of monitored variables. The controlled agent set μ also consists of a single agent a_μ enabled when the phase selector in the current state has the value 2. The effect of the agent a_μ on the current state is the union of the effect of a transition in τ_{eq} enabled by the current system states and variables, and the change of value of the phase selector from 2 to 1.

This system is obviously defined using a state set and a set of agents. The set of agents contains two subsets μ and $\neg\mu$ corresponding to the controller and environment actions respectively. The agent sets are disjoint since their effects on the system state always include a different effect on the *phase selector* variable. The inclusion of the phase selector in the enabling conditions of the agents makes the selection of enabled agent set deterministic in every state of the system, and enforces strict interleaving between the controller and the environment. This mapping shows that our semantic model represents a special case of the model used by Abadi and Lamport, and that their classification of properties and their relationship between properties apply to systems designed using our model.

2.4 System Interaction and its Properties

The behavior of the controller is more complex than just the parallel execution of its components. Behavior of all components and their interaction both contribute to determine the overall behavior of the controller. In addition to the specification of every component's individual behavior, we want to explicitly define their aggregate behavior as a part of the controller specification. We can specify the acceptable and unacceptable controller behaviors by defining controller properties. The following definitions specify the relationship between properties and system behaviors.

Definition 2.5 *A* **stuttering step** *occurs when the execution of an enabled agent results in an unchanged system state. Two behaviors are* **stuttering equivalent** *iff they consist of the same sequences of system states, with different number of stuttering steps.*

Definition 2.6 *A* **property** *is a set of behaviors closed under stuttering equivalence. If a property accepts one behavior, it will accept variations of that behavior where events occur in the same order, but at different intervals.*

A designer's goal is to produce a controller that will satisfy a set of required system properties. The controller influences the system behavior by executing some of its agents enabled in a given state. The enabling conditions and effects of the controller agents define the behavior of the controller and what system behaviors will be enforced by it. These agents form the controller strategy. The strategy defines the behavior of the controller, but the behavior of the system is defined as the result of the interaction between the controller and the environment.

Definition 2.7 *A* $\mu-$**strategy** *is a partial function mapping behavior prefixes to agents in the controlled agent set* μ. *Any behavior prefix that is not mapped to an agent in* μ *must enable an environment agent, or represents the halting state for the system.*

Definition 2.8 *A* $\mu-$*outcome of a* $\mu-$*strategy* f *is a behavior* σ *such that for every behavior prefix* σ_m *followed by a* $\mu-$*agent* a_μ, *the function* f *includes* $f(\sigma_m) = a_\mu$. *A* $\mu-$*outcome is fair iff it is achieved by executing agents in* $\neg\mu$ *an infinite number of times.*

The fairness requirement eliminates infinite sequences of actions generated by the controller strategy without any environment reactions. An unfair system, according to this definition, is infinitely faster than the environment or blocks its execution; unfair systems can not be classified as being open with respect to the environment, or as being realizable in practice.

Definition 2.9 *If* f *is a* $\mu-$*strategy, then* $O_\mu(f)$ *is the set of all fair* $\mu-$*outcomes of* f.

Interaction properties are often classified as safety and liveness properties. Safety rules are generally defined as rules that specify that certain *bad* states will not occur during execution, while liveness specifies that *good* states will eventually occur. This informal and practical definition restricts the domain of safety properties because it ignores the possibility that some states may be bad only after the occurrence of some sequence of events. Similarly, liveness properties may require some preconditions to occur before they require the eventual occurrence of some state. A formal definition for these types of properties is based on the finiteness of the execution traces that violate the properties. If all violations of a given property are detectable on finite traces, that is a safety property, while the liveness properties apply only to infinite execution traces. Some hybrid properties may be violated by both finite and infinite execution traces, but all such properties can be decomposed into pure safety and liveness properties.

Definition 2.10 *A* **safety property** *may reject a behavior only if that behavior contains a finite prefix that violates the property.*

Definition 2.11 *A* **liveness property** *cannot be violated by any finite behavior prefix, but it may reject some infinite system behaviors.*

Since safety rules can be verified on finite execution traces, the preconditions of their violations must occur along those traces. That means that the occurrence of safety violations can be predicted, and corrective actions can be taken by the system and its components to avoid the violations. The same principle does not apply to the liveness properties whose violations only occur on infinite traces. A useful concept in reasoning about system properties is the safety closure [AWZ88], which defines a safety approximaton for arbitrary properties.

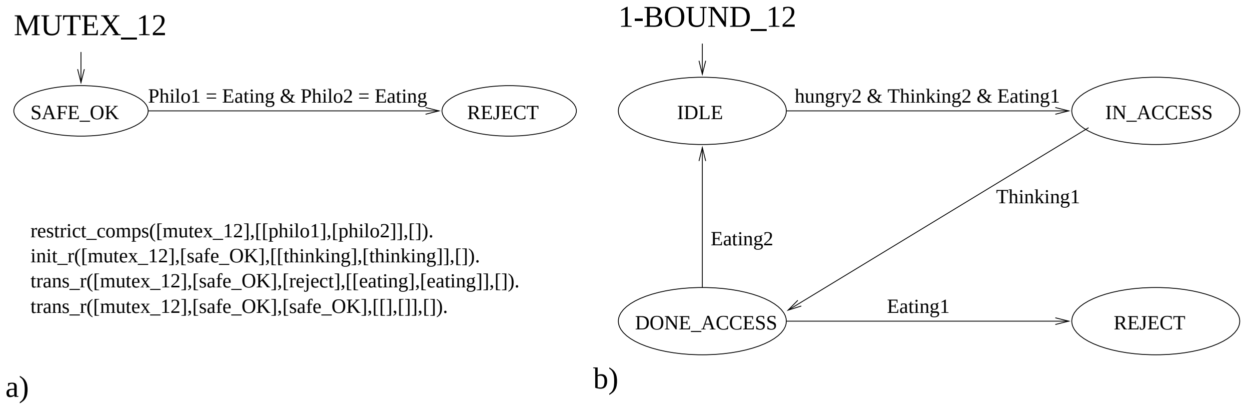


Figure 2.3: Mutual exclusion and 1–bounded overtaking for dining philosophers

Definition 2.12 *The* **safety closure** $\overline{P}$ *of a property* P *is the smallest safety property that accepts all behaviors accepted by* P.

2.4.1 Finite State Representation of Safety Properties

The safety properties in our system are defined using the same formalism used for the components, the finite state machines. Safety rules are defined as finite state machines whose behavior depends on the system states, similarly to the Esterel concept of *Observer* [BG92]. A safety rule observes the components in the sense that it uses their states to determine its enabled transitions. In practice, this means that the components and safety rules are interleaved during execution, and the states of both components and safety rule observers combined represent the system state.

Every safety property is defined as $P = (S, (C \cup A), I, \tau, R)$ where S is a set of states, C is the set of states for components observed by the property, A is the set containing all variables that the safety property can read, I is the initial state, and $\tau : (S \times 2^{C \cup A}) \to (S \cup \{R\})$ is the transition relation mapping the safety rule state and inputs to the next state. Transitions of the safety rule FSM are enabled by combinations of states of the referenced components and the values of system variables. The last parameter R specifies the rejecting state that represents the safety violation for this rule. All other states for a safety rule FSM are accepting states.

In the dining philosophers example, safety rules refer to component pairs and Figure 2.3 shows two rules that specify the interaction between two adjacent philosophers. The first rule accepts all system behaviors except those where the two adjacent philosophers are in their *EATING* state simultaneously. This rule represents the mutual exclusion requirement for two identified components and it monitors the states of those two components waiting for the safety violation. This FSM reacts to the occurrence of a safety violation by executing its transition to the state *REJECT*. The overall system behavior can be specified by asserting mutual exclusion safety properties for every pair of adjacent philosophers. Figure 2.3a) shows the mutual exclusion property both in the form of a graphical FSM and in Prolog specifications accepted by our automated synchronization tool. The predicate *restrict_comps* specifies the components whose states are used to compute the enabling conditions of the safety rule FSM, while the *init_r* and *trans_r* predicates specify the initial state and the transitions of the safety rule. The initial state can specify the expected initial states of some of the restricted components, and the transitions can use both the component states and a set of system variables in their enabling conditions.

The second property shown in Figure 2.3b) is the 1–bounded overtaking property, and it can be used to specify balanced table access by all philosophers. This property rejects those behaviors that have one philosopher eating repeatedly while one of its neighbors is hungry and waiting to eat. The enforcement of this property in a dining philosopher system implies that the starvation freedom holds, because every philosopher must be allowed to eat after waiting for each of its two neighbors to eat once. This safety property implies that the philosopher access is *fair*.

For every safety rule, we can define the set of its referenced components, $Ref(Sp)$, as the set of components whose states are used in the enabling conditions of the safety property. We can also define the set of restricted components, $Res(Sp)$, as the set of components whose behavior may be restricted by the property. The referenced components set contains all components whose state is used in some transition of the safety rule, while the restricted components set contains those referenced by some transition to the *REJECT* state. The component sets defined for the mutual exclusion and 1–bounded overtaking are the following:

$$Ref(Mutex12) = \{philo1, philo2\}, \qquad Res(Mutex12) = \{philo1, philo2\}$$

$$Ref(Bound12) = \{philo1, philo2\}, \qquad Res(Bound12) = \{philo1\}$$

We can also distinguish between several types of transitions within safety properties. The transitions leading to the *REJECT* state are the *violating transitions* and they occur when the behavior of the system becomes unacceptable by the safety property. If a state has some outgoing violating transitions, all other outgoing transitions from that state are called *synchronization transitions*. Synchronization transitions may remove a restriction on the execution of some component, and allow that component to advance if it was in a delayed state. All other transitions in a safety rule are *observer transitions* because their purpose is to make the safety rule follow the behavior of the system. The mutual exclusion property contains only one *violating transition* because it defines an invariant condition. In the case of the 1–bounded overtaking property, the transition from *DONE_ACCESS* to *REJECT* is the violating transition while the one from *DONE_ACCESS* to *IDLE* is a synchronization transition. Transitions to and from the state *IN_ACCESS* are the observer transitions for this property.

2.5 Realizability and Receptiveness

Safety properties classify system behaviors based on their membership in the property behavior sets. Acceptable behaviors belong to the intersection of the safety property behavior sets, and the unacceptable ones violate one or more safety properties. Unacceptable system behaviors violate a safety rule due to the occurrence of some property–specific violating event, whether caused by the controller or by its environment. Safety properties whose violations are caused by environment agents may be independent from the controller and thus impossible for it to enforce. Making these safety properties a part of system requirements, makes the system unrealizable. The system is *unrealizable* because its requirements can be violated regardless of the controller design. An example of an unrealizable safety property is a requirement for freedom from user reset. This property is violated when the user resets the system, and there is no way the controller can prevent it, so it can't satisfy the property. In order to automatically enforce system properties, we must be able to distinguish between properties that can be enforced by the controller and those whose violations may be unavoidable. The following definitions introduce the concept of property realizability.

Definition 2.13 *A $\mu-$strategy* f **satisfies** *a property* P *iff $O_\mu(f) \subset P$. A* **winning strategy** *satisfies all required properties for a system.*

Definition 2.14 *The* **realizable part** *$R_\mu(P)$ of a property* P *is the union of all sets $O_\mu(f)$, such that* f *is a $\mu-$strategy and $O_\mu(f) \subset P$*

Definition 2.15 *A property* P *is $\mu-$***realizable** *iff $R_\mu(P)$ is nonempty.*

There are some safety properties whose violations are caused by environmental events, but only after specific system behaviors. While these properties are realizable, they lead to problems in system testing. These properties accept some behaviors that can only be an outcome of incorrect controller strategies because the environment does not exhibit its worst case behavior. The problem is that the same controller behaviors can result in safety violations, when the environment executes the violating actions. In order to enforce those properties, the controller strategy must be restricted to generate a set of outcomes that is a subset of the property.

We have shown that safety properties can be unrealizable, but even the enforcement of realizable ones may require severe restrictions of the component and system behavior. The severity of the necessary restrictions may be so great to prevent the system from fulfilling its functional requirements. One such property is the requirement that reset does not occur while the system executes some important action; it is trivially realized by not allowing the system to ever execute the action. Receptiveness provides a tool for distinguishing between realizable and unrealizable safety properties. Distinguishing between these properties enables us to use the automated synchronization method only for the properties that can be enforced with the preservation of system functionality.

The intuitive distinction between these properties is that the unrealizable ones require restrictions on the environment events, while the realizable properties can be satisfied by controller strategies. Dill [Dil88] introduced the notion of receptiveness as the lack of restrictions on external events. The term *receptive* property suggests that the property is not influenced by the environment, regardless of its behavior. Abadi and Lamport applied this notion to the system properties, defining a property to be receptive if all behaviors it accepts belong to outcome sets of winning strategies. They also proved that a receptive safety property can only be violated by a controlled event, making receptiveness for safety properties equivalent to the limitation of constraints to the controller actions. The following definitions formally introduce the notion of receptiveness and its relationship with controller behavior.

Definition 2.16 *A* $\mu-$**receptive property** P *is equal to its realizable part* $R_\mu(P)$.

The name for receptive safety properties is derived from the relationship between the strategies that enforce those properties and the environment. Any strategy that satisfies a receptive safety property does so regardless of the environment behavior, meaning that it is prepared to react to the environment events without restricting them. This strategy treats the environment as an unconstrained input.

The receptiveness of safety rules is a theoretical concept, but it has profound implications on the realizability of properties. The realizability of all properties is based on their nonempty realizable part. Since the realizable part is a set of behaviors it is also a property. The following theorem shows the relationship between the realizability and receptiveness for arbitrary system properties.

Theorem 2.1 *For any property* P *defined as a set of acceptable behaviors, its realizable part* $R_\mu(P)$ *is a receptive property.*

Proof: *Proof in [AL93]*

This theorem shows that for any realizable property, there exists a receptive property that must be satisfied by any implementation that satisfies the original property. The importance of the receptive properties is even greater in the safety domain, because any receptive safety property constrains only the controlled agent set.

Theorem 2.2 *A receptive safety property constrains at most the controller agents.*

Proof: *Proof in [AL93]*

A corollary to theorem 2.2 is that violations of receptive safety properties are always caused by a controlled agent. The controlled agents in our semantic model represent groups of component transitions and one or more of the components must cause the event that triggers the safety violation. By delaying the components that cause the violation, the violation itself is delayed while the other components may proceed with their execution. If the system state changes while components are delayed, the safety violation preconditions may no longer hold, thus allowing the delayed components to proceed while preserving the safety of the system. Any receptive safety property can be enforced by restricting the behavior of components that may cause its violations. This establishes a basis for the enforcement of receptive safety rules by automated synchronization of components.

2.6 Receptive Safety Properties

The previous sections represent a theoretical overview of the behavior based system properties. We now need to show some extensions that apply specifically to the safety properties and their realizability. These theorems are simple extensions to the work in [AL93], but were not clearly expressed there, probably because their goal was to prove composition of working components rather than deal with automated property enforcement. The following theorems show that any realizable safety property can be represented in the form of constraints on system component execution.

Theorem 2.3 *The realizable part of a realizable safety property is a safety property.*

Proof:

Abadi and Lamport proved that the realizable part of a property may be represented as

$$R_\mu(P) = \overline{R_\mu(P)} \cap P$$

where $\overline{R_\mu(P)}$ *represents the safety closure of a property. The property* P *is a safety property so*

$$\overline{R_\mu(P)} \subseteq P$$
$$R_p(P) = \overline{R_p(P)}$$

The realizable part of a safety property is equal to its safety closure, so it is a safety property as well.

Theorem 2.4 *Realizable parts of safety properties are receptive safety properties and constrain at most the component actions.*

Proof:

Theorem 2.1 shows that the realizable part of any property is receptive. Theorem 2.3 shows that the realizable part of a safety property is a safety property. Since the realizable part is a receptive safety property, by theorem 2.2 it constrains at most the controlled agents. The constraint on the controlled agents maps to constraints on the components whose actions combine to define those agents.

The last theorem shows that the realizable parts of safety properties can be enforced by constraining components whose behavior contributes to the safety violations. We will show how the conditions for these constraints can be computed from the description of the components and receptive safety rules using simple behavior analysis. We will also show how the constraints can be incorporated into executable applications that enforce multiple receptive safety properties.

2.7 Enforcement of Receptive Safety Properties

The specification of system components defines a controller strategy *f* where components execute one enabled transition in every iteration. This strategy produces a set of outcomes that may or may not satisfy the system requirements. We will show how receptive safety properties can be enforced by producing a modified controller strategy *fs* where the components synchronize to enforce the properties. Theorem 2.2 shows that a receptive property can only be violated by controlled actions, and therefore a controller strategy that avoids the violating actions will satisfy the properties. We can analyze the system behaviors generated by the strategy *f*, find the preconditions of safety violations, and use them to enable the modified agents that enforce the safety properties.

2.7.1 Possible and Reachable Safety Violations

We define a safety violation to be *possible* if the states that cause the safety violation do exist in the referenced components. This means that we can select a combination of states for the components and the safety rule, and a combination of monitored variables that cause the components to execute the transitions that violate the safety. A state is *reachable* if it is contained in some behavior in the outcome set $O_\mu(f)$. Obviously every reachable safety violation is possible, but some possible violations may not be reachable.

The possible safety violations have very little meaning in the context of system verification, because every receptive safety property will have possible violations. Only a proof of their reachability means that the system is unsafe, and conversely only a proof that they are unreachable validates the safety of the system. In the context of safety enforcement, it is not necessary to know whether a particular safety violation is reachable, since our goal is to make it provably unreachable. The distinction between the possible and reachable safety violations is important because static analysis can detect the possible violations while reachability analysis is required for the reachable ones. The use of possible safety violations makes it possible to analyze and synchronize systems whose complexity makes reachability analysis prohibitively expensive, due to their complexity.

If static analysis detects a possible safety violation state, and we produce a new strategy *fs* that activates a different agent when its precondition holds, we guarantee that the safety violation is unreachable in the behaviors belonging to the outcome set $O_\mu(fs)$. This applies to all safety violations, regardless of whether they were reachable in the system controlled by the strategy *f*. If a safety violation was unreachable with strategy *f*, its precondition may never be satisfied, and the modified agents will not be executed.. This shows that synchronizing a system to avoid possible but unreachable safety violations results in a system that satisfies the safety properties and preserves as much of the original strategy as possible.

2.7.2 Synchronization by Delayed Transition Mechanism

The word synchronous is a combination of the Greek words *syn* meaning same and *chronos* meaning time. It describes certain events as occurring at the same time or with the same periodicity. Synchronization is the process of making the events occur simultaneously. In the domain of computer science, synchronization has a broader meaning of making events occur at appropriate times or intervals, as defined by the system specification. In the domain of concurrent and distributed computing, synchronization includes the adjustment of local clocks within specified tolerances; synchronization also refers to the system activities whose goal is to influence the temporal ordering of certain system events.

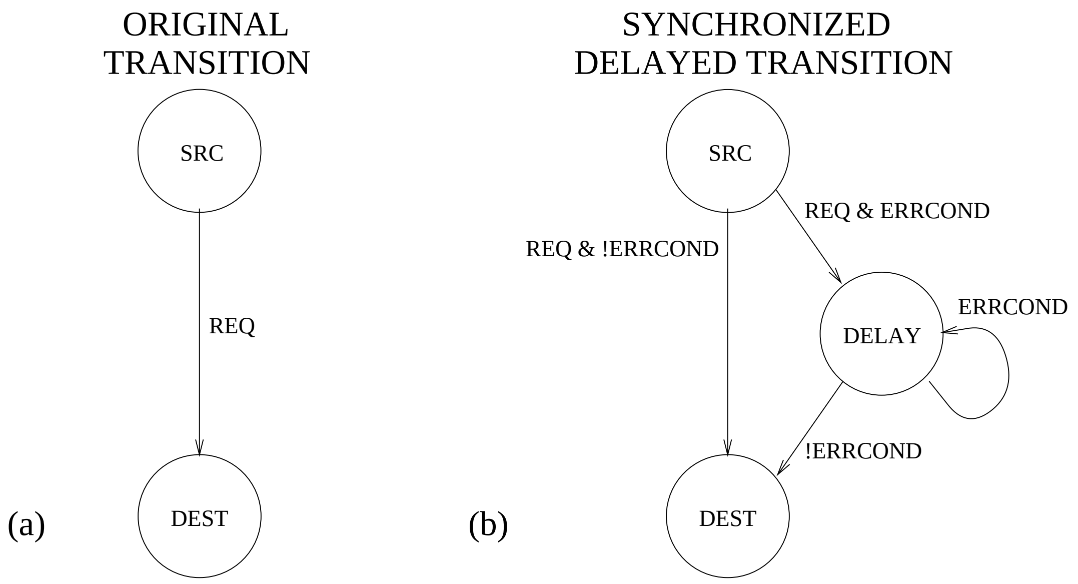


Figure 2.4: Overview of delayed transition implementation

One example of synchronization are barriers whose purpose is to make components execute a certain operation only when all components are ready. Conversely, locks guarantee that some sets of events do not occur simultaneously while their ordering is unrestricted. More general synchronization tools like semaphores or blocking messages can enforce both inclusion, exclusion and arbitrary ordering of events. The common element for these synchronization mechanisms is the delayed execution concept, where the component actions are blocked as long as necessary to enforce some system behavior.

The synchronization mechanism in GenEx is based on the delaying of component transitions. Delayed transitions are implemented by introducing one additional state where the FSM blocks as long as the completion of the transition may cause a safety violation. Figure 2.4(b) illustrates the implementation of a delayed transition for the transition in Figure 2.4(a). If the safety analysis finds that the transition could lead to a safety violation, the delayed transition is added to block the component whenever the safety violation preconditions hold. The enabling condition of the original transition **REQ** is combined with the condition **ERRCOND** that is the precondition of the detected safety violation, and the resulting conditions enable the delayed transition. The delayed transition leads to a state *DELAY* that, for safety monitoring purposes, represents an extension of the source state *SRC*. The original transition from the state *SRC* to the state *DEST* will be enabled only when its enabling condition **REQ** is satisfied and the safety violation precondition **ERRCOND** is not.

The transition from the delayed to the destination state will occur only when there is no potential for a safety violation. The transition occurs even if the original enabling condition of the transition no longer holds because the semantic of the component specification is that the enabling condition, once activated, determines the next state for the component. Since the delayed state is considered to be the same as the source state, the transition to the *DEST* state must be completed to satisfy the functional specification of the component.

2.8 Subclasses of Receptive Safety Properties

Receptive safety properties are defined as safety properties whose violations are caused by the component actions. We can distinguish two basic types of violations caused by the components. The simplest type is a sequencing violation, caused by an action of one component independently from the actions of other components. A more complex type of violation is caused by a combination of states for several components, and we call that a limited resource access violation. If all violations for a given safety property are of the sequencing type, that property is called a *sequencing property*. Similarly, if all violations of a safety property are of the limited resource access type, that property is called a *limited resource access property*. Examples of both types of properties are given in Figure 2.3, where the 1–bounded overtaking is a sequencing property and the mutual exclusion is a limited resource access property. Properties including both types of violations can be decomposed into two properties of the individual type.

The 1–bounded overtaking property references two components, but the only transition to the *REJECT* state is caused by the first philosopher in state *EATING*. This property is violated if the first philosopher starts to eat while the safety property FSM is in the state *DONE_ACCESS*. The violation by a single component makes the 1–bounded overtaking a sequencing property.

The mutual exclusion property violations require the simultaneous *EATING* by both philosophers, and this condition can occur when one philosopher joins another that is already eating, or when two philosophers get hungry at the same time and simultaneously enter the *EATING* state. If one philosopher is eating, the mutual exclusion is enforced by delaying the other one until the fork between them is free. When both philosophers try to access the *EATING* state simultaneously, the mutual exclusion can be enforced by delaying one component and allowing the other one to proceed. The choice of delayed component should be nondeterministic and fair.

2.8.1 Synchronization for Different Types of Receptive Properties

Sequencing properties specify the global behavior in terms of allowed sequences of events, by rejecting the occurrence of individual component states. This means that a transition to the *REJECT* state for the safety rule is enabled by the occurrence of a particular state for one component, independently from the behavior of other components in the system. The only way to prevent this safety violation is to delay the component until the safety rule monitor leaves its current state. The synchronization condition that enables a delayed transition of the first philosopher to enforce the 1–bounded overtaking is:

$$Delay = (philo1 = thinking) \wedge hungry1 \wedge (bound12 = access\_done)$$

While a single rule may impose restrictions on the behaviors of several components, these restrictions depend only on the state of the receptive safety property FSM and the respective components. The safety violation is caused by a single component regardless of any simultaneous actions by the other components. Every possible safety violation of a sequencing property can be avoided by delaying one system component.

The subclass of limited resource access properties specifies requirements like the mutual exclusion, whose violations may be a result of simultaneous accesses by two or more components to a critical section where exclusive access is required. Our model assumes synchronous execution of a single transition by all components, so two or more of them could enter the critical section simultaneously, thus causing a safety violation. This safety violation would not occur if only one component had entered its critical section, so not all components must be delayed to enforce the property. In the case of the dining philosophers, a single philosopher making the transition from *THINKING* to *EATING* does not cause a violation, but two adjacent philosophers making the same transition do. We will use this example to describe the method of computing the component synchronization conditions for limited resource access rules.

The precondition for the safety violation by simultaneous entry into the *EATING* state consists of the state that precedes the violation and the conditions that enable the transitions to occur. The safety violation precondition for one pair of adjacent philosophers is given below:

$$SV\_Cond12 = ((philo1 = thinking) \wedge (philo2 = thinking) \wedge hungry1 \wedge hungry2$$

When the safety violation precondition holds, some of the components must be delayed in order to preserve the safety of the execution. For limited resource properties, all components but one referenced in the violation transition are allowed to proceed without causing the violation.[1] In the case of mutual exclusion of philosophers, that means that one component is allowed to access the *EATING* state while the other one is delayed waiting for its turn. The synchronization conditions of the two philosophers need to be consistent so that exactly one component is delayed to preserve this property. We derive the individual synchronization conditions from the safety violation precondition having three rules in mind:

- Complete coverage of the violation precondition.
  The union of individual components' synchronization conditions must cover all system states that satisfy the violation precondition. Incomplete coverage means that in some cases the safety violation might occur regardless of the synchronization because no components are delayed. If two components must be delayed to prevent a safety violation, then the complete coverage principle means that any state that satisfies the violation precondition, enables the delayed transitions for at least two components.

- Minimal coverage of the violation precondition.
  This principle implies that no unnecessary components will be delayed in the synchronization for any particular safety rule. If one delayed component satisfies the property, the intersection of its synchronization condition with those of the other components will be empty. Similarly, if only two components must be delayed, the intersections of synchronization conditions for any three components will be empty.

[1]Every violating transition is enabled by a combination of component states, and if one component is not in the specified state, the violation does not occur. Compound safety properties such as mutual exclusion of multiple components are decomposed into simple pairwise exclusions. Every pairwise exclusion requires the delay of one component, but the compound effect of all properties is that all but one component must be delayed.

- No component can be predefined for delays in particular system states.
  The complete and minimal coverage principles can be trivially satisfied by picking one (or more, as necessary) component, and delaying it every time the violation preconditions hold. This creates an asymmetric system, and can lead to implementations with a number of undesirable characteristics, such as unnecessary deadlocks, livelocks or starvation of components. The synchronization conditions need to satisfy some form of fairness in the selection of delayed components.

In the case of the philosophers and their mutual exclusion, one delayed philosopher is always sufficient to guarantee the preservation of the safety property referencing two philosophers. The relationship between the individual synchronization conditions and the violation precondition is specified by the following system of equations:

$$Delay1 \vee Delay2 = SV\_Cond$$
$$Delay1 \wedge Delay2 = \mathbf{false}$$

These equations correspond to the complete and minimal coverage requirements in the case of one delayed component out of two. Similar equation systems can be created for any other combination of numbers of total and delayed components. A number of different values for **Delay1** and **Delay2** would satisfy this equation set, but the third principle(fairness) requires them to occur with equal frequency for all combinations of safety violation preconditions. This implies that we need a nondeterministic selection facility that enables the delayed transitions for either component. A simple solution to this problem is the introduction of a nondeterministic selection signal, *ndet_prio_21* that represents the nondeterministic relative priority of the components. Then the delay conditions for the two components are:

$$Delay1 = SV\_Cond \wedge ndet\_prio\_21$$
$$Delay2 = SV\_Cond \wedge \neg ndet\_prio\_21$$

These two delay conditions satisfy the equation system, and also satisfy the third principle, since no component is treated preferentially regardless of the system state. The nondeterministic signal *ndet_prio_21* enables the delayed transition of the first philosopher when it has the value *true*, and thus allows the second philosopher to advance into the state *eating*. The second component is delayed and the first is allowed to proceed when this signal has the value *false*. This signal amounts to a decision on the relative priority between the first and second components, thus its name. Similar signals may be used to order all pairs of components and produce a total ordering of all components.

This structure of synchronization conditions can be used even for the complex safety violations caused by the occurrence of three or more simultaneous actions. If k components can proceed but the $(k+1)$th must be delayed, the relationship between the individual synchronization conditions and their specification are as follows:

$$Delay1 \vee Delay2 \vee ...Delay(k+1) = SV\_Cond$$
$$Delay1 \wedge Delay2 \wedge ...Delay(k+1) = \mathbf{false}$$
$$Delay_1 = SV\_cond \wedge ndet\_prio\_21 \wedge ... \wedge ndet\_prio\_(k+1)1$$

This is the most complex synchronization condition that can occur with our specification of safety rules, because the enabling conditions of the violating conditions are simple conjunctions of component states. For any such conjunction it is sufficient to delay one component to guarantee that the safety property is preserved.

2.9 Guidelines for System Definition

Our method modifies given component specifications by delaying some of their actions in order to preserve the safety of the system behavior. Behavior of the modified components is stuttering–equivalent to the behavior of the originals, but it is unlikely to be identical if the component has the potential for causing safety violations. The synchronized application is guaranteed to satisfy the specified receptive safety rules, but inconsistent specifications of components and safety rules may lead to the occurrence of deadlocks or the undetected occurrence of safety violations.

The system specification must be partitioned into components and receptive safety rules, and this decomposition is determined by the nature of the specified functionality. Components should specify the functional aspect of system behavior, while the receptive safety properties specify the constraints on interactions between the components. Our method is most effective in enforcing partial ordering of actions for separate components. Enforcement of simultaneous actions or selection of actions requires specific component design. Behavior requiring limited time reaction can not be specified as a receptive safety rule, so it must be implemented at the component level or by enforcing its realizable part. Time–dependent properties can be automatically detected as potentially nonreceptive, and the user warned of their existence.

The following list illustrates some of the basic guidelines for system decomposition and specification.

- Components are related to one controlled signal, or a set of closely related signals. Signals are closely related if they change value simultaneously or if their intervals of activity are mutually exclusive. Generally, the signals controlling one physical device, or one aspect of the behavior of a complex device are closely related and should be specified by one component. This way of decomposing systems produces simple components whose interaction is specified by simple receptive safety properties.

- States embody the decision–making capabilities of a component. A component stays in a state until it can select another state to go to. Components should employ a greedy strategy in their behavior decisions, their transitions should be enabled as soon as the state of the system and the environment is such that the transition will eventually be completed. The automated synchronization will ensure that the selected transitions are not completed as long as that may violate the receptive safety properties.

- The components should be defined using a state oriented semantics, where their transitions have no meaning to the overall system behavior. The receptive safety properties are defined as sequences of states, and only component states can be used to detect safety violations. In cases when individual transitions are important for the system behavior, the safety properties must detect their source state, and then reject their destination state until the transition becomes acceptable.

- Controller actions with limited time requirements must be specified as transitions of individual components. In order to guarantee that the behavior is preserved in a synchronized system, the safety rules must not impose constraints on the completion of those transitions.

- Safety properties specify the interpretation of the interaction between two or more components. A safety property is defined using the states of components to enable its transitions, and it reaches the predefined REJECT state when the component interaction is unacceptable.

- If a component transition is delayed by some state of a safety rule, the actions of other referenced components must be able to make that transition acceptable to the safety rule. If this condition is not met, a safety rule may block a component forever thus removing its function from the system.

Chapter 3

The GenEx Toolset

We use receptive safety properties to specify acceptable interactions between system components. The receptiveness guarantees that the properties can be enforced by synchronizing the components. We developed a set of analysis and code generation procedures, integrated in the GenEx toolset, that compute the necessary synchronization conditions and produce an executable implementation of the desired controller. This chapter contains the description of the GenEx toolset and the design and development methods it supports.

The GenEx toolset contains a number of independent tools whose functionality can be integrated to analyze systems and produce executable applications. The analysis tools detect the violations of safety rules and compute the synchronization conditions for individual components. Integration and code generation tools combine the component specifications with the associated synchronization mechanisms and produce executable C code or formal models for the system. The generated code links with an environment dependent runtime support library that implements the underlying semantic model. Additional verification and analysis tools can detect if the asserted system properties are nonreceptive or inconsistent with the referenced components. The functionality of these tools is integrated using an automated script generator.

In the Section 3.1 we describe the basic steps in the production of automatically synchronized concurrent systems. The following three sections describe the tools used to analyze the component behavior with respect to the safety properties and compute their synchronization conditions. Section 3.6 describes the code generation algorithm and its results, while Section 3.7 describes the runtime support library. Finally, Section 3.8 introduces a number of auxiliary tools such as the script generator that integrates the functionality of the other system tools, verification algorithms for system consistency, and a runtime visualization tool.

3.1 System Development Using the GenEx Toolset

GenEx tools automate the synchronization and integration of controller components, and produce an implementation of the controller that satisfies the given receptive safety properties. The system designer must, manually or using other tools, generate the component specifications and the receptive safety properties. The components and safety rules are specified using finite state machines, and the safety rule transitions are enabled by the states of the components whose interaction constraints they describe. The following list illustrates the major steps in the system design using automated synchronization supported by GenEx, with the pluses denoting the steps performed by the automated tools.

– **Define System Components.** The components are concurrent elements of the controller, and their goals are largely independent. In the case of the dining philosophers, one component models each philosopher, and the guidelines in Section 2.9 describe how the components should be defined in more complex examples. Components must be defined by finite state machines defined in a tabular form, either with their local signals or using the global signal set for the system.

– **Specify System Requirements as Receptive Safety Properties.** The system requirements specify a set of properties that the controller strategy must satisfy. These properties can be automatically enforced only if they are represented in the receptive safety form. For all realizable safety properties, their realizable part is a receptive safety property that can be enforced automatically. Some liveness properties contain behavior subsets that are receptive safety properties, and those liveness properties can be enforced automatically using GenEx. Required properties can be analyzed independently to find receptive safety properties that represent them.

\+ **Verify Consistency and Receptiveness of the Properties.** This automated step compares given receptive safety properties with the system components and attempts to verify that the enabling conditions for the property transitions use existing states of the referenced components. Safety properties are receptive if they impose no restrictions on the behavior of the environment, represented by the monitored variables. Time is also a monitored variable, and the receptive properties must not be violated by time–related events. GenEx toolset includes a mechanism that verifies all of these factors, and can even suggest possible rule modification that eliminate problems like time–dependence.

\+ **Analyze and Enforce the Receptive Safety Properties.** The receptive safety properties reject some controller behaviors, and system behavior analysis detects those violations and finds the components that cause them. Components are modified to delay the actions that cause the safety violations, making the violations unreachable. The analysis of every safety property is independent and global analysis is not required. Total complexity of this process is equal to the sum of the analysis complexity for the individual safety properties.

\+ **Generate a Controller Implementation.** GenEx includes the capability to generate a controller implementation in C, whose structure makes it easy to link to data processing code segments and to system interface functions. The generated code makes no assumptions about the execution environment and supports different implementations, ranging from single centralized process to distributed execution with multiple processes.

+ **Generate a Controller and System Model.** The formal model of the system or its parts can be used to verify the correctness of the design using the symbolic model checking tool SMV [McM93]. While GenEx synchronizes the components to enforce the given receptive safety properties, and the generated controller is guaranteed to satisfy them, other system requirements may be violated. If the receptive safety properties are not consistent with each other or with the components, the intersection of their behaviors may be empty or limited to behaviors where sets of components are deadlocked. A partial or total system model can be generated using the SMV notation to verify that subsystems execute correctly together.

Software development using GenEx is applicable to component–based systems where the component interaction is in the domain of control behavior rather than the data processing. Component design should be partitioned into a control–oriented structure and data processing segments related to specific control actions. Component control structure is synchronized with the other components, and the generated code includes calls to the data processing code segments supplied by the designer.

Automated synchronization of control systems produces sets of modified components that include the functionality of the original components, but also satisfy the receptive safety rules that specify the component interaction. The code generated for the synchronized components can be linked with a runtime support kernel, and compiled into an executable application.

3.2 Automated Computation of Synchronization Conditions

The enforcement of receptive safety properties requires individual components to delay their actions when completion of those actions may violate the properties. To avoid having a centralized scheduler that becomes a bottleneck for distributed systems, we enable the components to determine when their actions are safe to complete. Every transition that has the potential to violate some receptive safety property is modified to satisfy a minimal delay required to preserve the property. When the system state satisfies the preconditions of a safety violation, the enabling conditions for the delayed transitions of one or more components become true, making those components stay in their previous state. Delay conditions for the components are constructed to become true only when the completion of the respective transitions may cause safety violations, and to remain false when the behavior of a component is safe.

Our method analyzes the system behavior and, upon finding safety violations, modifies individual components to make those safety violations unreachable. The modification of any component results in a modified reachability graph for the whole system, and possibly makes new safety violations reachable. Since the system state space is finite, the number of possible safety violation states must also be finite. Repeated execution of the analysis algorithm would lead to a fixed point where all reachable safety violations have been detected and corrected, and the system behavior satisfies the receptive safety rules. The problem with such an approach is that executing the reachability analysis an unspecified number of times may require unacceptable amounts of time and CPU resources, and is thus not a practical approach.

We modify the components before the analysis by adding nondeterministic delayed transitions to any transition that may cause any safety violation. By analyzing the behavior of the modified components, our system can detect all reachable safety violations, even those that become reachable only after component behavior is modified to prevent other violations. The complexity of the analysis for components with nondeterministic delayed transitions is obviously bigger than it would have been for the original component specifications without delays. Since many if not all of the delayed transitions must eventually be added to the components in order to prevent the safety violations, the complexity of the analysis is similar to the last analysis-correction iteration if the fixed point approach had been used.

We have developed two different methods for analyzing the system behavior and computing synchronization conditions. The first approach is based on reachability analysis of the system behaviors, its advantages are the capability for exact determination of safety violations and for the detection of reachable deadlock states. The other approach is static, based on backward tracking of combined violation transitions. The next two sections describe these synchronization algorithms.

3.3 Reachability Analysis of Receptive Safety Rule Violations

Reachability analysis of system behavior is a well known method for verifying correctness of a system with respect to its safety requirements [CES86]. The strength of this method is that it naturally produces proof for the detected safety violations, by tracing back along the path from the starting state to the violating one. The safety violation sequence helps the user in the reconstruction of the causes that lead to the violation, making this method useful in testing and debugging. Reachability analysis is limited in use to relatively small systems whose state space can be represented and analyzed efficiently. Complexity of the state space is, as a general rule, proportional to the product of the sizes of its components, thus making the reachability analysis an exponentially large problem for systems with numerous components.

We use reachability analysis in the context of safety enforcement, specifically for receptive safety properties. The controlled violations of receptive safety properties mean that every property defines a subset of components that can violate it, and those are the components that need to be synchronized to satisfy the properties. We can restrict the reachability analysis to the set of potentially violating components, and thus reduce the complexity of the reachability analysis graph to a fraction of the overall system complexity. The complexity of the reachability graph for any single safety rule depends only on the number and complexity of components referenced by that rule.

```
- Initialize state set S with the initial state I, representing the initial
        states for all components and the safety rule monitor
- Initialize transition set T as empty
- Mark state I as leaf
- Loop
  - Get a leaf state s in S
  - Foreach combination t of nondeterministic component
        transitions enabled in state s
    - Find enabling condition c that satisfies the enabling
        conditions of all component transitions in t
    - Create a state comp_st containing the destination states of
        transitions in t, with the monitored variable values specified in c
    - Identify the safety monitor state sm in state s, and compute its next
        state sm_next that occurs as a reaction to state comp_st
    - Create a state new_state by combining the comp_st with the
        new safety monitor state sm_next.
    - If new_state is not in the set S, add it to the set and mark it as leaf.
    - If state sm_next is REJECT, mark new_state as processed_state.
    - Add the tuple(s, c, t, new_state) to transition set T.
  - Endfor
  - Mark state s as processed_state
- Until S has no more leaf nodes
```

Figure 3.1: Finite State Machine Composition Algorithm

3.3.1 Reachability Graph Construction Algorithm

The descriptions of components referenced by the safety rule are combined to generate an equivalent FSM representation for the component behavior, as described in the semantic model in section 2.3.2. The component reachability graph is combined with the FSM representation of the receptive safety rule to find the safety violations states. The combination algorithm is given in Figure 3.1 in a pseudocode form. This algorithm conducts a breadth–first search of the system state space, expanding all states except those that violate the safety property. Since these safety violation states will be made unreachable by the component synchronization, their successor states will also be unreachable and are thus irrelevant for the analysis.

The expansion of every system state is driven by the number of individual component transitions enabled in that state. For every combination of component transitions, the algorithm attempts to find an enabling condition that activates all selected transitions. If the enabling condition exists, the selected combination of transitions can occur during execution, and the resulting state is added to the reachability graph. We use the selection of component rather than the selection of input combinations in order to reduce the complexity of the analysis. The input consists of a number of independent monitored variables, so there are up to 2^{num_vars} combinations of their possible values. When combining transitions, the complexity for each state is limited by the product of the number of outgoing transitions for each component state. The number of outgoing transitions is at least 2 for every state since every state has a default self–loop transition, but the number of components referenced by the receptive safety rules tends to be small, thus keeping the number of combinations within bounds for practical analysis. The enabling condition for a selected set of transitions is the intersection of their individual enabling conditions.

The resulting state must also be characterized by the state of the safety rule monitor for the enforced receptive safety property. The safety rule monitor acts as an observer of the component states, so its state is a function of component behavior. The state of the safety monitor is not a function of the component states, but of the states along the behavior prefix leading to the current state. This means that different paths leading the components to the same states may cause the safety rule observer to reach different states. An example of this is the 1–bounded overtaking property, that allows the second philosopher to enter its *EATING* state once while the first one is waiting, but rejects it when it occurs a second consecutive time.

3.3.2 Analysis and Automated Synchronization for the Reachability Analysis Method

The primary goal of the analysis phase is to find whether a safety rule is satisfied by the system and, if it is not, to compute the synchronization conditions that make the system safe. The combined reachability graph contains all reachable states of the system, including safety violation states. It also contains the predecessor states of the safety violations, and the transitions to the violation states with their enabling conditions. The predecessor of the violation state and the enabling condition of the violating transition define the precondition of the safety violation. The safety violation preconditions specify when some components must be delayed to preserve the safety. The violation preconditions do not identify the components or specify how many of them need to be delayed to preserve the safety.

All safety violations of receptive safety properties are caused by component actions. The same enabling conditions that activate a violating transition also enable its nondeterministic delayed versions that preserve the safety. By comparing a violating combined transition with one that preserves the safety under the same enabling conditions, we can identify a set of components that may contribute to cause the safety violation. Delaying all those components guarantees that the detected violating transition does not occur, but the safety may be preserved even by delaying just a subset of these components.

To identify minimal delay requirements, we must identify combined transitions in the reachability graph with the minimal difference in the number of delayed components. We are looking for a safety preserving combined transition that differs from the violating transition by the included transition of exactly one component. Further, that component should complete its transition in the combined violating transition while being delayed in the safety preserving combined transition. The component that is delayed in the safety preserving combined transition is identified as one cause of the violation, and it must be delayed to prevent the occurrence of this particular safety violation. The pattern of system states shown in Figure 3.2 illustrates the relationship between the reachable states used to identify the minimal set of delayed components. State *EE* represents the mutual exclusion violation with two adjacent philosophers in their *EATING* state, *TT* represents them in the *THINKING* state, and *ED* represents the first philosopher in the *EATING* state and the second in the *THINKING_DELAY* state. Both transitions **t** and **td** are enabled by the same condition, namely $(hungry1 \wedge hungry2)$. Combined transition **td** includes a delayed transition

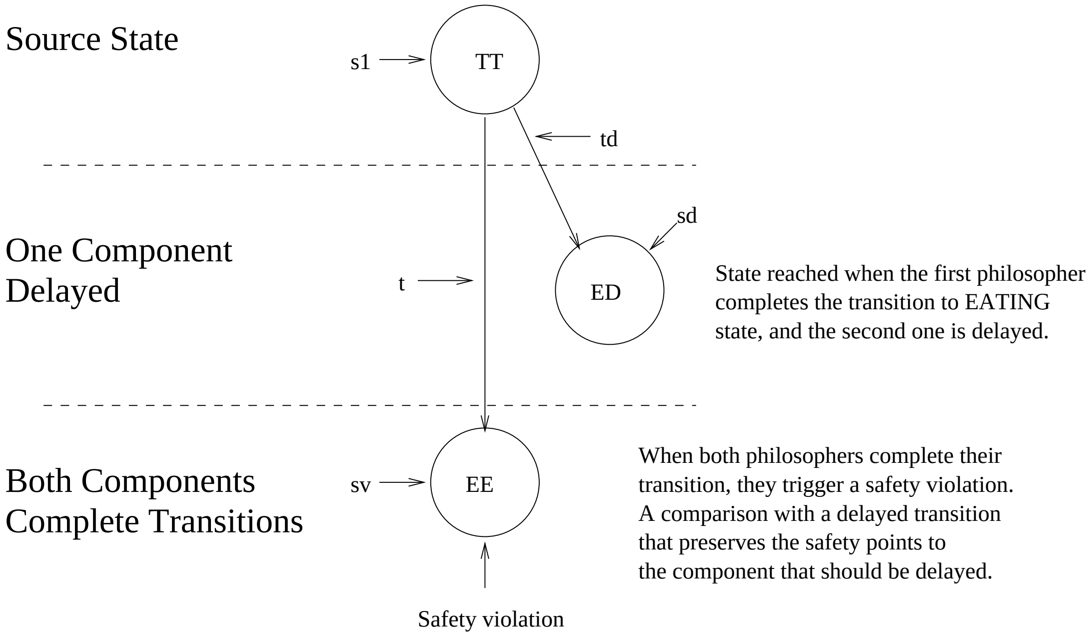


Figure 3.2: Analysis domain for one safety violation

for the second philosopher, instead of the completed transition to the *EATING* state that causes the safety violation. The comparison of these two combined transitions identifies the second philosopher as one possible cause of the safety violation, and shows that the violation is avoided by delaying the second philosopher. A separate analysis that uses another delayed version of the transition **t** will detect the conditions for avoiding this violation by delaying the first philosopher.

The algorithm that computes the component synchronization conditions is given in Figure 3.3, and it looks for this pattern of combined transitions in the system reachability graph. When it detects two such transitions, it identifies the component that needs a delay, and the minimal conditions that should enable their delay to prevent the safety violation. The algorithm uses variable names introduced in Figure 3.2 for the combined transitions and states. The enabling conditions of the combined transitions, and the invariant of the source state **s1** form the safety violation precondition.

The synchronization conditions for the component must incorporate another parameter, the non-deterministic relative priority condition described in section 2.8.1. This parameter is important for the limited resource access rules whose violations may be caused by the simultaneous occurrence of multiple component actions, while the individual actions preserve the safety. If several components attempt to enter the critical section, the choice between them must be nondeterministic to guarantee the fairness of the implementation. The comparison between a violating combined transition and its delayed version that preserves the safety results in the identification of a component that must be delayed. The identified component is said to have a lower priority than all components that complete their individual transitions in the safety preserving combined transition.

```
- Foreach safety violation state sv in S do
  - Foreach transition t in T whose destination is sv
    - Identify source state of t in s1, and its enabling condition in cv
    - Identify set of components CV whose state in the system state sv
        is a delayed state, and store the number of components in CV as nv
    - Foreach transition td in T whose source state is s1
        and enabling condition is cv
      - Identify destination state of td in sd.
      - If sd is a rejecting state then goto Next_Delayed
      - Identify set of components CD whose state in the system state sd
        is a delayed state, and store the number of components in CD as nd
      - If (nd ≠ nv + 1)or(CV ⊄ CD) then goto Next_Delayed
      - Identify component C as CD − CV
      - If any component except C uses a different transition in
        t and td then goto Next_Delayed
      - Identify the state of component C in system states s1, sd, sv
        in cs1, csd, csv respectively
      - Identify set of components CP whose state in the system state sd
        is not delayed, and is different from their state in s1
      - Create the priority condition cp that is true when all components in CP
        have higher priority than C
      - Create a transition for component C from state cs1 to csd
        with the enabling transition cv ∧ cp
      - Modify the enabling conditions of the transition from cs1 to csv
        to exclude the condition cv ∧ cp
      - :Label Next_Delayed
    - Endfor
  - Endfor
- Endfor
```

Figure 3.3: Algorithm for computation of component synchronization conditions

In the case of the two philosophers, shown in Figure 3.2, the second philosopher is said to have lower priority, and its delaying condition is computed as:

$$S\_Cond2 = hungry1 \land hungry2 \land ndet\_prio\_12$$

Another delayed combined transition exists that takes the first philosopher to its *THINKING_DELAY* state while the second one completes the transition to *EATING*. When that combined transition is compared with the violating transition, the first philosopher is identified as the component that must be delayed with the synchronization condition:

$$S\_Cond1 = hungry1 \land hungry2 \land \neg ndet\_prio\_12$$

By examining all combinations of violating transitions with their delayed alternatives, our algorithm will identify all components whose delays may enforce a safety property. It generates synchronization conditions for each of the components and the aggregate effect of all these synchronizations is that the safety violations will be avoided by delaying a minimal number of components.

3.4 Static Detection of Possible Safety Violations

The reachability analysis of component subsets detects all reachable violations of a given safety rule, and computes the necessary synchronization that will make the components satisfy the rule. The drawback of this method is that the complexity of the reachability graph may be an exponential function of the number of referenced components and their sizes. Given safety rules that reference several complex components, the complexity and the required time for analysis may become excessive for use in a practical development environment. Another potential problem with this approach is the redundancy of the generated synchronization conditions, The synchronization conditions computed for each of those states may not be equivalent and thus will all be implemented in the integrated system. Their redundancy will increase the condition evaluation overhead and decrease the performance of the application.

In the earlier discussion, we mentioned the difference between reachable and possible safety violations. We can use the possible violations to compute the synchronization conditions without constructing a reachability graph for the subset of components. The predecessor state of a safety violation and the respective violating transitions define the violation preconditions, Since the immediate violation precondition states determine the delayed components and their synchronization conditions, the previous states in the trace leading to the safety violation are irrelevant for the purpose of enforcing safety.

The static method for detecting possible safety violations is based on the backtracking of component transitions from a safety violation state. Every safety violation is specified as a transition to the *REJECT* state of the safety rule observer. The enabling conditions for that transition specify the states of individual components that contribute to cause the violation. The safety violation preconditions are all combinations of component states whose next state enables the given violation. This algorithm is given in Figure 3.4.

In the dining philosophers example, the static analysis method detects the same violation states and preconditions and produces the same delay conditions to enable the delayed transitions. The equivalent results are due to the simplicity of the components and their independence with respect to monitored variables that enable their transitions.

3.4.1 Comparison of the Static and the Reachability Analysis Method

In the preceding description of the reachability and static analysis methods, we have shown how both methods detect the safety violation preconditions for the given safety rules, and how the preconditions are used to determine the delayed components and their synchronization conditions. The difference between these two methods is mainly in their complexity and sometimes in the efficiency of the generated applications. Figure 3.5 illustrates the parts of the system behavior graph analyzed by the reachability and static analysis methods. The reachability method must generate the full reachability graph for the components referenced by the safety property and

```
- Foreach violating transition tv in safety rule Safe do
  - Identify set of components CV whose states enable the transition tv
  - Foreach combined transition t_comb of components in CV whose destination
        state enables the transition tv
    - Identify the source state s_prev and the enabling condition
        c_comb for the transition t_comb
    - Create safety violation precondition cv_pre, satisfied when c_comb
        is true and the components are in states specified in s_prev
    - Foreach component C in CV do
      - Identify set of priority components SP whose transitions
        in t_comb are between distinct states.
      - Create priority condition c_prio that encodes the nondeterministic
        priority signals when all components in SP have higher priority than C
      - Create synchronization condition for c_sync = cv_pre ∧ c_prio
      - Identify transition tc of component C in combined transition t_comb
      - Add delayed version of transition tc to component C enabled by c_sync
      - Modify the enabling conditions of the transition tc
        to exclude the condition c_sync
    - Endfor
  - Endfor
- Endfor
```

Figure 3.4: Algorithm for static computation of synchronization conditions

then identify state patterns on this graph. The static analysis method identifies all possible safety violations and uses backward tracing from the safety violation states to construct their possible predecessors. The row of encircled states in Figure 3.5 represents the possible violation predecessor states generated by the static violation analysis method. The intersection of the state spaces analyzed by these methods is the set of reachable safety violations and their preconditions, the very set of violations the system must be synchronized to prevent.

The complexity of the reachability analysis is proportional to the size of the combined behavior graph for the set of components. For large components and for safety rules that reference many non–trivial components, the complexity of the reachability graph will make this method non–viable. However, for subsystems with a few simple components, the complexity of the reachability graphs is trivial for the capacity of current computers. The reachability graph analysis can detect behavior anomalies such as deadlocks for some component subsets and alert the user.

The static violation analysis method has a significant advantage over the reachability method in the synchronization of complex systems, because their reachability graphs may be too large to be exhaustively analyzed. The static method may be preferable even in systems whose reachability graphs can be effectively analyzed, because it may generate simpler sets of synchronization conditions, without partial overlapping as in the case of reachability analysis. The main disadvantage of the static analysis approach is the potential unreachability of many possible safety violations. This can occur in systems with overlapping safety properties, systems where the components are enabled by shared monitored signals, or those using automated synchronization in combination

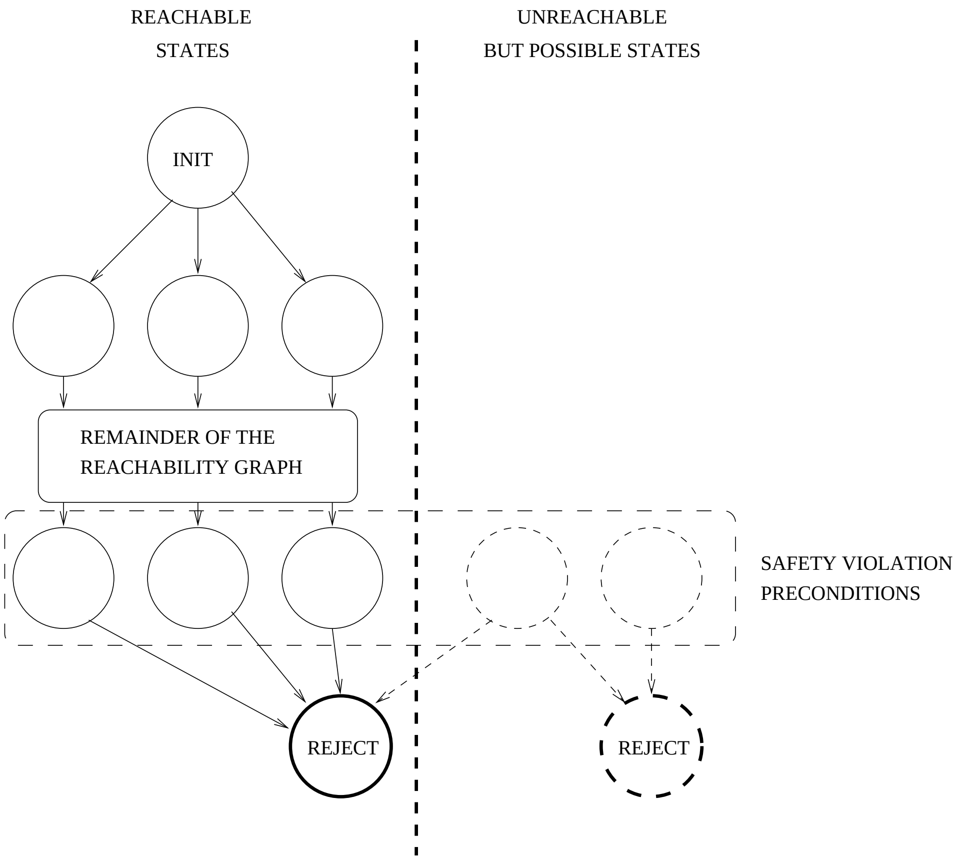


Figure 3.5: State space required for the two analysis methods

with other synchronization mechanisms. If the component behaviors are strongly correlated, some possible safety violations may not be reachable and the synchronization conditions computed for their preconditions are unnecessary. These unnecessary synchronization conditions will never be enabled since they only occur on unreachable states, so they will have no influence on the system behavior. The existence of unnecessary synchronization conditions results in increased execution overhead.

The main trade–off in the selection of the analysis method depends on the size of the analyzed state spaces, as shown in Figure 3.5. If the state space of possible safety violations is much smaller than the state space of the reachability graph, the static analysis method is faster and it may be the only option if the complexity of the reachability graph exceeds the available resources. If the state spaces are of comparable size, the reachability analysis algorithm may produce more efficient synchronization code. Reachability analysis can also be used to verify that the synchronized system is deadlock–free, or that it satisfies some time–dependent properties.

3.5 Formal Model Generator

The GenEx toolset produces integrated systems that satisfy given sets of receptive safety rules. The synchronization does not ensure the correctness of the system with respect to any other types of properties, except those that may be implied by the satisfied receptive properties. Model checking of the system is necessary to verify the functionality of the synchronized system. If the given safety rules have some real–time requirements, or if a set of safety rules is inconsistent with the system, the synchronization may result in the existence of deadlocked states. Another possible source of deadlocks is the circular dependence between components synchronized for limited resource access properties. Liveness and real–time properties are not enforced, and may be violated by the delays used to synchronize the components. The synchronized system can even exhibit safety failures if the synchronization rules were incorrectly specified, or missing. Formal system verification is a useful technique that can have a major influence on the production of correct, functional and reliable systems. GenEx produces formal models of the system or its subsystems to allow the users to verify the correctness of the generated synchronized systems with respect to the implicit and explicit correctness criteria.

GenEx generates models of the synchronized system in the SMV [McM93] notation that allows symbolic model checking; However, the complexity of the synchronized systems quickly grows out of the range that can be verified in practice, even with symbolic model checking. We address the system complexity problem by providing a flexible tool that generates different versions of the model, and works with whole or partial systems. The model versions differ by their time and memory requirements, and by the fidelity of system representation. The partial model generation capability provides the possibility to verify properties on subsets of components; this capability is very important for large systems where some properties may not depend on the behavior of all components.

The toolset can generate two different models, one optimized for faster verification, and the other that minimizes memory use and increases the probability that the model fits in the available memory. The speed–optimized version models the component and safety rule transitions as separate cycles, each resulting in a new system state. The number of reachable states using this model is larger than for the actual application, but the state computation is simple. The result is a memory–intensive but comparatively fast analysis, requiring the rules and fairness descriptions to account for the transitional states by including the phase information. The memory–use optimized version generates one state to represent every reachable system state, but the computation of each state is a complex process having to account for the different phases for execution of component and safety rule transitions. This model has the advantage that CTL formulae for the desired properties only refer to the system behavior, but the computation of next state is more complex thus requiring more time for the overall analysis.

3.5.1 Implicit Rule Generation

One of the main obstacles to the wider use of formal methods in industrial applications is the expertise required to use them in practice. Our toolset also provides rule generation support for some often required types of properties. These include local deadlock freedom, reachability, immediate state succession, and liveness. These rules are generated for all components and all states, and the user can pick those necessary and include them in the verification.

Deadlocks can involve components that are not referenced by a single rule, so the full system may have to be checked. The complexity of industrial scale systems is probably beyond the capabilities of SMV, so other approaches to deadlock detection are necessary. As in the case of safety violation detection, a static method can be used to verify the existence of deadlocks.

Static deadlock search method is based on a search for cycles in the delay–dependency graph. This graph can be constructed from the component and safety rule definitions, without combining their behaviors, and is therefore of polynomial complexity. The drawback of this method is that it can report unreachable deadlocks that prevent the user from using a deadlock–free system until a more detailed analysis proves its correctness.

Together with deadlock verification, model checking tools can verify that the synchronized system satisfies some reachability, liveness or real–time specifications. These are not receptive safety properties, and cannot be enforced by our synchronization method. These properties are very important for the correctness of a system, so even if enforcing them is not an option, their preservation should be formally verified. Automatically computed synchronization conditions that guarantee safety are also minimal in the sense that no acceptable states are made unreachable. This guarantees the preservation of all reachability and liveness properties, as long as they are consistent with the safety properties of the system.[1] Model checking can be done with the original components, and all the properties that can be satisfied without violating safety will be preserved in the synchronized system.

3.6 Code Generator

Code Generator produces executable versions of all synchronized components, as well as interfaces to the runtime support environment and links to the data processing code. Code is produced in C, since this is the most commonly used language in the area of embedded control systems which are the most likely targets for the application of GenEx synchronization. Main goals of the code generation are the speed and the flexibility of the code, as well as a compact and intuitive structure that can be mapped to the original specification and facilitates application testing and debugging.

Generated applications have a simple structure. Every component and safety rule is implemented as an independent set of procedures that represent the behavior of the corresponding finite state machine. In Figures 3.6- 3.8 we show three procedures representing the main parts of the implementation of a dining philosopher. The main procedure for an automatically synchronized philosopher is in Figure 3.6. This procedure is called once in each system execution to select an enabled transition for the component and execute it. This procedure merely selects a state–specific procedure for the current component state , and then updates the *currentstate* variable

[1] A reachability property is inconsistent with the safety when the only way to satisfy the reachability requires safety violations.

```
SMG_philo1()
{
  newstate= -1;
  trans=0;
  if(currentstate[_MD_philo1]==_ST_philo1__in__thinking)
       {SMG_philo1__in__thinking();}
  if(currentstate[_MD_philo1]==_ST_philo1__thinking__eating__delay__1)
       {SMG_philo1__thinking__eating__delay__1();}
  if(currentstate[_MD_philo1]==_ST_philo1__in__eating)
       {SMG_philo1__in__eating();}
  if(newstate != -1){currentstate[_MD_philo1]=newstate; change=1;}
}
```

Figure 3.6: Main procedure for a philosopher component

with the resulting state.

The state–specific procedure for the philosopher in state *THINKING* is partially shown in Figure 3.7. It evaluates the enabling conditions of the component transitions in order of their definition in the Prolog specification. The first transition whose enabling condition is satisfied will be selected for execution, and the remaining transitions will be ignored. Delayed transitions are first in the order of evaluation and, if any of them is enabled, the original transition to state *EATING* will not even be considered for execution. The original transition, shown in the bottom of the procedure, will be evaluated for execution when its completion would be safe, so it only requires an evaluation of its original enabling conditions. When a transition is selected, its destination state is selected as the new state for the component, and the action associated with the transition is executed.

The action procedure shown in Figure 3.8 corresponds to the transition from the state *THINKING* to state *EATING*. This procedure contains the effects of a specific transition on the system variables as well as links to the data processing code. Action procedures control two types of variables, the control variables defined in the component specification, and the variables carrying state data for synchronization with other components. The illustrated procedure modifies two variables used to represent the state of the first philosopher in the transition selection by other components.[2] Another purpose of the action procedures is to link the data processing code supplied by the user, and this action procedure allows the user to supply a procedure executed when the philosopher is allowed to eat, probably including the picking of the two forks. The user procedure must have a predefined name **philo1_eating_enter** and must be accompanied by the definition of a preprocessor variable **PHILO1_EATING_ENTER**.

Action procedures do not modify the value of the variables directly or immediately, they issue requests for modification that will be stored in a buffer until all components complete their transitions. Once all transition effects are accumulated, the new system state is computed by applying them all in parallel. The effects are applied in the order of component evaluations, but

[2]Notice how the second transition in the state–specific procedure consults the variable representing the second philosopher in state *EATING*.

```
SMG_philo1__in__thinking()
{
  if((sig1[_SG_hungry1]==1)&&(sig1[_SG_hungry2]==1)&&
       (sig1[_SG_excl12__safety_OK]==1)&&
       ((num==0)||((num== -1)&&(newstate== -1))||(num==1)))
   {newstate=_ST_philo1__thinking__eating__delay__1; SMG_action__24(); }
  else
  if((sig1[_SG_hungry1]==1)&&(sig1[_SG_hungry2]==1)&&
       (sig1[_SG_philo2__eating]==1)&&
       ((num==0)||((num== -1)&&(newstate== -1))||(num==1)))
   {newstate=_ST_philo1__thinking__eating__delay__1; SMG_action__24(); }

...

  else
  if((sig1[_SG_hungry1]==1)&&(sig1[_SG_philo2__eating]==1)&&
       ((num==0)||((num== -1)&&(newstate== -1))||(num==1)))
   {newstate=_ST_philo1__in__eating; SMG_action__25(); }
}
```

Figure 3.7: Procedure for philosopher in state *THINKING*

```
SMG_action__25()
{
 #ifdefn PHILO1_EATING_ENTER
  philo1_eating_enter();
 #endif
 set("philo1__eating");
 reset("philo1__thinking");
}
```

Figure 3.8: Action procedure for transition from *THINKING* to *EATING*

since the controlled variable sets of components are disjoint, no conflicts will arise.

Since code is generated for individual component states and not for the combined system states, the size of the generated code is smaller than the combined state space of the synchronized system. It is roughly proportional to the sum of the sizes of all synchronized components, while the system state space can be as large as the their product. This means that our method will not lead to code explosion where the system implementation is as complex as its state space.

Components can be grouped for execution in arbitrary ways, in a variety of execution environments. The runtime support kernel currently exists for both single process, multiple processes on a single machine, and heterogeneous distributed execution. Regardless of the execution environment, the generated code includes the specification for all components and safety rules in the system, thus making it possible to migrate or replicate components for reliability with minimal cost. These activities are not supported by automated tools, but the code structure makes them trivial for components without data processing dependencies.

3.7 Runtime Support Kernel

The goal of this research is production of a practical method for synchronization and integration of concurrent software systems. The most important characteristic of the generated code is its accurate representation of the formal model used in the analysis. Another important parameter of practical use is the executable nature of the generated applications. For a generated system to be considered practically useful, it has to be simple to extend or integrate with externally generated code. The concurrent nature of the target applications may require execution on separate machines, so execution support is needed for heterogeneous distributed systems.

GenEx addresses these concerns by providing an open linking interface,and heterogeneous distributed execution support. The structure of the executable code directly models the structure of the specified system. Every component and receptive safety rule is represented by its finite state implementation, independent from the nature of the execution environment. The simplest execution environment is the integration into a single process, and we will introduce the structure and execution ordering for that example. We will also show that the executable code correctly implements the assumptions about the execution environment made in our formal model.

The generated code is structured as a set of concurrent finite state machines, controlled by the execution support routines that perform the exchange of state data between components and safety rules regardless of their geographic placement. The environment interface routines sample the monitored variables once in every cycle, and update the system state information with their values. Each finite state machine determines its enabled transition, based on the previous system state and the resulting values of monitored variables, and executes it. The effects of the transitions are accumulated until all transitions are done, when they can be broadcast throughout the system as one atomic state change. While the execution of all component transitions does not necessarily occur simultaneously their effect is equal to a parallel execution. The equivalence of the execution environment with the synchronous transition model means that the synchronization mechanism produced for this semantic model satisfies the safety properties in the implementation.

The receptive safety rules are implemented as finite state machines, with a similar semantics to that of the components. Their transitions are executed in a sequence, but based on the identical state data, and therefore equivalent to a simultaneous execution. The safety rules act as observers, using the results of components' transitions to determine their enabled transitions. The transitions for the safety rules occur in a second phase, after the components make their transitions and their effects are incorporated in the system state. The results of the safety rule observer transitions are accumulated during their execution in every phase and, when all safety rule observers complete a transition, their effects are propagated to the system state. The results of safety rule observer transitions are a part of the system state and are available to the components for use in determining the enabling conditions in the following execution cycle.

The overall execution structure is given below:

```
Initialize the component and safety rule observer states.
Initialize signals
loop
   input monitored signals
   execute component transitions
   propagate effects to state data
   output controlled signals
   execute safety rule transitions
   propagate effects to state data
end loop
```

This sequence is implemented by the execution support routines that call the components and the safety rule observers to make their transitions. The execution support routines also take care of propagating the transition effects to make them accessible to all components, and invoke the user supplied input and output operations. This execution structure satisfies our assumptions about the relationship between the components, signals and safety rules. It is also an extendible execution structure, allowing the embedding of data processing segments in the component code without violating the synchronous execution assumption.

3.7.1 Distributed Execution Support

In the case of distributed execution, we use Polylith [Pur94] to provide a heterogeneous communication mechanism that supports the synchronization and state data broadcasting throughout the system. Polylith is a software bus implementation, providing platform independent high level communication support. The structure of distributed applications is a star of synchronized processes, with the control process in the center. The execution sequence is the same as in the single process case, with an additional propagation phase occurring between the input and the component transitions. This phase can be skipped if there are no properties of the limited resource type that require the components to know other components' states and monitored signals. The distribution of components and safety rules between the distributed processes is left to the user, with a set of common–sense guidelines.

- **Component locality.** Every component should be in a process located on a host where its inputs and/or outputs are.
- **Grouping by shared data.** Component clusters that share large sets of controlled or monitored signals should be grouped. This also applies to components with strong interaction, specially limited resource access rules that require more information for synchronization.
- **Safety rule observer locality.** Safety rule observers should be on a leaf node if all components they constrain are on that leaf node. Other safety rule observers should be located at the center node to minimize the communication time.

The most important parameter for distributed execution is the communication overhead, and it depends on the geographical distribution of the network and on the amount of transmitted data.

Our distributed runtime support kernel transmits only the signal change vectors instead of the entire system state information. Since only a part of the signals, hopefully a small one, is changed in every cycle, the change vectors will be short and take a short time to transmit. The locality of components and safety rules plays an important role in minimizing the communication cost, because the systems can be customized to transmit only the parts of their change vectors needed for synchronization.

3.8 Accessory tools

In addition to the main analysis, generation and runtime support tools, GenEx also includes a number of script generation and syntax verification tools. The purpose of these accessory tools is to simplify the synchronization and integration process, and to provide a filtering mechanism that quickly detects simple inconsistencies within the specification.

3.8.1 Script Generator

The script generator takes the specification of the relationship between the receptive safety rules and the components, and produces a script that includes all operations needed to verify the consistency of the system and produce an executable system that satisfies the specifications. The script consists of a list of commands whose invocation results in the generation of an executable system implementation. The script contains invocations for both static and reachability analysis of the safety, and the user chooses the method to apply for any given system. Some of the commands in the script are independent and can be executed in parallel, reducing the time needed to complete the integration.

The script generator produces a list of commands and files with Prolog predicates. The commands are in the form of Prolog invocations using the files as command sources. Each Prolog predicate file directs the interpreter to load the necessary predicate data and function specifications, process the data and output the result into a new predicate data file. The predicate data files serve as the communication mechanisms between distinct integration phases. The script for the synchronization of dining philosophers is given in Figure 3.9, and it contains the following commands.

- The first script command verifies the correctness of the relationship between the components and safety rules, and introduces the prototypes of delayed transitions into the components where they may be necessary. It also combines all signals used by the components and safety rules into a shared signal array that will be used for system state sharing, and generates the prototypes of delayed transitions for all transitions that may cause safety violations.

- The second script command includes the verification of correctness of component descriptions, where the transitions are verified for the proper condition list length, and similarly for the appropriate effect list length. The safety rules are verified for the proper enabling conditions. A safety rule transition must use valid component states for enabled transitions.

```
prolog <scripts/expand_components.txt
prolog <scripts/global_data.txt
prolog <scripts/rule_philo_excl12.txt
prolog <scripts/rule_philo_excl23.txt
prolog <scripts/rule_philo_excl34.txt
prolog <scripts/rule_philo_excl41.txt
prolog <scripts/do_static_sync.txt
prolog <scripts/priority_philo_excl12.txt
prolog <scripts/priority_philo_excl23.txt
prolog <scripts/priority_philo_excl34.txt
prolog <scripts/priority_philo_excl41.txt
cat philo_exp.spec >>philo_all_joint_uniq.spec
prolog <scripts/output_smv_model.txt
prolog <scripts/output_exec_model.txt
```

Figure 3.9: Script file for the synchronization of four dining philosophers

Another type of verification is the receptiveness of the safety rule, where nonreceptive safety rules are detected by their constraints on environment events or their time dependent nature.

- The script also includes two sets of commands for computing the synchronization conditions. The reachability analysis of component subsets is specified for each safety rule separately, and each analysis may be executed in parallel on a different machine. The static analysis, due to its lower complexity is combined into a single command for all safety rules.
- Another set of commands adjusts the delayed transitions to include the priority signals, separately for each safety rule. This set of commands may also be executed in parallel.
- The original transitions are combined with the delayed transitions, thus producing a system with synchronized components.
- A command for generating executable implementation of the system produces a system specification in a SCR like form accepted by our code generator.
- A model generation command produces SMV models of the controller or subsets of its components and safety rules. These models are useful in formally verifying that the generated system satisfies deadlock freedom or other liveness properties.

3.8.2 Specification Analysis Tools

The specifications of components and receptive safety rules have to conform to a number of consistency and correctness rules in order to produce an executable and reliable system. The following rules are supported by automated verification tools.

- **Correct signal lists.** In the tabular specification language, the length of the enabling condition list has to be correct for the transition to be included in the component. If a transition is found with incorrect length of the condition list, an error report is generated to the user. This analysis is completed before the synchronization process starts because it usually implies that a component is functionally incorrect and the synchronized system will be incorrect too.

- **Condition Overlapping and Completeness.** Overlapping condition sets imply that the component or safety rules are defined as nondeterministic finite state machines. While the determinism can be imposed at runtime by giving higher priority to the earlier transitions in the list, the reachability analysis is made more complex and more restrictive if different nondeterministic paths lead to different safety violations. The completeness of the enabling conditions in a given state is not a problem for the execution since a default self loop transition is always enabled and preserves the safety. However, incomplete set of transitions may be an indicator of incorrectly specified components and safety rules. Both overlapping and completeness can be verified by our tools, and if either is violated the user gets a warning documenting the violation.

- **Consistency between safety rules and components.** The safety rules are defined as finite state machines whose transitions are enabled by combinations of component states. If a safety rule transition requires a state that does not exist in the component, that transition is never going to get executed. Detection of undefined states in safety rule transitions is an error and is reported to the user.

- **Unique signal controller.** Every signal has at most one component controlling its value. If multiple components control a single signal, the value of the signal may be inconsistent with what the components expect due to their interleaved execution. The controlled signal lists are computed as part of the integration and synchronization process, for code optimization purposes. If some lists are found to intersect, that implies the respective components share control of a single signal, and the user is informed by a warning.

- **Safety rule receptiveness.** The receptiveness of safety rules is a precondition for successful enforcement by component synchronization. The basic condition of receptiveness is that the safety rule can not be violated by monitored signals. While all signals can be used by the safety rules to enable their transitions, the transitions to **reject** state must be enabled only by combinations of component states. Rejecting transitions that use signals in the enabling conditions are reported as errors.

- **Time dependent safety rule detection.** The synchronization process produces delayed transitions for components when their transitions lead to states that may activate a rejecting transition by some safety rule. This synchronization process effectively uses a one step lookahead to detect transitions that immediately cause safety violations. A possible problem with this enforcement strategy is the possibility that the safety rule observer will arrive at a state with the components already enabling its rejecting transition. Delaying components in this case will not help the enforcement of the safety because the only way to prevent a violation is to force a component out of the state that causes the violation. The violation can be prevented by a timely action that cannot be enforced by delays, so this property is potentially time dependent. Time dependent safety rules are detected by finding transitions that can be enabled by conditions that enable violating transitions in the next safety rule state. Detected time dependent rules are reported as warnings, since the time dependent safety violations may not be reachable in actual executions. If they are reachable, they will make the safety rule monitors reach their **reject** states and will be reported at runtime. j

3.8.3 Visualization and Debugging Tool

In addition to the generation of code for the application, the code generator also includes a placement routine that produces the drawing coordinates for the system visualization. The components are represented as sets of states connected by transitions on a two–dimensional grid. The states are grouped in a way to avoid intersection between transitions in distinct components. The generated visualization information includes state positions in the grid, state adjacency as represented by transitions, and state names that identify individual states. The set of system signals is also represented graphically as a list of boolean switches.

The generated code supports the integration with a visual animation tool that allows interactive testing and debugging. The tool represents the system at the level of formal components, safety rule observers and signals, and provides an intuitive and familiar model of execution. This visualization tool is based on the xtango [Sta90, Sta92] algorithm animation engine, and is embedded in the execution support kernel.

The animation tool represents the components and safety rule observers as graphs, showing their current and previously visited states. The signals are represented as switches and interactively modifiable, to allow the user to control the system execution. This tool can be used both as a prototype for visualizing and understanding the interactions between the components, and as an interactive debugger allowing low–level control over the execution. It can be linked to integrated applications that include embedded data processing, and can be used in parallel with a standard debugger to analyze the data computation aspect of the application.

Our visualization tool has some capabilities that are not present in standard debuggers, but are very useful in formal finite state-based systems. The finite state space makes it possible to reconstruct previous states by backtracking the execution, without loss of information or inconsistencies in the state data. We can single step the execution of the control aspect of an application both forward and backward, to memorized previous states.[3] Execution sequences can be memorized, and rerun under different conditions to analyze the system reactions to different environment behaviors.

The capability to reconstruct previous states provides a powerful technique for reconstructing the causes of some undesirable system behavior. The integrated system enforces the given receptive safety properties, but it may violate some real–time and liveness properties in the process of synchronization. Even receptive safety properties may be violated if the safety rules used to model those properties are incorrect or inconsistent with the components. Those failures can be detected during debugging, and the executions can be traced backwards, reconstructing previous states until the root causes of a violation are detected.

[3]The reconstruction of the component states and signal values is reliable, because it is based on memorized traces. If an application is already linked with data processing code, it may contain some implicit internal state data that will not be reconstructed. For those applications, the reconstructed control states are correct and can be used for tracing backwards, but going forward again may lead to state corruption due to the possible influence of data processing code on the interaction.

3.9 Summary

GenEx toolset consists of a number of analysis, generation and visualization tools that support the design of synchronized concurrent systems that enforce the requirements given as receptive safety properties. GenEx tools simplify the specification of system components by reducing the need for explicit synchronization. The automated synchronization of components liberates the designer from a conceptually simple task with a potential for high combinatorial complexity. The code generation and runtime support library make the resulting application easy to replicate or relocate to a different execution environment. Finally, the visualization and debugging tool provides a very powerful intuitive interface to verify that the behavior of the synchronized system is what the user intended it to be.

Chapter 4

Dining Philosophers

The dining philosophers is one of the classic synchronization problems. We have shown in the previous chapters how the philosopher behavior and interactions can be specified and automatically synchronized to produce a functional and correct implementation of this system. In this chapter we will show some details of the system specification and of the computation of synchronization conditions. We will also explain how our method avoids deadlocks and starvation in the synchronization for this and any other exclusion based system.

Dijkstra introduced the dining philosophers problem [Dij] and analyzed its mutual exclusion requirements and possible implementations. A group of n philosophers is alternatively thinking and dining at a round table, but they are restricted by the lack of forks at the table. There are only n forks at the table, one between every two plates and every philosopher can eat only when both adjacent forks are available. The purpose of this system is to synchronize the philosophers so they can all eat, in some order since the adjacent ones cannot do it simultaneously. This system has a progress requirement specifying that any philosopher that becomes hungry will eventually be allowed to approach the table, take both adjacent forks and eat. This implies that the philosophers must avoid deadlocks, livelocks, and starvation of individual philosophers by their neighbors.

4.1 Classic Solutions

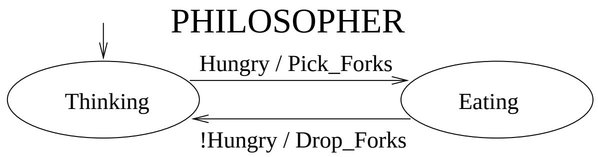
Dijkstra gives two algorithms that make the philosophers synchronize their accesses to the table, and satisfy the progress property. He also modeled the system as a polygon where nodes represent the philosopher and edges represent their mutual exclusion requirements; this is a simple and intuitive model of the system. One algorithm uses global mutual exclusion to isolate the critical actions where the philosophers access shared data structures that hold the ordering information in the form of a matrix of dependency edges. The second algorithm uses distinct semaphores to enforce mutual exclusion between adjacent philosophers. The semaphores can only be accessed in a predefined order to prevent deadlocks. These algorithms avoid starvation by enforcing either

FIFO access in order of requests, or 1–bounded overtaking for some pairs of philosophers. This property specifies that one philosopher can eat at most once while an adjacent philosopher is hungry. Both the 1–bounded overtaking property and FIFO access are proper subsets of the realizable part of the starvation freedom property, and thus limit the execution to unnecessarily restricted patterns. The limitation is evident in the fact that a philosopher that is still waiting for some resources can be the only one blocking another philosopher. If the philosophers take control of the resources atomically, the resources are only taken when they can be immediately used.

Some later approaches to this problem use the forks as the synchronization mechanism whereby one philosopher can take the fork, and the other adjacent philosopher is then blocked waiting for that fork to become available. The forks are essentially equivalent to the semaphores that enforce the mutual exclusion of pairs of adjacent philosophers in the second Dijkstra's algorithm. The explicit nature of the forks used as a synchronization mechanism leads to the analysis of their ordering, and the impact of that ordering on the system execution. Synchronization with sets of semaphores has one inherent problem in the occurrence of deadlocks whenever the access patterns to the semaphores create a closed circuit. If all philosophers reach for the fork on one side first, and then for the other one, they could potentially all pick the first fork simultaneously and get deadlocked. The deadlock occurs because no forks are available, and every philosopher is waiting for another fork, thus not releasing the fork they are holding. The existence of this deadlock state implies that the progress property is not satisfied by the system.

One type of solution to this is the introduction of a global limitation on the number of philosophers that may be trying to pick the forks and eat at the same time. It has been shown that by allowing at most $n-1$ of the philosophers to attempt to eat, the cycle in semaphore access can never be closed because at least one philosopher is always out of the critical section. This limitation on the number of philosophers in the *Dining Room* is usually implemented in the form of a counting semaphore with the capacity $n-1$, that the components(philosophers) access before trying to reach for the forks(local semaphores), and release after returning the forks . A problem with this solution is the potential for the serialization of accesses because when $n-1$ philosophers all have one fork, only one of them can pick another and eat. When the philosopher returns the forks and exits the dining room, only one other philosopher is allowed to pick its two forks and eat. The probability of the occurrence of serialized access grows with the higher access frequency, making this design a serious bottleneck. Reducing the number of philosophers allowed in the dining room does not solve this problem and might not even alleviate it. All the philosophers allowed in the dining room may still be serialized, and they block the philosophers whose forks are available from entering the dining room.

Another type of solution for this problem is based on the breaking of the symmetry between the philosophers, by making one or more philosophers reach for their forks in the opposite order. This way at most $n-1$ forks may be picked simultaneously, and one fork is always available or some philosopher is eating and will make both forks available when done. By reordering the semaphore access for one philosopher, the cycle in access to the semaphores is broken and no further deadlocks can occur. The shortcomings of this solution must do with the maintenance of different implementations for the same component behavior, and the serialization of accesses under heavier load. To address the serialization issue, the even/odd philosophers need to access their forks in opposite order. This produces a system with reasonable performance, but with



a)

```
mode([philo],[[hungry]]).
init([philo],[thinking],[[]]).
trans([philo],[thinking],[eating],[[t]]).
trans([philo],[thinking],[thinking],[[f]]).
trans([philo],[eating],[thinking],[[f]]).
trans([philo],[eating],[eating],[[t]]).
```

b)

Figure 4.1: Description of one philosopher

some problems related to its configuration and maintenance.

These approaches show the tradeoffs that sometimes must be made between the simplicity and elegance of a system design and the performance of its implementation. The cause of the problems in this case is the inappropriate level of atomicity for the synchronization mechanism. The semaphores can enforce mutual exclusion between two philosophers, but every philosopher has to access two semaphores to synchronize with both of its neighbors. Since two separate synchronization calls are needed, the process is not atomic, and the interleaving with other components becomes critical. To keep the interleaving controlled and acceptable, we sometimes need to introduce arbitrary asymmetrical restrictions on individual component behaviors.

4.2 Automated Synchronization for Dining Philosophers

We present a different approach to synchronizing the dining philosophers, one based on automated synchronization and with the appropriate atomicity for the problem. In our case the forks are no longer used as the synchronization mechanism, they are merely the limited resource that is the cause for the synchronization. The synchronization mechanism is produced by analyzing the component behavior with respect to the specified interaction properties. The synchronization for limited resource access properties was described in Section 2.8.1, using the mutual exclusion of two philosophers. We will quickly repeat the results of that analysis and explain how they apply to the integrated system with synchronization for multiple properties.

The specification of one philosopher component is given in Figure 4.1, using the same FSM notation as in earlier examples. The behavior of a philosopher consists of two states, $THINKING$ and $EATING$, and the transitions between them are caused by the *hungry* signal for the philosopher. The transition from $THINKING$ to $EATING$ is enabled when the philosopher is hungry, and it includes the actions needed to pick the forks; since forks are not used as a synchronization mechanism, they can be omitted from the system specification.

This specification represents the functional behavior of the philosopher, and it has to be modified to satisfy the mutual exclusion requirements of the system.

Since each philosopher needs to have both adjacent forks to eat, two adjacent philosophers can

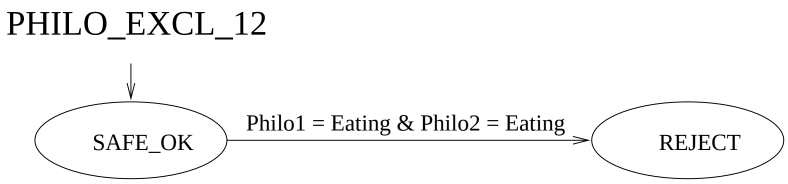


Figure 4.2: Philosopher mutual exclusion

$$
\begin{aligned}
Delay2 = {} & (philo1 = eating) \vee (philo3 = eating) \vee \\
& (philo1 = thinking \wedge hungry1 \wedge \neg ndet\_prio\_21) \vee \\
& (philo3 = thinking \wedge hungry3 \wedge ndet\_prio\_32) \\
Proceed2 = {} & \neg Delay2
\end{aligned}
$$

Figure 4.3: Compound enabling conditions for the second philosopher

not eat simultaneously. We use this restriction to create the receptive safety properties that the system must satisfy. The mutual exclusion property for two adjacent philosophers is given in Figure 4.2, and it allows at most one of them to be in the *EATING* state at any time. The mutual exclusion property is asserted for every pair of adjacent philosophers, and this set of properties makes the system's global interaction specification. We can use these components and receptive safety properties to specify the system exactly as described in Dijkstra's original model; we assert the desired mutual exclusion rules for every pair of adjacent philosophers as edges in the polygon model.

The safety violations can occur when one philosopher is *EATING* and its neighbor attempts to do the same, or when two adjacent philosophers get hungry at the same instant, and execute the transition to *EATING* state simultaneously. The following synchronization conditions are produced to delay the second philosopher and prevent it from violating its mutual exclusion property with the first one.

$$Delay(1) = (philo1 = eating) \wedge hungry1$$
$$Delay(2) = (philo1 = thinking) \wedge hungry1 \wedge \neg ndet\_prio\_21$$

Similar synchronization conditions are computed for the mutual exclusion of the second philosopher with its other neighbor, and for all other philosophers and their mutual exclusion constraints. When the philosopher is synchronized with both adjacent philosophers, its delayed transition will be enabled whenever one of the adjacent philosophers is either eating or has a higher priority and is ready to make a transition to the *EATING* state. The transition to the *EATING* state is enabled when all delay conditions associated with it are false. The compound enabling conditions for the second philosopher are given in Figure 4.3, and the delay condition shows how all reasons for a delayed transition can be aggregated into a single boolean expression for evaluation purposes. Computing the condition that enables the philosopher to proceed could be a complex task in the case of a component with many states and many interactions. Fortunately we do not need to produce the boolean expression in DNF form since, for execution purposes, we can enable the transition whenever the delay condition produces the value *false*.

4.2.1 Liveness of the Synchronized Philosophers

The synchronized philosophers satisfy the safety rules used to compute their synchronization conditions. The safety rules specify that the adjacent philosophers must eat at distinct times, and a hungry philosopher waits for its neighbors to finish eating. Beside the safety of the system, we want to verify its consistency with some liveness properties.

Liveness properties specify that a system eventually reaches some desired state starting from a state that satisfies its preconditions. The progress property requires that the any hungry philosopher will eventually be allowed to eat. Two conditions may occur that prevent the component from ever reaching a reachable goal state. The component can get involved in a deadlock where a number of components are blocked waiting on each other to release some resource. Another problem is the possibility of starvation where one component infinitely waits for a resource that will not become available.

Deadlock states occur as a result of the components requesting resources nonatomically, and in different orders. If component accesses to some resources form a cycle, then deadlocks are possible. In the case of dining philosophers, every philosopher atomically requests to be allowed to the table, at a time when none of the adjacent philosophers is. Since the resource (license to eat) is acquired atomically, no cyclic request dependencies exist, and deadlock is impossible.

The starvation problem may occur when a component competes for a resource with a group of components that do not require exclusive access between them. Some components in the group can keep the resources locked for the group while others release them, perform noncritical tasks, and return to the critical section. This way each component in the group sometimes exits the critical section, but they as a group can block the access to the critical resource forever. In the case of dining philosophers, starvation occurs when two philosophers deny access to the table to the philosopher between them by always overlapping their accesses to the *EATING* state.

In our implementation of the dining philosophers, this pattern is possible for finite executions, but the starvation can not be indefinite because the nondeterministic relative priority signals implement what amounts to extreme fairness. Extreme fairness is defined in [Fra86] [Mai93] and it applies to systems with probabilistic choice of possible executions. When a path to the desired state has some finite probability from an infinitely often occurring state, the desired state is eventually reached along all paths. Extereme fairness is also required for the exit of the philosophers from the critical section because strng fairness there is insufficient to guarantee the starvation freedom. Two philosophers that enter the critical section in overlapping intervals can starve a philosopher between them while their exits from the critical section clearly satisfy the strong fairness.

The fairness assumptions needed to prove the starvation freedom are that the *hungry* signals are independent from the states of other components in the system, and that when a philosopher is *EATING*, there is a minimal constant probability p that the *hungry* signal becomes false during any cycle. This assumption is more restrictive than the weak and strong fairness assumptions about the components leaving the critical section, but this restriction is required to satisfy the starvation freedom when philosophers do not reserve one fork waiting for the other. The assumption about the nondeterministic relative priorities is that they have a 0.5 probability of giving priority to any of the referenced components.

Theorem 4.1 $(Hungry1 = true)$ *leads–to* $(Philo1 = Eating)$

Proof: *Assume that the inverse of the statement holds, namely that there is an infinite path where (Hungry1 = true) holds, but (Philo1 = EATING) never occurs. If the philosopher is in the state* THINKING *in the first state on this path, its transition to* THINKING_DELAY *is enabled and immediately executed. Thus the philosopher has to be in the state* THINKING_DELAY *with (Hungry1 = true) for an infinite period.*

The only thing preventing the philosopher from advancing to the EATING *state is the fact that at least one of its neighbors is eating. The probability that the eating neighbor stops being hungry in the next state is p, and the probability that the other neighbor is allowed to enter its eating state before* **Philo1** *is* 0.5*. There is a* $p/2$ *probability that* **Philo1** *can advance to its* EATING *state without violating the exclusion properties. On an infinite execution, the probability of this transition being taken at least once is 1. Therefore a state with philosopher in* EATING *state is reachable, contradicting the assumed existence of an infinite starvation path. This proves that our design for the dining philosophers system is starvation free.*

4.2.2 Explicit Starvation Freedom Enforcement

The guarantee of starvation freedom based on the nondeterministic priorities is a theoretical construction, but for practical purposes we may need a more specific restriction on the period a philosopher will wait before it is allowed to eat. That is exactly what the original Dijkstra's algorithms do by enforcing the receptive properties of FIFO and partial 1–bounded overtaking for the philosophers. These properties are not documented anywhere as a part of the system specification, thus making it impossible to reason about the system behavior and performance without analyzing the implementation code. GenEx allows the explicit specification of these properties, and automatically implements them by synchronizing the components.

The starvation freedom property is implied by the receptive safety properties specifying FIFO access and 1–bounded overtaking. We can assert the 1–bounded overtaking property as a constraint on the interaction between any pair of adjacent philosophers. This makes the starvation freedom explicit and limits the wait for those philosophers. The waiting period is obviously not defined as a function of specific timing intervals because that would make it a non–receptive property. To make the rules receptive, the interval that one philosopher can spend waiting should be specified in terms of controlled events occurring in the system. In the case of 1–bounded overtaking, the events of interest are the access to the state *EATING* by the adjacent philosopher.

The 1–bounded overtaking property is given in Figure 4.4. This safety rule is activated when the second philosopher is hungry, and the first one is eating. After the first philosopher exits the *EATING* state, the safety rule goes to the state *DONE_ACCESS* where it rejects repeated access to *EATING* by the first philosopher. The safety rule leaves the state *DONE_ACCESS* once the second philosopher reaches the *EATING* state and the starvation is avoided. Similar rules can be imposed for other pairs of philosophers to specify the interleaving of their eating.

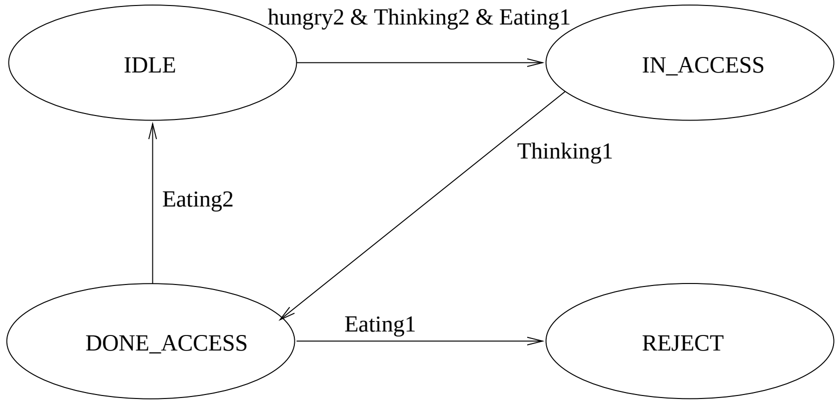


Figure 4.4: The 1–bounded overtaking property for a pair of philosophers

4.3 Complexity Growth for Dining Philosophers

The complexity of the state space for the dining philosophers is an exponential function of the number of philosophers, with up to $3^{N_{philo}}$ states [YY91]. Every philosopher has 3 possible states, where the second state is reached when the philosopher takes one fork. In our implementation, every philosopher has only two states related to their functional behavior, and the delayed state that is added to the components is reachable when a component is waiting to access the critical section. The analysis of this state space is clearly not a viable way to find the safety violations and compute the synchronization conditions that prevent them.

The analysis of the dining philosophers implementation does not require full state space traversals as was shown by Young and Yeh [YY91, Yeh93]. Their method extracted the regularity of the philosopher definition to reduce the complexity of the part of the state space that it had to analyze, and limited its growth to a linear function of the number of philosophers. Constrained expressions [ABC+91] provide another way of analyzing the behavior of the philosophers where the time requirements are a linear function of the number of philosophers.

Receptive safety rules in the dining philosophers example are defined for pairs of components, so only those components can violate the rules and need to be analyzed and modified to make the system satisfy the properties. For every safety property, there are two components whose behavior must be analyzed in order to enforce the safety, and all other components can be ignored. The reachability graph for mutual exclusion enforcement for any pair of philosophers consists of at most 9 states since each philosopher has 3. Every accepting state in the graph has an outgoing violating transition, so every state must also be analyzed using the static analysis algorithm. The analysis must be performed on all pairs of adjacent philosophers, so the total complexity of the analysis needed to enforce the mutual exclusion is:

$$N\_States = 9 * N\_philo$$

The same complexity of the reachability and static analysis is a result of the simplicity of the components and safety rules that makes every possible violation reachable, and every reachable state a potential source of safety violations. For more complex properties, the difference between the reachability and static analysis becomes much clearer. The state space for the reachability analysis of 1–bounded overtaking could have up to 36 states since the safety property has four states including the *REJECT*. There are less than 15 actual reachable states, and only two of them are predecessors of safety violations. The safety violations occur when the first philosopher enters the *EATING* state while the safety property is in the *DONE_ACCESS* state and the second philosopher is in state *THINKING_DELAY*. Using the static analysis, only the sources of possible safety violations are examined, and in that case we can see that the current state of the second philosopher is irrelevant for the safety violation, and that only the *EATING* by the first philosopher with the safety rule in state *DONE_ACCESS* can cause a violation. The complexity of the static analysis in this case is about 5 times lower then the reachability analysis.

This reduction in complexity is the result of the independence between all mutual exclusion properties. The symmetric nature of the problem was not taken advantage of and every pair of components is synchronized by performing the analysis of their behavior. Since all components and their interactions are identical, the synchronization conditions for every component are symmetrical, each defined based on the states of the adjacent philosophers. The analysis of a single pair of philosophers produces all the information that is necessary to compute the synchronization conditions for this system. The synchronization conditions can be replicated for all other pairs of components, using replication and string substitution to generate the synchronization conditions for all philosophers. The complexity of the analysis for this system is a constant value since only one pair has to be analyzed. This approach however still requires linear time complexity because the new delayed transitions must be generated for every component.

The dining philosophers problem can be generalized to include mutual exclusion constraints on an arbitrary adjacency graph. The process algebra and constrained expressions approaches that use the regularity of the dining philosophers to reduce the complexity could fail because of the irregular structure. The GenEx approach would still work linearly, whether by analyzing all pairs that require mutual exclusion, or by analyzing one pair in constant time and then replicating the synchronization conditions for all philosophers.

4.4 Summary

We have shown how the dining philosophers system can be implemented using the GenEx toolset. Our implementation is very natural and simple to construct from the specification of the required mutual exclusions. The automatically generated synchronization is atomic and therefore the system is deadlock free. The nondeterministic priorities enforce extreme fairness for the component actions, and the implemented system is starvation free. We have also shown how additional receptive safety rules can restrict the interaction to enforce stronger progress properties.

Chapter 5

Production Cell Controller

In this chapter we will give an example of the use of our method in an industrial application. This example will show how the complex behavior of the system can be decomposed into individual component behaviors and interaction between pairs of components. System requirements include nonreceptive safety, bounded reaction time and fault tolerance. We give a receptive representation of these system requirements and show how they are enforced by automated synchronization. The automated synchronization process analyzes the system behavior and modifies the components to make them enforce the specified receptive properties. The analysis is static and limited to subsets of components, and the generated implementation enforces all properties by allowing each component to locally determine its actions based on the system state information.

The Production Cell system was the object of a large case study involving over a dozen leading formal design and verification methods [LL95]. The goal of the case study was to produce a controller for the production cell and to verify its correctness. Most of the implementations limited themselves to the implementation without verifying its correctness. The main obstacle to the verification of this controller is that the state space for the system was estimated at fifty million states. This implies that the verification process entails excessive computational complexity. We will show how, using our method, the production cell controller can be automatically generated by the integration of individual device controllers. The components of the controller are modified by adding the automated synchronization that enforces the receptive properties of the system. The synchronization conditions are computed to satisfy some of the receptive safety requirements, thus making their verification unnecessary. More importantly, the analysis needed to automatically synchronize the components of the production cell controller is dramatically lower than the analysis that would be required for the verification of the same properties.

A second Production Cell system was proposed as a subject for further study, with somewhat modified components and more complex requirements including fault tolerance and runtime reconfiguration. We use the original production cell system to introduce the problem and show one design of the controller. We use the second production cell system to show how the system can be evolved to satisfy nontrivial changes in specifications and requirements. The controller that we designed for the second production cell required minimal modifications of the system

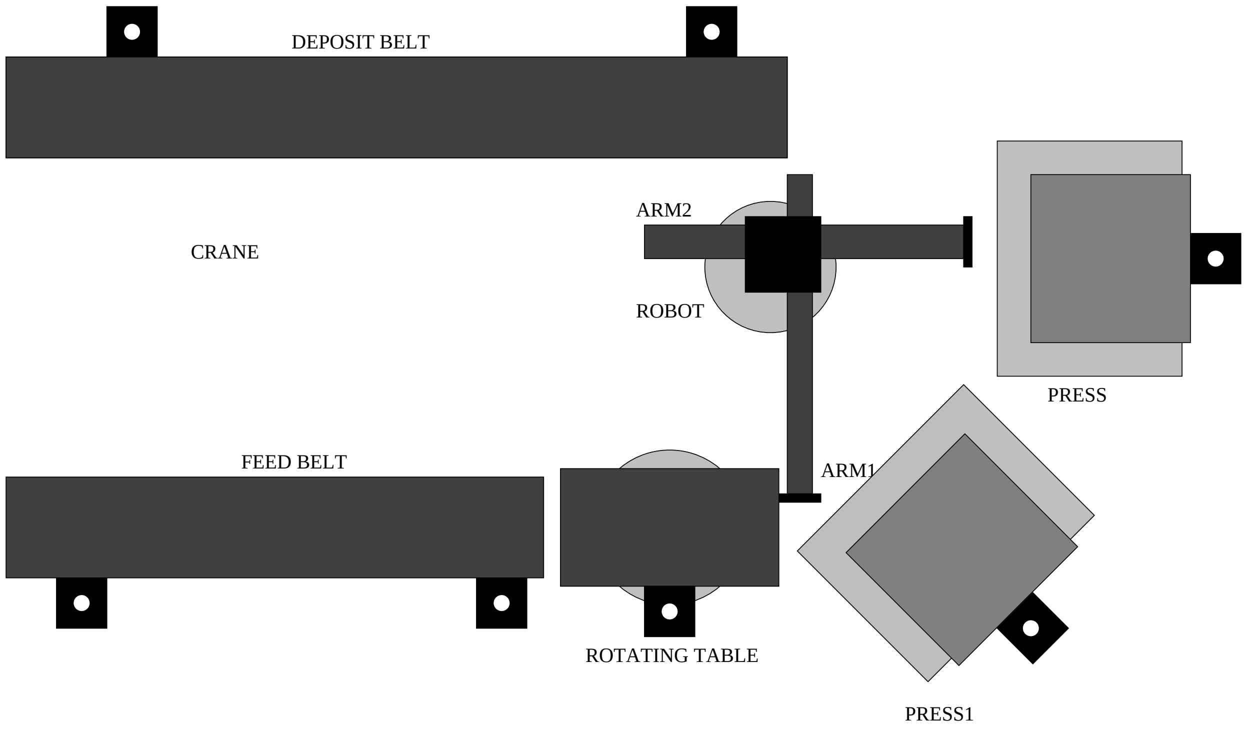


Figure 5.1: The Production Cell system

components, allowing a high level of reuse.

5.1 Production Cell System

The production cell system in Figure 5.1 is a medium complexity industrial system with safety, reliability, real–time and liveness requirements. The goal of this system is to transport metal blanks to a press to be processed, and then transport them out of the processing system. For simulation purposes, the system is implemented as a closed loop where processed metal blanks are returned to the entry point. The system contains six devices that respond to control directives from a controller and return sensor data about their position and about detection of metal blanks on the conveyor belts. The devices are: 1) a feed belt that brings the new metal blanks and deposits them on the rotating table, 2) a rotating table that positions the blanks to be picked up by the robot, 3) a rotating robot with two extensible hands for loading and unloading the press, 4) a press that processes the metal blanks deposited by the robot, 5) a deposit belt where the robot places the processed metal blanks, and 6) a traveling crane that closes the loop by bringing the blanks from the deposit belt to the feed belt. A controller for this system has to direct the behavior of each device keeping their combined behaviors within the set of specified properties.

The system has two types of safety requirements: 1) mobility restrictions for individual components, and 2) collision and blank loss avoidance requirements that describe safe interactions between system devices. The system can be controlled by moving one device at a time to complete their part of the processing, so the system is realizable. Since the system is realizable, there exists a receptive representation of its behavior, and of every individual required property. When the properties are formulated in receptive form, each of them can be satisfied by controlling the components it references. The property receptiveness allows the partition of the analysis and integration process.

The goal of the controller is to synchronize the devices to accept the blanks, press them and deposit the pressed blanks on the deposit belt, while restricting their movement to safe ranges and avoiding collisions. A single component that controls this system would suffer of insufficient parallelism of execution and the resulting inefficiency, or its size and complexity would make it infeasible to design. A simpler and more reliable way to control this system is using simple controllers for each device, integrated for safe and reliable interaction. Every component operates its assigned device at peak performance, while respecting the constraints that guarantee the safe execution of the system. This architecture allows us to take advantage of the parallelism in the system without having to exhaustively analyze all system states.

Some devices have individual mobility restrictions that specify the extremes of their movement, and exceeding those limits damages the devices. They also have positioning requirements that specify their position where interaction with other devices will succeed. In some cases, like the rotating table, the extreme safe elevation corresponds to the position required for the robot to pick up the blank. Mobility restrictions and positioning requirements are of real–time nature, because the controller has to react and stop a moving device within a predefined time interval (immediately for the rotating table). The safety of the system is also violated by component collisions and by the inappropriate handling that results in metal blanks being dropped. Collisions occur when two machines work in the same area, while metal blanks can be dropped when the machines are not in compatible states for transferring them, e.g. when the feeding belt unloads a blank with the rotating table in a high or diagonal position. These safety violations are time–independent and depend only on the component interaction.

The time–dependent requirements for the components to stop their movement in desired positions must be implemented within the component specifications, because they are not of the receptive nature that can be enforced by our synchronization mechanism. Time–independent safety properties are specified in the receptive safety rule form, and asserted as constraints on the behavior of the components. We analyze the components and receptive rules using GenEx to produce a synchronized and executable controller application.

The system also has a liveness requirement that every blank that enters the system eventually has to be processed. This liveness requirement is the highest end–to–end functional requirement of this system; its satisfaction makes this system useful. The liveness property can not be formulated in the form of a receptive safety property, and logically can not be enforced using component delays. We will introduce one pattern for the specification of components and safety rules that create no deadlocks and consequently preserve the liveness of the system.

5.1.1 Components

The desired behavior of the individual machines is outlined in the task description of the production cell system. We designed our controller components to satisfy those behaviors by initiating and halting their movement in the correct sequence and in a timely fashion. The timeliness applies in particular to the enforcement of real–time requirements for stopping a machine when it reaches a desired or extreme position. These properties must be satisfied by the component design because they can not be enforced by imposing additional delays.

Each individual component is defined as a finite state machine(FSM). Every transition has an enabling condition that activates it, and the completion of the transition may result in changes to some system variables. Generally, the components use monitored environment variables for enabling conditions, and change the signals that control the movement of the devices in the system. Some of the controlled signals serve as system memory, to help relate later decisions to previous behaviors. Figure 5.2 shows the finite state machine specification of two components. We use a single component to control every device except the robot that is controlled by three interacting components, one for each arm and one for the base.

The states embody the decision making properties of the components, where the selection of a transition determines the future behavior of the component. Transitions represent the component actions, taking them from one state to another and changing the values of the controlled signals in the process. The components in our system are designed to perform two functions: they specify the finite state sequences of a device movements, and they implement the time-dependent aspect of control requirements, such as mobility restrictions and correct positioning.

Press sensors distinguish three of its positions as interesting for the system behavior. When the press reaches its high position, a loaded blank is successfully pressed, and the press opens so that the blank can be picked, and closes to its middle position to be loaded with a new blank. Metal blanks are loaded on the press by the first robot arm when the press is in its middle position, and the second robot arm picks processed blanks from the press in its low position. The press controller initializes the press by bringing it to the middle position, and then requires a cyclical sequence: close to high position, stop, open to low position, stop, close to middle position, stop. The press may be in any initial position, including the undetectable ones, so the controller has to close the press to detect its position and then guide it to the middle position. The behavior of the press controller is illustrated in Figure 5.2. The transitions that stop the press movement satisfy the time dependent requirements for correct press positioning and respect for maximal movement range.

The behavior of the robot, as specified in the production cell study is as follows: first arm picks a new blank from the rotating table, second arm picks a processed blank from the press, second arm drops the blank on the deposit belt and finally the first arm drops its blank on the press. This behavior can be executed while the robot rotates counterclockwise from its rightmost to its leftmost position. The robot has three actuators, one for the base rotation and two for arms extension, and most of the time only one of them can be active. Robot rotation while any of the arms is not completely retracted may lead to a collision between the robot and the press. These signals are closely related, according to the component definition guidelines given in section 2.9, so the robot could be controlled by one component. The signal dependency is restricted to mutual

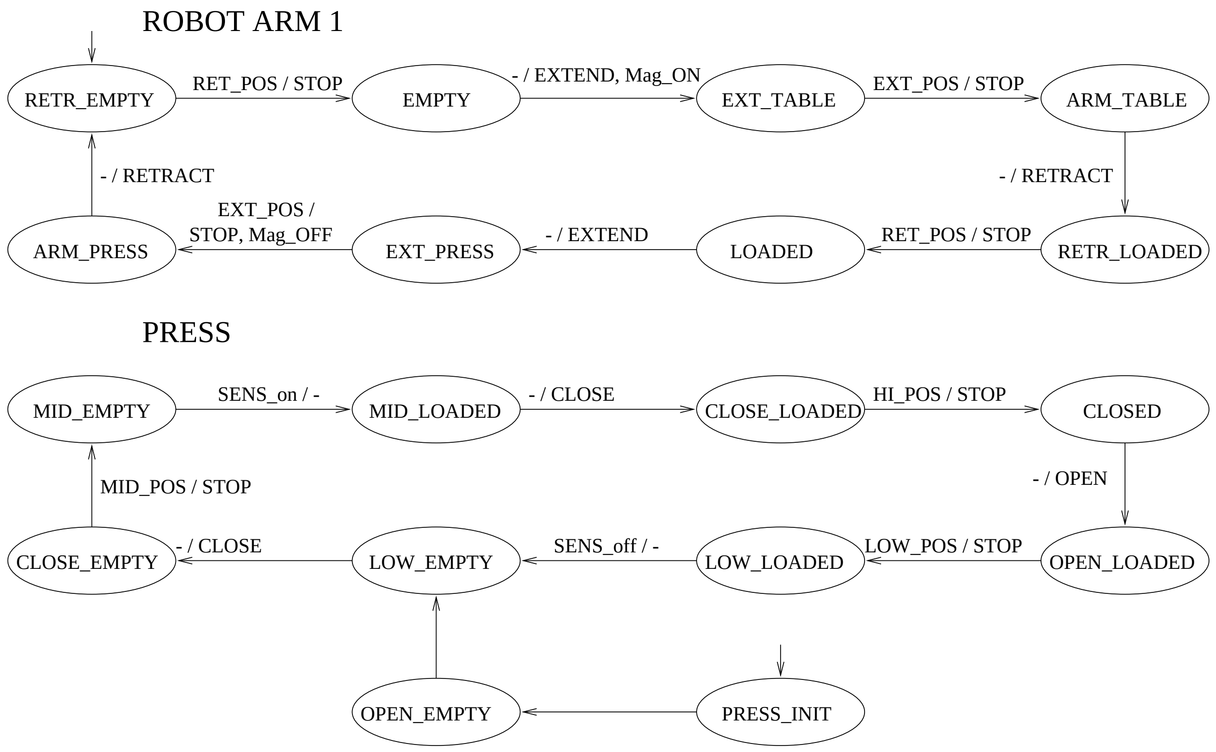


Figure 5.2: Components of the Production Cell system

exclusion, i.e. the rotation and arm extension signals are never active at the same time. Mutual exclusion can be enforced on separate components by the safety rules, so we can specify the robot using three simpler components instead of one complex one.

The arms alternatively extend and retract, turning the magnets on and off when they need to pick or drop a blank. The arm controllers control both the arm extension and the magnet activation because the magnets are trivial to include as part of arm behavior. The behavior of the arms is simple, extracted from the robot behavior specifications. Both arms initialize by retracting to allow the robot to rotate to the position where the first arm points to the rotating table. The first arm extends with magnet on to the table and then extends again to reach the press; upon reaching the press the magnet is deactivated thus placing the blank on the press, and the arm is retracted to allow rotation back toward the rotating table. The second arm first extends toward the press with activated magnet and then retracts first to the deposit belt where the magnet is deactivated, and then to its minimum extension where it allows robot rotation. The behavior of the robot arms is shown in Figure 5.2.

The robot base rotates, positioning the robot arms in a way to interact with the press, rotating table or deposit belt. The robot base operates in a predefined cycle: rotate clockwise until arm1 points to table, stop, rotate counterclockwise stopping when arm2 points to press, and when arm2 points to deposit belt and finally when arm1 points to press. This control sequence is not appropriate for the initial state when the press is empty, so the second arm has nothing to pick.

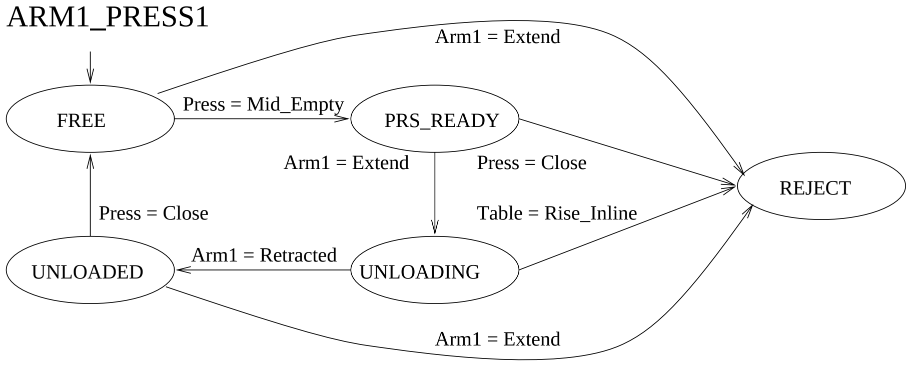


Figure 5.3: One safety rule for the Production Cell

The initial rotation of the base will skip the positions where the second robot arm can interact with other devices, so that arm will be idle until the next rotation when the press is already loaded. The robot base uses a controlled variable that initially indicates the press to be empty, and is modified to indicate a loaded press after the first rotation. When this controlled variable is false, the robot base only stops in the states where the first arm picks and drops a blank, and when it is true it stops in all selected positions.

5.1.2 Safety Rules

The component descriptions are completely independent, each component responding only to the device sensor inputs. However, it is clear from the description of the system that the components must be synchronized to pass the metal blanks. If the robot arm extends while the press is above its middle position, the two devices will collide, and if it drops a metal blank while the press is loaded, the two metal blanks will collide. Another requirement violation occurs if the robot drops the blank while the press is too low, because that makes the fall unsafe. These components must be synchronized to satisfy the requirements violated by the previous scenarios. These safety violations are unobservable by the controller components because the controller only receives information on specified device positions, while a collision may occur when both devices are in the indistinguishable intermediate positions. However, the safety violations in the physical system are caused by previous decisions by the controller and its components. We need to map the safety requirements to decisions of controller components in order to integrate them into a safe and reliable system. The press waits for the robot arm to deposit the metal blank and retract, than it closes to process the blank. The robot waits for the press to return to its middle position when the new blank can be deposited. This informal description of the desired interaction between the components is easily formalized into a receptive safety rule requiring that behavior.

The safety rule **ARM1_PRESS** in Figure 5.3 specifies the interaction between the press and the first robot arm. The first robot arm loads the press when it is empty and in its middle position. The robot arm may collide with the press if it starts to extend before the press stops in its middle position, or if the press starts to raise before the robot arm retracts after loading it. The safety rule specifies that the robot arm has to wait for the press to stop in its middle position before extending, and that the press stays in that position until the robot arm retracts after dropping the blank on the press. The rule prohibits the press from closing until the robot arm loads it and retracts, and it prohibits the arm form extending until the press is stopped in its middle position where it can receive a new blank. Since the rule restricts only component actions, it is a receptive safety property, and can be enforced by GenEx. Note that the rule imposes no constraints on the component actions that stop the device movement because these transitions enforce the time–dependent aspects of behavior. Also, every state of the safety rule constrains only one of the referenced components. The other component is allowed to proceed, and its actions will eventually lead the safety rule to a state where the constraint is removed and imposed on another component. This is the pattern of safety rule definition that we introduced in chapter 2.

This receptive safety rule enforces what is basically a handshaking algorithm for the **robot_arm1** and the **press** components. It requires a specific interleaving of transitions by the two components, insuring that the robot arm unloads the metal blank on the press. Although the components are defined independently, this rule references both of them, telling the GenEx synchronization tool to analyze them together and to modify their interaction by delaying one or the other and allowing them to complete the delayed transitions only when they preserve the safety. The independence between the components simplifies design and maintenance and promotes component reuse for similar systems.

The safety rule **ARM1_BASE** synchronizes the behavior of the first robot arm and the robot base. Possible safety problems caused by bad interactions between the robot base and arms occur when the arms extend while the base is rotating, so the arms can collide with the press. If the first robot arm extends to drop a blank at a time when it is not turned toward the press, the blank is dropped in an unsafe location causing a different safety violation. Another problem can occur that does not cause a physical collision, but makes the system state inconsistent and leads to other failures; if the first robot arm extends to pick a blank when it is not turned to the rotating table, the table then rotates back to the feed belt and accepts another blank thus causing a collision.

The basic safety that the rule requires is that the arm extends only when the base is stationary in states where the arm points to the table or a selected press. Once the robot arm starts to extend, the base remains stationary until the arm retracts. This rule entails two instances of handshaking in a sequence, one for picking the blank from the table and the other for dropping it on the press. A similar rule is necessary to synchronize the second robot arm with the robot base, and it has one difference related to the possibility that the arm might not need to extend and pick a blank if the press is empty. The base is restricted from rotating only if the press is loaded and the arm is ready to extend. The base may resume rotation if the arm is retracted and waiting for a loaded press to be selected.

The robot arms interact directly with the press, so their interaction has to be specified using a pair of safety rules. These safety rules specify the position of the press that allows the robot arms to extend to the press, and that the press has to remain in that state until the arms retract. These rules guarantee that the press is in the right position for the arms to drop or pick the blank, and that the press will not close and cause a collision before the arms retract. The rule

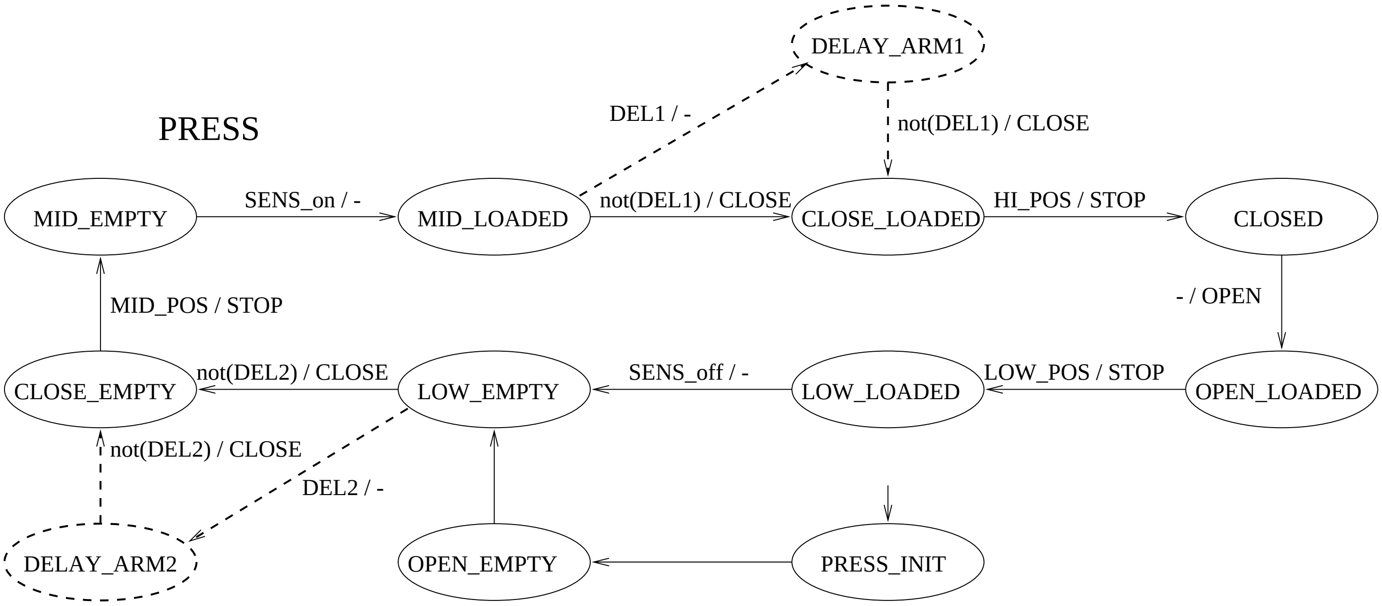


Figure 5.4: A modified version of the press component

ARM1_PRESS1 enforces the following sequence of events: press empty in mid position, arm1 extending to press, arm1 drops blank, arm1 retracted, press closes to process the blank. The synchronization of the second robot arm with the presses is specified by the rule **ARM2_BASE** and the sequence: press loaded in low position, arm2 extending to press, arm2 picks blank, arm2 retracted , press closes to middle position.

5.1.3 Synchronization of the Production Cell Controller

The components specify the functional behavior of the system and determine *what* actions individual devices will perform. The receptive safety rules specify the subsets of the set of possible system executions that are acceptable with respect to the component interaction. The behaviors that violate some of the safety rules are considered unacceptable, and should not occur in the executable application. The components specify *what* their next action is, but they are not required to be immediate. The system requirements are mapped to the component level, and they are enforced by delaying the components that have the potential to violate the receptive safety properties.

The process of making the components use other components' state data in determining their transitions is automated. The goal of the component modifications is to synchronize them, and make their behavior sensitive to the system state. The automated synchronization of local controllers requires safety analysis for all safety rules and the related components. All potential safety violations are detected, and their preconditions are used to compute the synchronization conditions for the components. The synchronization conditions enable the components transitions to a delayed state where they remain while they are a potential cause of safety violations.

The synchronized version of the **press** component is given in Figure 5.4. Two transitions of the original press had the potential to violate the receptive safety rules that specified the interaction between the press and the robot arms. These transitions are modified by the addition of delayed states, shown using dashed lines. The delayed states are reachable only when the completion of the original transition would lead to a safety violation, and as long as those conditions persist the **press** component stays in the delayed state. The transition from *LOW_EMPTY* to *CLOSE_EMPTY* is delayed as long as **ARM2_PRESS** is in state *A2P_PICKING*. The safety rule stays in the state *A2P_PICKING* until the robot arm returns to the retracted position where the press can no longer collide with it once it starts to close. A similar delay applies to the transition from state *MID_LOADED* to *CLOSE_LOADED* where the press has to wait for the first robot arm to retract before being allowed to close.

5.2 Two Press Production Cell System

We will now examine a variation of the production cell system, to show how the automated synchronization approach helps us reuse the functional description for the device controllers. The production cell is expanded to include another identical press, on the assumption that the press is the bottleneck device in the system. By using both presses in parallel, the performance of the system improves because the robot can pick a processed blank from one press and place a new one while the other press is processing its blank.

This production cell system has two identical presses, both reachable by the rotating robot, and identical in construction and controls to the press used in the first system. The original controller can be reused as a generic press controller and instantiated for the two presses using their specific control and sensor signals. The controllers for the two robot arms can be reused without any changes, since both presses are located at the same distance from the robot, thus requiring same arm extension. Since their interaction with both presses is equivalent, the robot arms have no need to distinguish between them.

The largest changes occur with the robot base whose functionality has to expand to handle two presses. The base has to recognize six different positions where it has to stop the rotation to allow the robot arms to interact with other devices. The robot base controller executes the press selection, and guides the robot to the appropriate positions where the arms can pick and drop the blanks on other machines. The robot base controller selects the presses in alternated order, so each press is used as much as possible. The press selection is based on a controlled variable that represents the previous selection, and its value remains constant until the other press is selected. The robot base controller uses two controlled signals, one for each press, to handle the initial interaction with the empty presses.

Depending on the press selection, the robot base goes through two different sequences of operations in every rotation. If the first press is selected, the robot base allows the second arm to pick a blank off the press, rotates to point the first arm to the table, stops, rotates to point the first arm to the first press, stops, and rotates to point the second arm to the deposit belt. If the second press is selected, the first arm accesses the table first, then the second arm accesses both

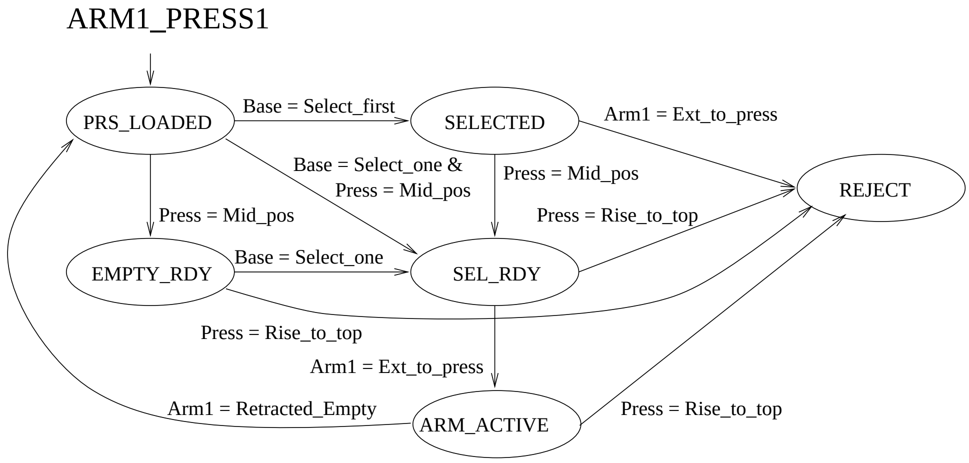


Figure 5.5: A safety property for the two press production cell system

the press and the deposit belt, and finally the first arm accesses the second press. In the initial rotations with empty presses, the operations of the second arm are skipped and the corresponding signals are set after the presses are loaded. After a rotation is completed, the robot base rotates clockwise to the position where second arm points to the first press where next rotation can start. The robot base is the most complex component in the system, because it embodies both the sequencing and the decision aspects of this production cell.

The safety rules for the presses and robot arms need to be modified, and they must reference the robot base for information on the press selection. The safety rule for the press that is not selected for loading should not interfere with the behavior of the robot arms. This effect was not apparent in the single press system because there were no two safety rules that would alternatively block a single transition. The safety rule **ARM1_PRESS1** is given in Figure 5.5, and it shows how the safety rule interleaving works. The rule accepts all behaviors until it gets enabled by the selection of the first press, and then starts to reject behaviors that lead to safety violations involving the first press. When the desired interaction is completed, the rule deactivates and again waits for the first press to be selected.

5.3 Fault Tolerant Production Cell Controller

The above described controller satisfies all safety, real–time and liveness requirements of the Production Cell. We will show how the controller can be expanded to ensure fault tolerance for the system. This example is a better approximation to the system requirements in the Production Cell2 case study, that includes the two press production cell and imposes additional fault tolerance requirements. The actual production cell 2 study assumes that all devices are failure prone, while we limit failures to the two presses. We consider press failures to be more “interesting” because the system can keep operating when one press fails. Failures of any other device require the system to stop execution until the device recovers and then restart. The most important aspect of the restart is the gathering of information about the state and position of components and blanks, a subject unrelated to our research.

The presses are the failure prone devices, and their design is enhanced with the addition of sensors that detect the presence of metal blanks on the press. The mechanical press failures affect their control mechanism making them nonresponsive to controller commands. Sensor failures make their output fixed at the value **false**, thus making it impossible to detect a blank on the press or to position the press in one of the predefined positions. To ensure fault tolerance, the controller has to detect press failures and reconfigure the system to use only the functional press until the failed one recovers. If both presses fail, the controller leads the system to a safe state, and waits there until the presses recover to a functional state. The controller raises an alarm whenever it detects a failure, and deactivates it when it gets a recovery signal from the production cell operator. The recovery signal is given when all devices are back in operative condition.

5.3.1 Failure Detection and Recovery

Failure detection is a problem outside of the scope of this research, and we will explain the basics of our approach, while concentrating on the interaction control. From the standpoint of the system controller, failures are not necessarily observable events, and in practice they are most often not directly observable. However, they must be indirectly observable[1] and the controller tries to detect them by comparing the expected correct behavior of the press with the actual behavior to detect previous failures.

Failure detection is a real–time requirement while non–interaction with a failed press is a matter of decision and sequencing. The failure detection, like other real–time requirements, is enforced by the controller components, as part of their functional behavior. The failure detection task is handled by the press monitors, two equivalent components each instantiated for one press. The press monitors track the actions of the press controller and the monitored variables representing the press position, and raise a fault alarm when the press shows unacceptable sensor inputs. The press monitor reinitializes together with the press controller when the user signals the system is fully recovered. The press controller and monitor interact using their controlled variables, the press activation signals and the press failure signal, and both have access to the monitored signals representing press position. Monitor also uses some robot control signals to distinguish picked blanks from sensor failures.

The press monitor reacts to the following inconsistencies in the press behavior:

- Failed position sensor.
  When the press is stopped in one of its three positions and the appropriate position sensor turns **false** without a movement command, the press monitor declares a press failure.
- Failure to reach a position.
  When the press is moving from one position to another, if it fails to reach its destination

[1]If a failure is totally unobservable, meaning that the behavior after the failure is equivalent to the behavior without the failure, then the failure is irrelevant for the controller and the system.

in a predefined amount of time the press monitor declares a press failure due to mechanical or position sensor problems.

- Failed blank detector.
  When the press is loaded with a blank, and the detection sensor has the value **false**, the sensor is assumed to have failed and a press failure is declared.

The specification of press behavior has to be changed to account for the possibility of failures and the subsequent recovery. A version of the press controller with fault detection and recovery is shown in Figure 5.6. Since a failure can be detected with the press in any state, every state has an additional transition to the *PRESS_FAILED* state, enabled by the failure signal from the press monitor. The press controller remains in the failed state until the press monitor deactivates the failure signal, after receiving the recovery signal. After the recovery, the press has to repeat the initialization protocol, to detect its current level and whether it is loaded or not. If the press is loaded after a recovery, it may be impossible to determine whether the blank has been processed or not. Due to the indirect and delayed detection of some failures, a blank may be processed by the press without the press detecting it arrived to the upper position. This nondeterminism is resolved by assuming that a blank found on the press at initialization or recovery should not be processed, since it may be incorrectly positioned or already pressed.

The initialization sequence for the press is used both at the start of execution and after a recovery, and it has to be able to restart the press from any position and regardless of the existence of a blank on the press. If the press is initialized without a blank, it moves to the middle position to be loaded, and if it is already loaded at initialization, it moves to the lower position to allow the robot to pick the blank.

Other components that must react to the press failures are the robot arms. They must be able to interrupt an initiated approach to a press, to avoid colliding with it or unloading a blank with the press out of proper position. The robot arm controllers react to press failures only when the arm is extending toward the failed press, making the arm return to its retracted position to try restart the approach once a press is available and functional. A robot arm controller that reacts to press failures is also given in Figure 5.6.

The faults can occur at any time, and with the system and its components in any state. In addition to the real–time reactions to failures, the overall behavior of the system has to change after a failure, to recover the system to a consistent state. The production cells without fault tolerance have a defined repetitive behavior after they complete the initial loading. At the end of every rotation both robot arms are empty, and both presses are loaded. When presses fail, some actions in a rotation are completed and some must be aborted or ignored so at the end of a rotation some press may be empty, and the first robot arm may still be carrying a blank that was to be loaded on the failed press. The behavior of the robot base needs some modifications to account for the possibility that the first robot arm is loaded in the starting position, and that the picking of a blank from the rotating table has to be skipped. Another possible effect of press failures is that the press is loaded after recovery. The behavior of the robot base already handles this situation in the initial state, but now it can also occur after a failure.

The safety rules that specify the interaction between the presses and the robot arms need an

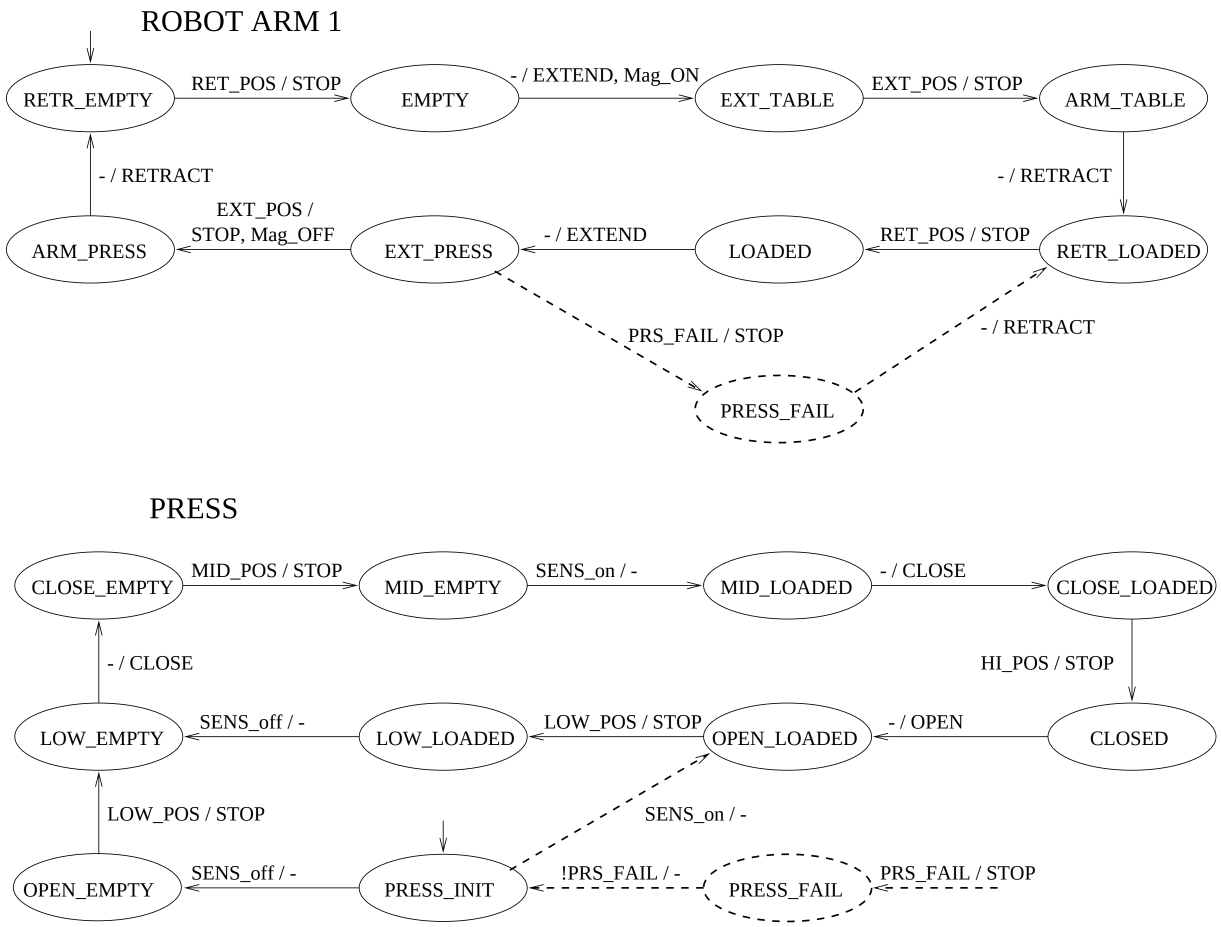


Figure 5.6: Fault reactive version of the press and the robot arm

additional failure state where they go when the respective press fails and stay until it recovers. While in their *PRESS_FAILED* state, these rules impose no restrictions on the behavior of the robot arms, and they react to the press recovery by going back to the initialization state, just like the press itself.

Since the behavior of the robot arms is modified, the safety rules for their interaction also must be slightly changed. The safety rules require the base to wait for the robot arms to complete their interaction with the presses, but in the case of press failure the arms abort the interaction, requiring to repeat it with an available press. The safety rule has to react to the aborted interaction, by returning to the initial state for that interaction. The rule **ARM1_BASE** also needs an additional state to return to the start of the rotation with the arm loaded. This way the arm is not required to pick another blank off the rotating table before depositing the blank it is currently holding on the press.

An additional safety rule specifies the requirements for a press recovery. If a failed press recovers too quickly during the same rotation when it failed, it may lead the first arm to drop a new blank on the loaded press, causing a collision. In order to simplify the system we specify a sequencing rule that requires the failed presses and their monitors to delay their recovery until the end of a rotation and the beginning of the next one. That way the system initiates a rotation with reliable state data and any subsequent failures remain in force during the entire robot rotation.

These minimal modifications are sufficient to handle all the possible executions that occur as a result of press failures. The press monitors encapsulate the complexity of failure detection, and force the presses to their failed states. The robot arms also have a small role in reacting to press failures during interactions. The safety rules specify the additional acceptable execution traces that capture the state space of fault consequences and recoveries.

5.4 Benefits of Using GenEx to Synchronize the Production Cell Controller

The use of GenEx allows us to specify the components of a production cell controller independently, and to integrate them into a functional system that satisfies the specified safety rules and preserves the time–critical behavior of the components. By using the receptive safety properties to describe the desired interactions between the components, we can specify the aggregate behavior of the system and have it automatically synchronized to satisfy the desired properties. The compositional nature of the generated applications allows us to create very complex systems using simple components, simple receptive safety properties. The enforcement of receptive safety properties is based on the analysis of violation preconditions, computationally far simpler than the reachability analysis. This means that all the design benefits come at a lower cost than doing formal and exhaustive system verification.

The local and functional emphasis in the component design provides a decoupling mechanism that allows us to create components independently, even if those components must interact. Every component defines its own behavior and determines its sequence of actions, while the system synchronization mechanism transparently delays the components that may violate the desired system properties.

The creation of complex systems is reduced to local decisions on the function and interaction between independent components. If the functional behavior of the components can be combined, their coupling is just a question of specifying the constraints for their interaction. Complex systems can be designed by combining the components whose functions they require, and adding the interaction properties in the form of receptive safety properties. Since the safety properties are specified for small subsets of components in direct interaction, even the safety properties tend to be simple and organized into patterns that facilitate their reuse in different systems.

The reusability is illustrated by showing the simplicity of modifying the system to use two presses instead of one. The behavior of most components remains identical as in the single press example, with the exception of the robot base that has to recognize previously undefined positions and enforce the alternation between the presses. The safety rules require some modifications if they deal with the modified components, otherwise they can be reused in their entirety from the single press system controller.

The use of automated synchronization also supports the modification of a system through the addition of new requirements, such as fault tolerant behavior. We've shown that the system can be upgraded to a fault tolerant version by modifying the components to include local reactions to detected failures, and by similarly making the safety rules deactivate on a detected press failure, and reactivate once the press recovers. The decoupling of the components makes it possible to seamlessly remove and add components to the executing system, depending on the need and availability.

5.5 Summary

We have shown how a medium complexity system can be synchronized to satisfy its requirements using our method. The complexity of the system was estimated to be around 50 million states [LL95], a hard problem to analyze even using the best of the existing automated verification tools. The complexity of the analysis required to synchronize the system for the specified properties was up to 500 states for the reachability analysis, and up to 20 states per property using static analysis. This reduced complexity is a result of decomposing the analysis by referenced components, and looking only for possible violations.

Chapter 6

Reliability of Automated Synchronization

In this Chapter we will show how the requirements of complex systems can be specified using simple receptive safety properties, and that enforcing each receptive safety property independently produces a system that satisfies all of its receptive safety requirements. We will also show that our synchronization mechanism can enforce any receptive safety property, and that the enforcement of one property will not contribute to violations of any other properties. We will also describe a set of patterns for system design that specify ways to produce deadlock–free synchronized systems. Finally, we will show how we can distinguish receptive safety properties from the nonreceptive safety properties which can not be enforced using our method, and how some nonreceptive properties can be implemented manually or modified for automatic enforcement.

6.1 Correctness and Decomposability of GenEx Synchronization

Our automated synchronization method must satisfy several conditions to become a useful tool in the development of complex concurrent systems. It must produce systems that satisfy their requirements and it must produce them quickly. The correctness with respect to the requirements is ensured by enforcing the receptive safety properties that represent the system requirements. Another correctness requirement is the guarantee that the synchronization mechanisms are composable, i.e. that their integration preserves the features of all individual safety properties. The complexity of the analysis is reduced by partitioning it for individual receptive safety properties, and using a static violation detection method. In this section we will show that the automatically computed synchronization mechanism satisfies the receptive safety property it was computed for. We will also show that these mechanisms can be combined to enforce multiple properties simultaneously.

6.1.1 Closure of Regular Languages Under Intersection

The closure of the set of regular languages under intersection shows one way of using simple languages to specify complex ones. Take the 1–bounded overtaking property for dining philosophers, shown in Figure 4.4. This property specifies that the first philosopher can eat at most once while the second philosopher is hungry. This property is specified by a finite state machine and therefore defines a regular language we will call L1. Assume an equivalent 1–bounded overtaking property is asserted to limit the number of accesses by the third philosopher while the second one is hungry, and we call the respective regular language L2.

The regular language L = L1 ∩ L2 is the intersection of the two languages describing the system behavior. This language describes the set of system behaviors where the second philosopher waits for its two neighbors to eat at most once while it is hungry. This limitation on the number of accesses by the adjacent philosophers guarantees that the second philosopher can not be starved. Since both L1 and L2 are regular languages, by closure of regular language intersection [JEH79], so is the language L. Knowing that L is a regular language guarantees that a finite state machine *M(L)* can be produced. M(L) accepts system behaviors if they belong to the language L1 ∩ L2, and rejects them otherwise.

If L defines a realizable property, there is a strategy *f(L)* that satisfies the property. We have shown how a strategy *f(L)* can be based on observing the state of M(L) and preventing the component actions that would lead to violations of the property. By enforcing the receptive safety property L, we would effectively enforce both L1 and L2.

The finite state representations of L1 and L2 are FSMs with three non–rejecting states and one violation transition, while the *M(L)* has 9 non–rejecting states and 7 distinct violating transitions. While the finite state representations of L1 and L2 are simple, the complexity of FSM representation of a language intersection can be an exponential function of the number of intersected languages. This is because the size of the FSM for L is proportional to the complexity of the desired behavior. However, we can show that there exists an equivalent structure to the M(L) whose implementation does not require the explicit enumeration of all state combinations.

Languages L1 and L2 are simple regular languages, corresponding to FSMs M(L1) and M(L2) that accept all behaviors acceptable by the individual languages and reject them otherwise. Parallel execution of M(L1) and M(L2) using the system events as input to both FSMs produces a system whose states are ordered pairs of states for the two FSMs. A behavior is accepted as long as both FSMs are in a non–rejecting state, and rejected when any of the individual FSMs rejects it. The state space of this system is equivalent to the state space of M(L). The complexity of the code for this implementation is equal to the sum of the sizes of the FSMs for the individual languages, drastically smaller than the size of the combined state space required by the FSM M(L).

6.1.2 Compositional Enforcement of System Requirements

Most complex systems have multiple behavior requirements, and all those requirements must be satisfied by the system execution. Every requirement is defined as a property, a set of acceptable execution traces. The overall requirements for the system are equal to the intersection of the properties that specify the individual requirements. The system is realizable if there is a strategy for the controller that generates only behaviors that belong to the requirement property intersection. We will show that the controller for a realizable system can be produced by satisfying the individual requirements.

Theorem 6.1 *A realizable controller can be implemented by enforcing the realizable parts of its individual requirements.*

Proof:

Given a set of required properties $R_1, R_2, ...R_k$, they define a compound property $R_0 = R_1 \cap R_2...R_k$. Since the system is realizable, all properties have nonempty realizable parts

$$(\forall i \in [0,k])(Rp_i = R_\mu(R_i \wedge Rp_i \neq \emptyset)$$

Assume there exists a $\mu-$strategy f *that satisfies $Rp_1, Rp_2, ...Rp_k$. Then, by definition of realizable parts and*

$$(\forall i \in [1,k])(O_\mu(f) \subseteq Rp_i)$$
$$O_\mu(f) \subseteq R_0$$
$$O_\mu(f) \subseteq Rp_0$$

Since the set of all fair outcomes of f *is a subset of the realizable part of the system requirements,* f *is a winning strategy.*

6.1.3 Enforcement of Receptive Safety Rules

Theorem 6.1 shows that a system with complex requirements can be produced by enforcing the individual requirements. We will now show that our analysis and synchronization method can enforce individual receptive safety properties by delaying the violating transitions.

Theorem 6.2 *Given a set of component specifications and receptive safety properties $Rp1, ...Rpk$, GenEx automated synchronization method enforces the properties $Rg1, ...Rgk$ where $\forall i : Rgi \subseteq Rpi$.*

Proof :

Analysis of the component behavior with respect to a receptive safety property detects all possible safety violations, and GenEx modifies the components by adding delayed transitions enabled by the preconditions of the safety violations. The delayed transitions prevent the components from reaching the states that cause the property violations.

Controller implements a strategy S *and we define property Rg to be the outcome set of S. This outcome set excludes all behaviors rejected by the property Rp, and therefore $Rg \subseteq Rp$.*

This theorem shows that automated synchronization using GenEx can produce controller implementations for realizable systems defined using receptive safety properties. The difference between the behavior of the generated implementation and the realizable part of the required properties represents the possibility that the synchronization produced by local analysis may not be globally optimal. This manifests itself in the form of delayed component transitions, when their completion would have preserved the safety. The nonoptimal synchronization conditions can be generated only for the limited resource access properties. The following example shows how these unnecessary delays occur, and how they could be avoided using global analysis.

Example:

Assume the first philosopher has some receptive safety property **SP1** imposing a constraint on its return to the *THINKING* state after *EATING*. This means that it will also have a delayed state *EATING_DELAY*, where it waits to safely return to *THINKING*. A delayed transition implies that the first philosopher may not leave the *EATING* state immediately after the variable *hungry1* becomes *false*. In the analysis of the philosopher mutual exclusion for the first and second philosopher, the condition $\neg hungry1$ no longer guarantees that in the next state the first philosopher will be in the state *THINKING*. The synchronization condition produced by analyzing the components is:

$$Delay2 = (philo1 = eating)$$

The second philosopher must wait for the first one to completely exit the critical state *EATING* before being allowed to safely reach its critical state. In the original dining philosophers system, one philosopher is allowed to proceed as soon as its neighbors are not hungry, meaning the philosopher can start *EATING* simultaneously with the adjacent philosopher that moves to state *THINKING*. With a delayed transition enabled by the condition **Delay2**, the second philosopher will not advance until the state of the first philosopher changes, so it waits an extra cycle.

This delay is obviously unnecessary if the safety rule **SP1** is in a state where the first philosopher can safely advance to the state *THINKING*. To allow the second philosopher to advance in these situations, it would need to observe the state of the **SP1** property monitor in its synchronization conditions. If the mutual exclusion and the property **SP1** were combined into a single finite state property, the synchronization conditions for the combined property could exploit the dependence to reduce the delays for the second philosopher. This benefit would come at the cost of more complex analysis and more specific synchronization conditions with higher computation overhead.

$DELAY = (ARM1\_PRESSinPRS\_READY) \vee (ARM1\_PRESSinUNLOADING)$

Nonoptimal delays cause different types of problems for executable systems. A nonoptimal delay causes the component to delay the completion of a transition by one interval until it can verify that the other component in contention for the limited resource has completed its transition out of the access state. The delay results in a reduction of performance due to the unnecessary wait until the component in the access state advances to its next critical section, and releases the one previously held. If the limited access properties apply to a cycle of consecutive states for a set of components, the unnecessary delays may form a cyclic dependency and result in a component deadlock. These deadlocks, as well as the reduction in performance due to delays, can be resolved and the component delays reduced using global dependency analysis techniques. Another approach to reducing the effects of nonoptimal component delays is the use of a system design pattern that breaks the dependency cycles and makes the consequent deadlocks unreachable. We will discuss this approach in section 6.2

6.1.4 Nonconflicting Nature of Safety Enforcement

We have shown that our synchronization method can enforce individual receptive safety properties by delaying components whose actions may violate the safety. We have also shown that by enforcing the individual receptive safety properties, we can produce a system that satisfies all those properties simultaneously. It also satisfies all nonreceptive properties whose receptive parts include the intersection of the enforced properties. The next theorem will show that the synchronization mechanisms that enforce individual receptive safety properties can be combined into a controller that enforces all those properties.

Theorem 6.3 *A delayed transition that enforces a receptive safety property* **P1** *can not cause a violation of another receptive safety property* **P2***.*

Proof:

Assume, on the contrary that the delayed transition τ causes a violation of the safety rule **P2***. Transition τ starts at state* S1 *and sinks in state* S1d *of a component* **comp***, and the state* S1d *is the delayed state where the component remains to avoid violating the receptive safety property* **P1***. The system state preceding the safety violation consists of the states* Sr2 *for the property* **P2** *and* S1 *for the delayed component. The transition τ is executed and* S1d *becomes the new state of the component* **comp***, activating the* **P2** *transition to the state* REJECT*. The transition to the* REJECT *state is enabled by the current system state including the state* S1d*, and the safety property* **P2** *executes the transition and detects the safety violation.*

From the standpoint of the component, the transition from S1 *to* S1d *does not count as a state change, so it does not cause any event observable by the safety property. That means that the safety property* **P2** *is either violated by a different controlled event, or that the property* **P2** *is a nonreceptive property and that it was violated by an environment event. Both possibilities contradict the assumptions of the theorem, and therefore the assumption that the transition τ was the cause of the violation is false. The delayed transitions can not cause safety violations for the receptive safety properties.*

This theorem proves that when GenEx modifies the components by adding the delayed transitions that enforce the receptive safety properties, the integrated system satisfies all properties. Since all receptive safety properties are enforced, the implementation satisfies its realizable requirements.

6.1.5 Correctness of Integrated Systems

We have shown that system interaction requirements, given in the form of receptive safety properties, can be represented independently, and components can be synchronized to enforce them. We have also shown that a strategy that enforces individual system requirements in effect enforces their intersection, which is the combined requirement of the system. Finally we have shown that the synchronization mechanism that enforces one receptive safety property can not enforce a behavior that violates other system requirements.

This means that, given a set of components and receptive safety properties that specify the system requirements, our method produces synchronization mechanisms that enforce each receptive safety property individually. Since these synchronization mechanisms are nonconflicting, the generated system enforces all of its required properties and satisfies its aggregate behavior requirements. Since the system includes a separate safety observer for each property rather than a combined state machine for all properties, the size of the generated code can remain smaller than the system state space.

6.2 Design Patterns for Deadlock–Free Systems

Deadlocks occur as a side effect of enforcing safety rules inconsistent with the component behaviors or with each other. When GenEx synchronizes a system to satisfy some safety rule, the component transitions are delayed for as long as the violation preconditions for that rule are satisfied by the system state. If the violation preconditions can not be invalidated by the continued execution of other components or environmental events, the delayed component remains blocked in the same state. If other components might invalidate the synchronization conditions, but are themselves blocked waiting for other components' actions, the system may deadlock with all blocked components waiting for each other to enable their progress.

The verification of deadlock freedom is a complex problem because it requires finding the possible deadlock states and then proving that those states are unreachable. The complexity of the reachability analysis is comparable to the complexity of the system state space, and makes this approach non–viable for complex systems. Instead of detecting deadlocks, we will provide simple design patterns that ensure the freedom from deadlocks for the synchronized application. The design patterns specify the form of the system safety rules and their relationship with the components, that will guarantee the deadlock freedom. The main goal of these patterns is to avoid the reachability analysis, and use static analysis instead.

No restrictions on the structure of the components are necessary to guarantee the deadlock freedom of a synchronized system, and they would not be acceptable because the main guideline in component design is their intended functionality. The components of process control systems generally have a cyclic control structure, corresponding to the nonterminating nature of the applications, but some initialization activities may require acyclic components or acyclic segments initializing the cyclic components. The structure is captured as a finite state machine, and can be analyzed using static methods. We regard the component control graph as consisting of advancing edges that lead it to new states, and returning edges that take the component to previously visited states.

6.2.1 Patterns for Deadlock–Free Design Using Limited Resource Access Rules

Limited resource access properties specify combinations of component states that should not occur simultaneously. The synchronization mechanism that enforces limited resource access allows the specified number of components to access, while making all others wait for the resources to become available. A component that holds some exclusive resource blocks all components waiting for that resource, and a cyclic blocking pattern produces a deadlock. The dining philosophers system is one example of deadlock–free design using limited resource access rules. We will now show a design pattern that guarantees that a set of components can satisfy a set of limited resource access rules and be deadlock-free.

The design of the dining philosophers illustrates a simple instance of a system where the receptive safety properties can be automatically enforced without the risk of generating deadlocks. There are no deadlocks because the philosophers are synchronized to either enter the critical section or wait in the *THINKING_DELAY* state. Since any philosopher in the state *EATING* must eventually go back to *THINKING*, it will make the shared resources (forks) available to the adjacent philosophers. The key to deadlock freedom in this system is the atomic nature of resource allocation. A philosopher waits until all the resources it needs for the *EATING* state are available, and then allocates them all.

We will define a mapping of component states to the set of natural numbers that will help us specify deadlock–free systems satisfying limited resource access properties. For the set of components' states $S_{all} = S_1 \cup S_2 \cup ...$, the mapping $M : S_{all} \longrightarrow N$ maps every state to a natural number, and based on this mapping the transitions of individual components are classified as being *advancing* or *returning*. A transition from state S1 to S2 is an *advancing transition* if $M(S1) < M(S2)$, and a *returning transition* if $M(S1) > M(S2)$. Two distinct states connected by a transition must be mapped to different numbers.

The deadlock freedom is guaranteed iff the limited resource access rules are specified in a way that satisfies two conditions for some mapping M:

- All limited resource access properties apply to states of distinct components that map to the same value.
- There exists a value k in the mapping and every simple cycle for every component contains a state $S_{free} : M(S_{free}) = k$, such that no limited resource access properties apply to any of those states.

Systems satisfying these conditions will have the form of a sequence of layers with arbitrary mutual exclusion requirements within individual layers, as illustrated in Figure 6.1a). The figure shows three components whose transitions are represented by full lines, and every state is marked with its mapping value. The dashed lines in the figure represent the mutual exclusion between the specified states. Every cycle for every component includes a state whose mapping is the number *1*, and those states are free from access restrictions.

The requirement for the mutual exclusion properties to apply to states within a single layer guarantees that the specified system is realizable, deadlock free, and starvation free. The requirement

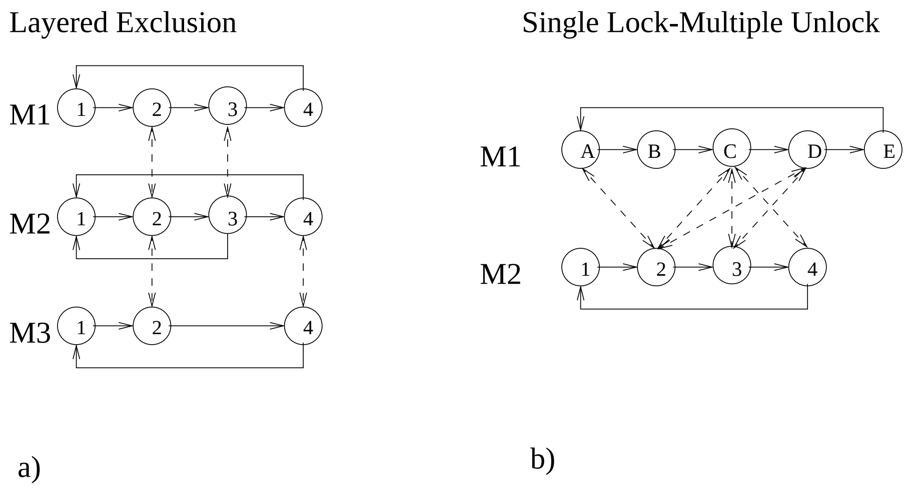


Figure 6.1: Patterns for deadlock–free design using limited resource access rules

for a layer of states with no restrictions guarantees that the implementation generated using local analysis will also be deadlock free. Deadlock freedom results from the fact that there is always some component that can complete its selected transition. That component is either in a state whose successor is the unconstrained state, or its next state requires currently available resources. Components advance until they eventually reach the unconstrained state. As a component enters its unconstrained state, it frees the resources reserved by its predecessor state, thus allowing other components to advance and free resources in previous exclusion layers. The proof of starvation freedom from the dining philosophers chapter applies to this system, and guarantees that every component will eventually be allowed to proceed to its next state, assuming strong extreme fairness for components leaving their critical states.

This design pattern requires the components to access the shared areas in the same order and that may not always be practical or appropriate. Another design pattern exists that allows the limited resource access properties to reference component states in arbitrary order. The implementation of a system with limited resource access properties will be deadlock–free if the specification satisfies the following conditions:

- If state S2 is a successor of state S1 in component C1, and a nonempty set of limited resource access properties applies to the state S1, any constraints on the access to state S2 must apply to states referenced by the constraints on S1.
- Along any simple cycle in any component, there must be a state with no restrictions on access

This design pattern models the single lock–multiple unlock strategy often used in the design of databases and concurrent systems, and it is illustrated in Figure 6.1b). The component transitions and mutual exclusion properties are again shown using full and dashed lines, respectively. The states are identified by alphabet letters for the first component and by numbers for the second, and we will refer to the mutual exclusion properties by the names of states they reference. Thus *A2* represents the mutual exclusion for state *A* of **M1** and state *2* of **M2**. This example shows that states *2, 3, 4* of **M2** are subject to decreasingly restrictive mutual exclusion properties, thus the transitions within this sequence can never be blocked by the states of the component **M1**. Since every cycle in a component has a state without mutual exclusion restrictions, the component eventually releases the locked resources and allows the other components to access them.

6.2.2 Deadlock–Free Systems using Sequencing Rules

Sequencing properties are defined by the fact that their violating transitions are enabled by the state of one component. Every violating transition for a sequencing property can thus restrict the execution for at most one component. A specific state of a sequencing property can restrict as many components as it has outgoing violating transitions enabled by states of different components. These restrictions remain in force at least until one of the synchronization transitions for that state is enabled. The restricted components are essentially waiting to be released by the components that enable the outgoing synchronization transitions. A deadlock occurs if there is a cyclic waiting pattern for the states of a set of safety properties and their respective restricted components.

We have shown in section 6.1.3 that synchronization mechanism for sequencing properties is optimal even when it is produced based on local analysis. That means that deadlocks can be caused by sequencing properties only if the properties are inconsistent with the system components. The production cell is one example of a system specified using only sequencing safety properties. The restrictions imposed by the sequencing properties in this system always apply to one state of one component, while the other restricted component enables safety rule transitions and eventually causes it to arrive in a state where the blocked component is allowed to proceed with its execution.

The design pattern for the deadlock–free sequencing systems imposes certain constraints on the structure of the sequencing rules. Every sequencing rule is limited to restricting at most two components, and unrestricted components may be referenced only by observer transitions. In any state of the sequencing property, only one of the restricted components may be used to enable violating transitions, while the other enables the synchronization transitions leading to the relaxation or removal of the restriction. Sequencing properties with this structure can only restrict one component at a time, and wait for the other one to relax the restriction or remove it altogether. The waiting pattern in this case corresponds to a directed graph and deadlocks can occur if there are cycles in this graph. Sequencing properties with this structure specify interleaving for specific actions of two components.

The first deadlock–free design pattern is the acyclic restriction graph for sequencing rules and components. Nodes in this graph correspond to the components and sequencing properties, and every sequencing property has an undirected edge connecting it to its restricted components. Any cycle in this graph represents one or more possible deadlock states, and an acyclic graph implies that deadlocks will never be caused by the sequencing properties. Deadlocks require cycles in the reference graph because the sequencing rules can block at most one component at any time while waiting for an action by the other. Any component referenced by only one sequencing rule can either be blocked or be free to proceed and enable synchronization transitions of the sequencing rule thus unblocking the other restricted component. Since the graph contains no cycles, there can be only finite sequences of components blocking each other through the sequencing rules.

The feed_belt in the second version of the production cell is referenced by only one sequencing property, and as such can not be the cause of a deadlock state. The robot and the presses in the production cell system and their sequencing properties form a cyclic restriction graph, and could potentially deadlock, and then block the **rot_table** and the **feed_belt**, propagating the effect of the deadlock. This possible deadlock is not reachable, and we will describe a design pattern for sequencing properties that guarantees deadlock–freedom even if the restriction graph contains cycles.

If the acyclic assumption is not satisfied, the specification of deadlock–free systems using sequencing properties requires additional constraints on the structure of the properties and their relationship to the components. The constraints informally require the safety properties to have their cyclic behavior synchronized with the components they restrict. When the components complete their cycles, the safety properties also reach the states where they started the cycle and are ready to impose the same deadlock–free restriction sequence.

Every sequencing property state with outgoing violating transitions blocks transitions of one restricted component while waiting for the other to reach a specific state. The blocked transition of one component is said to be waiting for a transition of the other restricted component that causes the safety rule observer to advance and remove the block. We can define a waiting dependency relation *Waits*, between transitions of components restricted by a sequencing property. If a transition *t1* of component C1 is blocked waiting for transition *t2* of component C2 with the safety property P in state sp, we define that $Waits((C1, t1), (C2, t2), (P, sp))$ holds.

We will describe the design pattern for deadlock–free design using sequencing rules defined as a simple sequence of advancing accepting transitions, with possibly multiple returning accepting transitions. Violating transitions can not make a part of a cyclic behavior, because they lead to $REJECT$ states which have no outgoing transitions. The 1-bound overtaking property defined for the dining philosophers and shown in Figure 4.4 has one simple cycle of accepting states, and it is an example of a safety rule with one sequence of advancing transitions followed by a returning transition. These properties have no conditional branches or alternative paths, but the approach can be generalized to properties with branching and alternative ways of reaching the returning transitions. Any sequencing property with multiple alternative ways of completing a cycle can be decomposed into linear properties whose advancing sequence represents one individual execution path of the original property.

System consisting of components and sequencing properties is deadlock–free if there exists a mapping *M* for the states of all components $S = (S_1 \cup S_2 \cup ...)$ and sequencing properties $P = (P_1 \cup P_2 \cup ...)$, and a mapping *M1* for the component transitions $\tau = (\tau_1 \cup \tau_2 \cup ...)$, where:

- $M : ((S \cup P) \longrightarrow N and M1 : \tau \longrightarrow N$
- $(\forall t \in \tau) M(src(t) < M1(t) < M(dest(t)$
- $Waits((C1, t1), (C2, t2), (P, sp)) \Longrightarrow M1(C2, t2) < M1(C1, t1)$

- $Waits((C1,t1),(C2,t2),(P,s1) \wedge Waits((C2,t3),(C1,t4),(P,s2) \wedge M(P,s1) < M(P,s2) \Longrightarrow M1((C1,t1)) < M1((C1,t4))$
- $Waits((C1,t1),(C2,t2),(P,s1) \wedge Waits((C1,t3),(C2,t4),(P,s2) \wedge M(P,s1) < M(P,s2) \Longrightarrow ((M1((C1,t1)) < M1((C1,t3)) \wedge M1((C2,t2)) < M1((C2,t4)))$
- Any execution of components C1 and C2 up to a returning transition, leads all sequencing properties P, where $Res(P) = \{C1, C2\}$ to their initial state of the cycle.

This mapping function labels every state and transition in the system, and its existence guarantees that the first execution of the advancing paths for the components and the sequencing properties will not cause a deadlock. The last condition specifies that the sequencing properties are cycle–synchronized with the components and return to the initial state of the cycle when the components do. This means that the execution of the system returns to a state that is equivalent to the initial state after every cycle. The deadlock freedom proven for the first execution of the system cycle thus holds in the subsequent transitions.

6.3 Detection of Non–receptive Safety Properties

Safety properties are defined as sets that can exclude a behavior only if some of its finite prefixes violates the property. Receptive safety properties are the subset of safety properties whose violations are caused by controlled actions. The class of safety properties includes many non–receptive properties, classified into two main groups as properties violated by environmental events, and time–dependent properties. Our method enforces only the receptive safety properties by delaying the occurrence of controlled actions that violate the properties. An effective method for enforcing receptive safety properties must be able to identify the potentially non–receptive properties, and warn the user about their existence.

The only way to verify that a safety property is not receptive is to detect a violation that is caused by an environmental event. This requires a full reachability analysis of the system behavior that may not be viable for complex systems. Static analysis can detect potentially nonreceptive properties by detecting *possible* safety violations caused by the environment that may not be *reachable*. These potential environment safety violations are reported to the designer who can choose to redesign the rules if they really are nonreceptive, or to implement them if the violations are unreachable in the system.

To identify the properties whose violations are results of environmental events, our method looks for violating transitions whose enabling conditions include monitored variables. Those transitions are, at least partially, controlled by the environment possibly making the properties they belong to nonreceptive. User can choose to enforce those properties using GenEx, regardless of their possible nonreceptive nature. GenEx will synchronize the components to prevent the safety violations caused by the controller, while ignoring the violations caused by the environment. This approach produces a reliable safe system when the monitored variables that enable the violating transitions are partially dependent on the system and the user can verify that they will not cause any safety violations.

6.3.1 Detection of Time Dependent Safety Rules

Time–dependent properties are identified by the fact that their violations may occur independently of any explicitly defined event in the system. The term *event* refers to the occurrence of a condition due to a change in the system state. A violation can occur without the occurrence of an event, iff any state for the FSM representation of the property is reachable with its violating conditions already holding. When the violating condition holds at arrival to a particular state and does not become invalid in the next execution cycle, that safety rule reaches its *REJECT* state. This safety violation may not occur if the components execute the appropriate transitions to make the violating condition invalid in the next cycle, before the transitions for the safety rule FSMs are executed. This property accepts certain system behaviors without accepting the same behaviors after a longer delay, thus it is clearly a time–dependent property.[1] While this property shows the potential of being time–dependent, it is by no means a given that it actually is, because the path that reaches the time–dependent violation may not be possible due to the component dependencies and synchronization resulting from other rules.

As was the case with safety properties whose receptiveness was questionable due to their reliance on environmental properties, GenEx can detect the potentially time–dependent safety rules and warn the user of their existence. Those rules can be used to synchronize the components and, provided the time–dependent execution sequences are unreachable, the system will satisfy the rules. The detection of possible time–dependent safety rules is done automatically, based on the relationship between the incoming and violating transitions for a state of a safety rule. For every state we need to compute its rejecting conditions and its incoming invariant, and if those intersect, a time–dependent safety violation can occur. The incoming invariant of a given state equals the union of the enabling conditions of all incoming transitions, and the rejecting condition is the union of the enabling conditions of all violating transitions leaving that state.

The incoming transitions, their enabling conditions and source states define the incoming invariants for safety rule states.The union of enabling conditions for the violating transitions with a source in a given state defines the rejecting condition for that state. If the set of rejecting conditions for a particular state intersects its incoming invariant condition, that state may require time–dependent behavior.

The pseudocode algorithm in Figure 6.2 illustrates the time–dependent property detection. For every violating transition with source state **st**, this algorithm analyzes the incoming invariants for the state **st**, and the enabling condition for the violating transition. If the enabling condition of the violating transition intersects the incoming invariant of the source state, the source state imposes a possible time–dependent requirement on the system behavior.

The safety rule **ARM1_PRESS1** given in Figure 5.5 in the two press system is an example of a potentially time–dependent property whose time–dependent execution sequences are unreachable. We will show how this detection algorithm finds the possible time–dependent requirement in the safety rule, and also show what makes this time–dependent restriction unreachable. The safety rule restricts the components **robot_arm1** and **press1** and also references the **robot_base**

[1] It is not a real–time property since our system has no time guarantees cycle execution, but the requirement of this rule is that a certain action be completed by the immediate successor state making it a hard quasi–real–time property.

```
- Foreach safety property P in the system
  - Foreach violating transition tv in P
    - Identify source state of tv in st
    - Identify enabling condition of tv in cv
    - Foreach transition t_in with destination st
      - Identify source state of t_in in s1
      - Initialize c_ex, exiting invariant condition of state s1 as true
      - Foreach violating transition tr with source state s1
        - c_ex = c_ex ∩ ¬(enabling_condition(tr))
      - Endfor
      - Identify enabling condition of t_in in c_in
      - Incoming invariant of st is c_inv = c_ex ∩ c_in
      - If c_inv ∩ cv ≠ ∅ then output s1 and st
        as a possible time--dependent path for P
    - Endfor
  - Endfor
- Endfor
```

Figure 6.2: Pseudocode algorithm for identification of time–dependent safety rules

component.

The time–dependent requirement of this safety rule is detected when **tv** identifies the violating transition from the state *SELECTED* with enabling condition **cv** = (**robot_arm1** = *EXT_TO_PRESS*), and **t_in** identifies the transition from *PRS_LOADED* to *SELECTED* with enabling condition **c_in** = (**robot_base** = *SELECT_FIRST*). The state *PRS_LOADED* has no outgoing violating transitions, and its exiting invariant **c_ex** is **true**. The incoming invariant of the state *SELECTED* is equal to **c_in**, the enabling condition of the incoming transition **t_in**. The intersection of **c_in** and **cv** is not empty, so if the **robot_arm1** starts to extend to press before the **robot_base** reaches the state *Select_FIRST*, the safety property would impose a requirement for the first arm to leave the *EXT_TO_PRESS* state immediately.

This safety property would impose time–dependent requirements if the components **robot_arm1** and **robot_base** happened to be in a specific pasir of states at the same instant. However, another safety rule **ARM1_BASE** blocks the robot_arm1 from entering the state *EXT_TO_PRESS* until the robot base is positioned with the first arm pointing to the rotating table or the selected press. This makes the time–dependent safety violation of the rule **ARM1_PRESS1** unreachable, and thus the rule is a receptive safety property when combined with the **ARM1_BASE** property. The exact verification of time–dependency for the safety rules is impossible, in the general case, without the full reachability analysis of the system. Static verification can show that a property has the potential for time–dependent behavior, and identify the possible violation sequences by analyzing a single safety property.

6.4 Enforcement of Non–receptive Safety Properties

An alternative way of enforcing the real–time, reachability and liveness rules uses the concept of a realizable part for the property. Every property of a system has a realizable part, consisting of all system executions that satisfy the property without the possibility of its violations. The realizable part of a safety property is a receptive safety property, and as such can be enforced using GenEx automated synchronization. This approach may be used to enforce real–time properties and safety properties whose violations are caused by environment events.

The production cell example shows the application of this technique to make a safe and reliable system. The safety properties that specify the occurrence of collisions and unsafely dropped blanks, use only sensor signals and/or timing data to determine whether a safety violation has occurred. The collisions between the robot arms and presses do not occur immediately when the press starts to rise, but if the press starts to rise before the arm is retracted,it becomes a question of timing whether the collision will occur. The only controlled action that determines the collision occurrence is the press closing. We construct the receptive safety rule that rejects press closing until the robot retracts. This rule specifies a realizable part of the collision freedom property, and is thus enforceable using GenEx. If some information was available on the relative speeds of the press and the robot arm, the realizable part might include some additional behaviors where press may start closing after a predefined interval without waiting for the robot arm to retract completely. Without restriction on the speeds, the observable event of a retracted arm is the only condition that guarantees the collision freedom for these devices.

Similar approach is needed with real–time properties whose enforcement can not be implemented within a single component. As we have shown earlier, some real–time properties can be implemented by the individual components and preserved by not specifying those component transitions as violating conditions for any safety rule. Other, more complex properties may not be implementable within a single component and may require synchronization between several components. These must be enforced by enforcing their realizable parts.

We will demonstrate this on a multiple server system with two classes of client requests, high priority real–time requests with minimal CPU requirements and long running non–real–time requests. The high priority tasks can be executed on either server, while the low priority tasks may be server specific, and can not be preempted once they start. Assume for simplicity reasons that the system has two servers, and that each of them has the capacity to handle all real–time requests. The real–time property requires the availability of resources to process the high priority requests, and this condition is satisfied if one server is always reserved for the real–time requests. The lower priority tasks require specific servers, so the available server must be switched periodically. The realizable part of the real–time execution property can be specified as all behaviors where at most one server can be executing a low priority task. The available capacity for the lower priority tasks is equal to one server, and if more is required the requests will get queued until overflow or the server will become a bottleneck resource for the system. Even if the servers are overloaded with the lower priority tasks, the high priority real–time tasks will always have an available server to execute them. The slowdown of the lower priority processing is the result of the CPU idling that is required to satisfy the real–time requirement, and is not a side effect of the automated synchronization.

This system is scalable to any number of servers with any distribution of servers for real–time tasks. The system can also be modified for different types of tasks, possibly with better information on the expected running time. An upper limit on the running time of low priority tasks would allow us to design a better availability function with higher CPU utilization.

Chapter 7

Automated Synchronization in Reengineering

Reengineering is the process of converting existing software into new applications suitable for different environment conditions. The term reengineering is more restricted then reuse, because it assumes the production of new software mostly from the components of the existing applications, while reuse envisions the use of some components in a number of systems. Reengineering is thus driven more by changes in underlying technology then by new functional requirements. Some examples of technology changes that drive the need for reengineering are the migration toward distributed execution, visual interfaces and collaborative environments like the Internet. All of these advances require ever higher levels of parallelism and availability of system functions. The increases in parallelism and availability increase the system complexity and the importance of controlling the component interactions. We will show how GenEx can be used to simplify this process, even for existing applications defined in a sequential programming language.

7.1 The AEGIS Tracking System

The AEGIS system tracks a number of moving objects and attempts to classify them based on their friendly or unfriendly nature and their proximity and movement toward strategic targets. The algorithms used in the classification are not dependent on synchronization, and they are encapsulated in one component making them irrelevant for the system interaction. The system also requires information gathering and graphical presentation of the processed data. The original system design is based on a set of independent processes sharing a common data repository.

Each component of the system has a specific function and cooperates with other components to satisfy the system requirements. The initialization of the system is performed by a loader that requires exclusive access to the shared memory. After the loader completes its function, the spreadsheet and tracker are allowed to start executing and accessing the shared memory. The spreadsheet executes independently from the other components and has no further synchronization requirements. The spreadsheet acts as a producer of data, and all of its updates are atomic, so the consumers can read them at any time and get consistent data. The tracker component uses the data produced by the spreadsheet, and computes the parameters to be used by the display and list components. The display and list can only function when data is available from the tracker, so they have to synchronize with the tracker and access the shared memory after the tracker's acceses.

7.1.1 Synchronization by a Controller Process

The manual implementation of the AEGIS Tracker is synchronized by a controller process and by implicit delays in some components. The controller synchronization is based on message passing, where components send the controller a message when they complete their critical actions, and the controller sends messages to the components that can proceed with their actions. This is a simple and efficient method for synchronizing small numbers of components for a serialized protocol. If additional constraints on interaction need to be imposed, the controller design is not scalable and would become a bottleneck instead of facilitating the component interaction.

However, the synchronization by the controller process makes it very simple to identify the interaction constraints in the system. A message arriving at the controller represents a system state where some actions are enabled, and outgoing messages correspond to the components whose actions are enabled. We can reconstruct the interaction requirements that the controller is enforcing, and specify them in a formal notation to use them for automated synchronization.

The first message the controller awaits is the message from the loader, confirming the initialization is complete. After receiving the message from the loader, the controller sends a message enabling the spreadsheet, and enters the body of the execution loop. The main loop consists of the tracker activation, where the controller sends the tracker a message allowing it to start processing the tracking data. When the tracker is done, it sends a message to the controller, and the controller then activates the display and list component, using appropriate messages. After activating the display and list, the controller goes back to the beginning of its loop and activates the tracker again.

The tracker component includes local delays that make it relinquish control of the CPU even when it is active. Without the delays, the tracker takes all available CPU capacity, thus blocking the interactive components. This implicit synchronization is never documented, and it is hard to reconstruct its purpose.

The control aspect of component structure is very simple, and driven by the synchronization mechanism. Each component has an initialization, and an active and passive state. The active state is when the component is executing its function, and the passive state is when it is done, or waiting to be allowed to activate again. The loader and tracker send messages to the controller when they leave the active state and enter the passive state. The entrance to the active state for all components except the loader is conditional upon the reception of a message from the controller.

7.2 Automated data processing extraction

The components of this system have significant data processing functionality, within a simple control structure. Our method operates with formal control structures, but the data processing aspect has to be preserved for the generated application to be equivalent to the original. We need to extract the data processing code embedded in the control structure of the component, and link it with the code generated for the synchronized components. The first step in this process is defining the semantics of the connection between the control structure and data processing code.

We consider the components to have a finite state control behavior that roughly corresponds to the control structure of the component implementation in a sequential programming language. Part of this control structure is unrelated with the interaction between the component and the rest of the system, so it can be abstracted away in the representation whose goal is system synchronization. The abstracted part of of the component can be assumed to implement its data processing functionality. A simple example of the partition into control and data oriented functionality is given in Figure 7.1, using the code for the **tracker** component of the AEGIS system. The left side of the figure shows the structure of the manually designed component, and the right side shows the equivalent description in the form of a control oriented FSM and embedded data processing code segments. The data processing code is implemented in the form of procedures associated with FSM transitions, and called when their respective transitions are executed. This example distinguishes between two types of data processing code: the real local data processing and the messages between the tracker and the controller. The local data processing is a part of the component functional description, while the messages represent the manual implementation of the synchronization mechanism.

A functional description of the **tracker** component is derived from the synchronized one by removing its synchronization mechanisms. The purely functional component consists of its control structure, and the associated data processing code. Given the structure in the GenEx notation, an implementation of this component can be generated to automatically include the links to the data processing procedures. The component preserves the data processing linkage information even when its structure is modified for synchronization purposes. After the system is synchronized, the generated code for each component preserves the functional behavior of the original.

We use an automated code extraction tool to separate the data processing component code from their control structure while preserving their relationship and the control dependencies. The tool requires the user to annotate the code that has to be extracted, by specifying the beginning and end of the data processing code associated with individual states. The code between the annotations is extracted into procedures to be called on entry to the respective states. This tool can handle embedded annotations where the data processing code for some states is between data processing code segments associated with another state. An example of this is when the code for one state ends in a conditional branch, and both branches include code for successor states.

The extracted code is copied into a separate header file, and grouped into procedures called by the generated code for the component control structure. Once the components are synchronized and the code for them is generated, the executable application can be created by adding the data processing code and the execution support kernel and compiling the system.

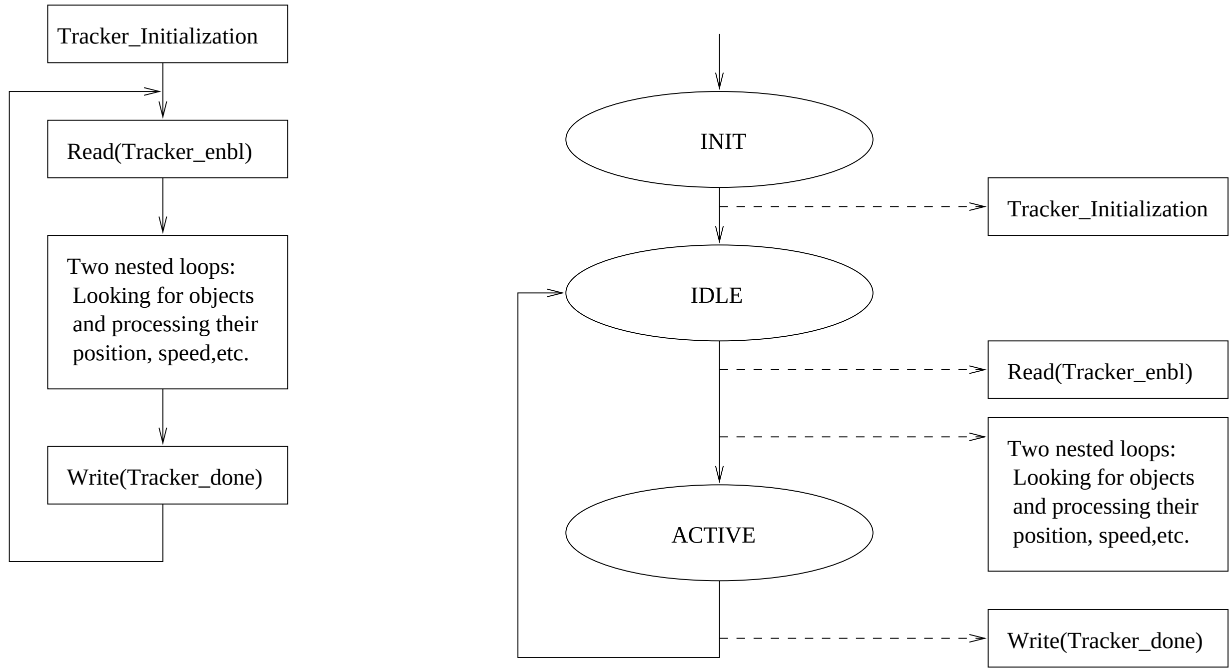


Figure 7.1: Tracker code structure

7.3 System Specification

The AEGIS system is specified as a set of finite state components and receptive safety rules that describe their interactions. The control aspect of component behavior is almost trivial, as shown by the **tracker** component FSM description in Figure 7.1. The behavior of the **display** and **list** components is identical to the tracker, while the **loader** and the **spreadsheet** operate as sequences with a self–looping transition in the last state.

The receptive safety rules are defined based on the system interaction requirements derived from the specification of the controller component in the manually designed version. The first of the two safety rules is shown in Figure 7.2a), and it ensures the safety of the system initialization by blocking the **tracker** and the **spreadsheet** access to the shared memory until the **loader** sets the initial values. When the **loader** completes the initialization, it proceeds to the *LO_DONE* state and enables the transition of the safety rule **INIT_SEQUENCE** from *INIT_ENABLED* to the state *SYS_ENABLED* where the **spreadsheet** and the **tracker** component can access the shared memory.

The second receptive safety rule for this system, shown in Figure 7.2b), specifies the valid sequences of accesses to the shared memory by the **tracker** and the **display** and **list** components. The **display** and **list** are blocked in their access to the shared memory until the **tracker** completes its access. This safety rule has a looping structure and it blocks the **display** and **list** whenever the **tracker** is waiting to enter the *TR_ACTIVE* state.

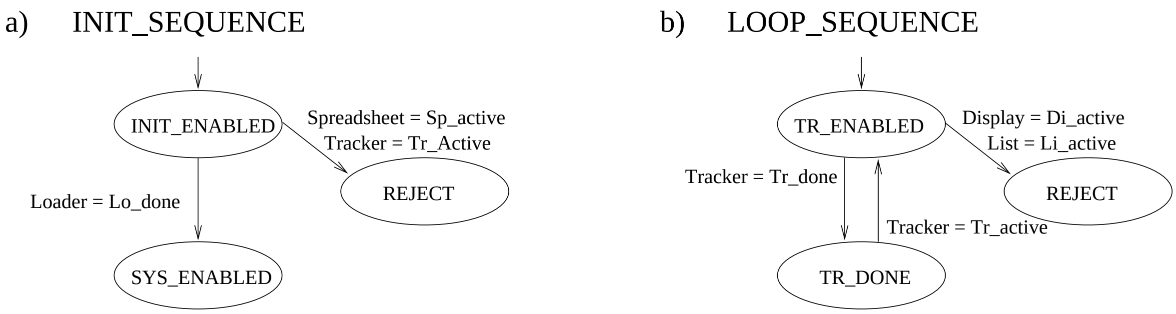


Figure 7.2: Receptive safety rules for the AEGIS system

Automated synchronization using GenEx produces an integrated system where the components consult the safety rules' state data in determining their enabled transitions. The synchronization of the components works independently of the runtime organization of the systems. The same generated code can, depending on the runtime support library, execute in a single process form or as a collection of distributed processes comprising one or more components and safety rules. This flexibility makes the automatically generated aegis system portable to a variety of environments including those without support for multiprocess execution required by the original manual implementation.

7.4 Requirement Modifications

Previous section has demonstrated how the control aspect of the aegis system can be separated from the data processing, and how the system can be reintegrated using automated synchronization. The resulting application is equivalent to the original implementation synchronized by a manually designed controller. If the system synchronization requirements change, increasing the controller complexity, a manual implementation may become a source of errors and ineficiencies. The system synchronized using GenEx can be modified by adding new receptive properties to the specification and the components will be automatically modified to enforce the new properties.

We can see that in the manual implementation, the tracker never waits for the **display** and **list** components to complete their actions before going back to the *ACTIVE* state. This means that the purpose of the synchronization is not to enforce exclusion of memory accesses, but to slow the display and list and reduce their CPU usage.[1] Neither **tracker** nor the **display** of **list** components have any sort of synchronization with the **spreadsheet** component, even though they are involved in reader–writer interaction on the shared memory. This causes no problems in the original implementation because the spreadsheet performs atomic updates of the values in the shared memory since at most one element of the spreadsheet, the one under the cursor, can be modified at a time. Were the spreadsheet to be substituted by a radar tracking device, all

[1]The **tracker** can access the shared memory before the **display** and **list** components complete their accesses previously authorized by the **controller**.

values would become modifiable at any time, and the question of shared memory updates would become critical.

Let us assume the specifications change, and require the mutual exclusion between the writer (**spreadsheet** or whatever substitutes it) and the readers. This requires an additional safety property that specifies that the writer can not be active writing data to shared memory while the readers are reading it. The simplest way to manually enforce this rule is to serialize the accesses of all four components, by requiring the controller to allow their accesses in a specific order. This imposes an unnecessary restriction on the execution of the critical components in the system, the **tracker** and the data acquisition component (the **spreadsheet** in the original implementation). These components must be executed regularly in order to guarantee the timely identification of the observed objects.

We can specify a modified system where the **tracker** and **observer** have higher priorities than the **display** and **list**, and all components require exclusive access to the shared memory. The exclusive access for the **display** and **list** serves to reduce the latency of the activation for the **tracker** and **observer**, since our system can not enforce preemptive priority. Additional constraint for the initial order of activation specifies that the **observer** activates first, the **tracker** is next, and the **display** and **list** can activate later.

The receptive safety rules for mutual exclusion and priority access are defined by simple finite state machines. The **observer** and **tracker** also must enforce bounded overtaking described to prevent any one of them from using too much CPU. When all these properties are specifies, GenEx produces a new synchronized implementation of the AEGIS system satisfying all modified requirements.

7.5 Summary

We have shown how a manually implemented and synchronized system can be modified into a set of components with data processing calls. These components can be automatically synchronized for any set of receptive safety properties and, assuming the properties define a realizable system, synchronized using GenEx and used to produce a reliable safe implementation.

Chapter 8

Conclusion

We have identified a subset of safety properties that can be enforced at the system level without requiring explicit synchronization mechanisms at the component level. We have developed a methodology and the supporting tools for the automated synchronization of concurrent software systems that enforces their receptive safety properties. By limiting the domain of enforcement to the receptive safety properties, we eliminated the need for the computationally costly reachability analysis. The constraint to receptive safety properties is not an overly restrictive requirement, since they are the only safety properties that can be enforced by open systems. Our synchronization method partitions the system analysis, and uses static techniques to drastically reduce the complexity of violation detection, as well as prevention.

8.1 Future Work

We currently have a functional set of tools that analyze systems with receptive safety requirements, and produce synchronized implementations that satisfy those requirements. Our tools also support model generation needed for formal system verification, and graphical representation of system state that makes system verification more intuitive. The generated applications are hardware and enviroment independent thus giving a lot of configuration flexibility to the user. This synchronization method could be improved by reducing some of the restrictions imposed by the definition of property domain, and by the current implementation method. Other improvements may be achieved in the runtime flexibility of the generated applications.

8.1.1 Extension of the Property Domain

Our method can synchronize systems to satisfy any set of finite state receptive properties. An obvious extension is to the set of context–free receptive properties, and also for context sensitive and unrestricted domain. The mechanism we use to enforce finite state receptive properties can enforce more complex properties, with only one additional reqirement. The enforcement of receptive properties demands the identification of violating transitions, thus the properties have to be represented in a form that makes their violating transitions explicit to the analyzer. Pushdown automata (PDA) style of representation for context–free receptive properties satisfies this condition.[1]

The receptiveness of a property is the fundamental condition that makes it possible to automatically produce an implementation that satisfies it. However, for many nonreceptive but realizable properties there exists a receptive property that represents their realizable part as defined in chapter 2. In some cases it may be possible to produce receptive properties by strenghtening the constraints of the given nonreceptive properties. One example where this may work was discussed in chapter 6 as time–dependent safety properties. The possibility of modifying time–dependent properties to ensure that incoming transitions are enabled by non–violating component states shows how this approach may work in some cases.

Another avenue of further research is the verification that the receptive properties are minimally and sufficiently restrictive to imply some non–receptive properties. The sufficient restriction means that a receptive property does imply the satisfaction of a non–receptive property, and the minimality implies that no unnecessary restrictions are made by the receptive property. If a receptive safety property is minimally and sufficiently restrictive, then it describes the receptive part of the respective nonreceptive property.

8.1.2 Optimized Synchronization Mechanism

The implementation of the synchronized systems is automatically generated from the component specifications and the modifications required by the synchronization. The code for each component contains procedures for every state, and they determine what transition is enabled for the current state. Only the procedure for the current state is executed, so the overhead in the execution is not excessive, but it can always be reduced. Some possible optimizations in the executable code include the elimination of redundant conditions, unreachable transitions, and the use of binary decision diagrams to shorten the decision tree.

The transitive blocking of sets of components synchronized for limited resource access properties leads to unnecessary slowing down that can be eliminated using global runtime analysis. The transitive and even cyclic blocking that causes deadlocks can be detected and components can be enabled to advance simultaneously while preserving the safety. This capability requires a global system state analyzer that detects the occurrence of transitive blocking patterns, and a different implementation of delayed transitions that disables the delays when based on a particular bypass signal.

An important source of inefficiency for systems synchronized using our method is the complete synchronous execution assumption. This assumption can and should be weakened, specially in distributed systems where the communication is both expensive and time consuming. The synchronization between components is required only when they may lead to the occurrence of

[1] Context–free or higher order languages are not closed under intersection, so their equivalent representation may not be a PDA. The synchronization conditions would be based on the states of the individual PDAs representing the receptive safety properties, so these properties would be enforced similarly to the finite state ones.

safety violations, and they can be allowed to execute freely when their actions have no influence on the system safety. Also the synchronization between certain components can be contained in a geographically or topologically limited area of the system without forcing synchronous execution with other non–local components. If a component can determine a unique enabled transition without knowledge of other components' states and system signals, it can execute asynchronously from other components.

8.1.3 Dynamic Reconfiguration, Migration and Substitution

The present implementaton of the synchronized systems is configurable to different execution environments, but only at compilation time. Once the application starts to execute, no components or receptive safety properties can be modified or added to the system. Systems with uninterrupted execution requirements demand the capability for removing subsystems from a running application and swapping new implementations in their place. The addition and modification of requirements is another example of a desirable runtime capability, as is the migration of components or subsystems.

These capabilities require no modifications to the analysis and computation of synchronization conditions, and their implementation is purely a question of different runtime structure of the applications. The support for the migration aspect can be inserted into the runtime support and requires minimal alterations of the user code that will enable the migration of data structures used by a component's data processing part. Both dynamic reconfiguration and substitution may require a different implementation structure for the components, where the evaluation of synchronization conditions is separated from the body of the component implementation, so that changes in the system have no influence on components' state and data processing. Both dynamic reconfiguration and substitution may require the user to specify the system states when the transformations are allowed, and how to get to those states.

Bibliography

[ABC+91] G. S. Avrunin, U. A. Buy, J. Corbett, L. Dillon, and J. Wileden. "Experiments with an improved constrained expression toolset". In *Proceedings of TAV4*, October 1991.

[AFB+88] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. "Software Requirements for the A-7E Aircraft". Technical report, Naval Research Laboratory, March 1988.

[AG93] J.M. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, pages 22–40, January 1993.

[AG94] R. Allen and D. Garlan. "Formalizing Architectural Connection". In *Proceedings of the 16th International Conference on SW Engineering*, 1994.

[AL93] Martin Abadi and Leslie Lamport. "Composing Specifications". *ACM Transactions on Programming Languages and Systems*, 15:73–132, January 1993.

[AW89] Martin Abadi and Leslie Lamportand Pierre Wolper. "Realizable and Unrealizable Specifications of Reactive Systems". *Lecture Notes in Computer Science*, 372:1–17, 1989.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. "Detecting Equality of Variables in Programs". In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.J. Dill, and L.J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[BG92] G. Berry and G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". *Science of Computer Programming*, November 1992.

[BK93] Eric A Brewer and Bradley C. Kuszmaul. "How to Get Good Performance from the CM5 Data Network". In *Proceedings of the 1994 International Parallel Processing Symposium*, pages 858–867, April 1993.

[Bro86] Michael C. Browne. "An Improved Algorithm for the Automatic Verification of Finite State Systems Using Temporal Logic". In *Proceedings of the Symposium on Logic in Computer Science*, pages 260–266, August 1986.

[CE82] E. M. Clarke and E. A. Emerson. "Synthesis of synchronization skeletons from branching time temporal logic". *Lecture Notes Comp. Sci.*, 131:52–71, 1982.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CG94] M. Chechik and J. Gannon. "Automatic Verification of Requirements Implementations". In *Proceedings of the 1994 ISSTA*, pages 1–14, Seattle, Washington, August 1994.

[CGK97] S. C. Cheung, Dimitra Giannakopoulou, and J. Kramer. "Verification of Liveness Properties Using Compositional Reachability Analysis". In *Proceedings of the 6th European Software Engineering Conference*, pages 227–243, September 1997.

[Che96] M. Chechik. *"Automatic Analysis of Consistency Between Requirements and Designs"*. PhD thesis, University of Maryland, College Park, 1996.

[CK95] S. C. Cheung and J. Kramer. "Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints". In *SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–151, October 1995.

[CK96] S. C. Cheung and J. Kramer. "Checking Subsystem Safety Properties in Compositional Reachability Analysis". In *18th International Conference on Software Engineering*, pages 144–154, March 1996.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 464–475, June 1989.

[CRR91] N. Halbwachs C. Ratel and P. Raymond. "Programming and veryfying critical systems by means of the synchronous data–flow programming language LUSTRE". *Software Engineering Notes*, pages 112–119, ? 1991.

[Dij] Edgser Dijkstra. "Two starvation–free solutions of a general exclusion problem".

[Dil88] David L. Dill. *"Trace Theory for Automatic Hierarchical Verification of Speed-Independent circuits"*. PhD thesis, Carnegie Mellon University, 1988.

[EC82] E. Allen Emerson and Edmund M. Clarke. "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons". *Science of Computer Programming*, 2(3):241–266, Dec 1982.

[FG94] Jeffrey Fischer and Richard Gerber. "Compositional Model Checking of Ada Tasking Programs". Technical report, University of Maryland, College Park, February 1994.

[Fra86] Nissim Francez. *"Fairness"*. Springer Verlag, New York, 1986.

[GMM90] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. "TRIO: A Logic Language for Executable Specifications of Real-Time Systems". *Journal of Systems and Software*, 12(2):107–123, May 1990.

[GS93] D. Garlan and C. Scott. "Adding Implicit Invocation to Traditional Programming Languages". In *Proceedings of the 15th International Conference on Software Engineering*, 1993.

[Har87] David Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[Hen80] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications". *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[HL96] M.P.E. Heimdahl and N.G. Leveson. "Completeness and Consistency in Hierarchical State–Based Requirements". *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE". *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992.

[Hoa78] C.A.R. Hoare. "Communicating Sequential Processes". *Communications of the ACM*, 21(8):666–677, August 1978.

[JEH79] Jeffrey D. Ullman John E. Hopcroft. **Introduction to Automata Theory, Languages and Computation**. Addison Wesley, Reading, MA, 1979.

[Kat93] Shmuel Katz. "A Superimposition Control Construct for Distributed Systems". *ACM Transactions on Programming Languages and Systems*, 15(2):337–355, April 1993.

[KIL+97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Lopes, Chris Maeda, and Anurag Mendhekar. "Aspect Oriented Programming". In *Proceedings of DSL97 – First ACM SIGPLAN Workshop on Domain–Specific Languages*, January 1997.

[LHHR94] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. "Requirements Specification for Process-Control Systems". *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[Lim93] Alvin See Sek Lim. *"A State Machine Approach to Reliable and Dynamically Reconfigurable Distributed Systems "*. PhD thesis, University of Wisconsin., Madison, Wisconsin, 1993.

[Lim96] Alvin Lim. "Compositional Synchronization". In *International Conference on DCS*, 1996.

[LL95] Claus Lewerentz and Thomas Lindner. *"Formal Development of Reactive Systems"*. Springer Verlag, Berlin, 1995.

[Mai93] Michael G. Main. "Complete proof rules for strong fairness and strong extreme fairness". *Theoretical Computer Science*, 111(1-2):125–143, April 1993.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[PM97] J.C. Park and R. Miller. "Synthesizing Protocol Specificationsfrom Service Specifications in Timed Extended Finite State Machines". In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 253–260, May 1997.

[Pur94] James Purtilo. "The POLYLITH Software Bus". *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[Sta90] John T. Stasko. "TANGO: A Framework and System for Algorithm Animation". *IEEE Computer*, 23(9):27–39, September 1990.

[Sta92] John T. Stasko. "Animating Algorithms with XTANGO". *SIGACT News*, 23(2):67–71, Spring 1992.

[WK95] Kevin G. Wika and John C. Knight. "On the Enforcement of Software Safety Policies". In *Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 83–93, June 1995.

[Yeh93] Wei Jen Yeh. *"Controlling State Explosion in Reachability Analysis"*. PhD thesis, Purdue University, August 1993.

[YS97] Daniel M. Yellin and Robert E. Strom. "Protocol specifications and component adaptors". *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

[YY91] Michal Young and Wei Jen Yeh. "Compositional reachability analysis using process algebra". In *Proceedings of the Symposium on Softvare Testing, Analysis and Verification (TAV 4)*, pages 49–59, October 1991.

[ZM+94] Nikolaj Bjorner Zohar Manna, Anuchit Anuchitanukul et al. "STeP: the Stanford Temporal Prover". June 1994.