Automated Computation of Decomposable Synchronization Conditions

Gilberto Matos James Purtilo

Computer Science Department and Institute
for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Elizabeth White

Department of Computer Science
George Mason University

ABSTRACT: The most important aspect of concurrent and distributed computation is the interaction between system components. Integration of components into a system requires some synchronization that prevents the components from interacting in ways that may endanger the system users, its correctness or performance. The undesirable interactions are usually described using temporal logic, or safety and liveness assertions. Automated synthesis of synchronization conditions is a portable alternative to the manual design of system synchronization, and it is already widespread in the hardware CAD domain. The automated synchronization for concurrent software systems is hindered by their excessive complexity, because their state spaces can rarely be exhaustively analyzed to compute the synchronization conditions. The analysis of global state spaces is required for liveness and real–time properties, but simple safety rules depend only on the referenced components and not on the rest of the system or its environment. Synchronization conditions for delayable safety critical systems can be computed without the state space analysis, and decomposed into single component synchronization conditions. Automated synthesis of decomposable synchronization conditions provides a solid groundwork for the independent design of system components, and supports reuse and maintenance in concurrent software systems. This approach to integration of concurrent systems is embodied by GenEx, an analysis and synchronization tool that integrates system components to satisfy a given set of safety rules, and produces executable systems.

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

1 Introduction and Motivation

Synchronization is an important aspect in the development of complex concurrent systems. Most programming languages lack synchronization support, or are limited to simple synchronization protocols (like mutual exclusion supported by Java). Due to the absence of explicit synchronization support, reliability verification for complex concurrent systems relies on informal methods. Inspections and testing can find errors, but their absence can not be asserted without formal verification. Reliability based on inspections and testing is probabilistic and depends on the structure and complexity of the system. This level of reliability is not satisfactory for the safety–critical systems.

Automated formal verification can verify the absence of errors, assert a system's reliability, and find errors earlier in the development process thus reducing the development cost [Man96]. Formal development methods are widely used in the research community [Man96], but their acceptance in development environments is lagging. The main reasons for this are the high level of skill that is generally required for formal verification, its lack of scalability, and its resistance to incremental system evolution [dLSA95].

Automated correction of inconsistencies between software systems and requirements is generally undecidable, but there are some domains where this approach is a viable alternative to manual error correction [EC82]. Most current formal approaches to concurrent systems use finite state machines (FSM) to model components, and finite state domains can be effectively analyzed and transformed. Error correction in FSM–based delayable systems is automated by computing the synchronization conditions from the detected requirement violations. The synchronization conditions restrict the system behavior, and the resulting system model satisfies its safety requirements [Lim96]. The automated rule–based synchronization process takes the description of a system and its safety rules, and produces a synchronized model that satisfies safety, and an executable implementation of that model (see Figure 1). The user specifies the system as a set of components and variables, where the components read and write the variables to communicate with the environment and with each other. The user also provides a set of safety rules that reject some states and execution paths representing safety violations. The automated synchronization tool determines the preconditions for the safety violations and uses them to compute the synchronization conditions for each component. Adding the synchronization conditions to the components makes the system comply with the given safety rules.

Previous research in the area of automated synchronization was concentrated in two domains. Theorical research discussed the complexity of automated synchronization for temporal logic assertions [EC82]. The complexity of system synchronization for temporal logic assertions is exponential, and not all formulas can be satisfied by synchronizing the system components. Attempts at practical synchronization concentrated on constrained systems such as hardware–based systems [Lim96] or a sequential safety kernel [WK95]. These domain restrictions provide a limit on the system complexity or make the synchronization trivial as in the safety kernel case.

Automated synchronization of concurrent software systems is the primary goal of our research. Analysis of the full state space for a software system is not a viable approach, and therefore we envision an approach for a restricted set of properties that allows the partition of the analysis and synchronization process. In such an approach, the complexity of the analysis and synchronization is independent from the complexity of the whole system, and depends only on the complexity of the referenced components.

We propose automated rule–based synchronization as an alternative method for developing and maintaining concurrent systems with complex safety requirements. This method, as implemented in the prototype tool GenEx, generates reliable synchronization for a variety of execution environments ranging from single processors to distributed systems. Unlike previous approaches, GenEx can synchronize systems even if their combined reachability graph is too complex to compute. The unlimited complexity of systems that GenEx can synchronize makes it a practical software development tool for complex concurrent applications.

Section 2 describes the formalisms and notations used in GenEx, and gives a brief description of the automated rule–based synchronization process on a simple concurrent system. A high level overview of the computation and use of the synchronization conditions is in Section 3, followed by a detailed description of the algorithms in Section 4. Section 5 gives a brief overview of other work related to our own, and explains the differences. Section 6 outlines the work we propose to do in the future, and the conclusion presents the benefits of this method.

2 Notation and a Driving Example

This section introduces the FSM–based notation for representing the behavior of concurrent systems, and describes the capabilities of automatic synchronization on a client–server access example.

2.1 Finite State Machine Representation in GenEx

GenEx uses a FSM notation inspired by the tabular notation used in SCR [Hen80], and functionally equivalent to a subset of the SMV notation [McM93]. Each component is a Mealy finite state machine [JEH79]. The system contains a set of Boolean variables

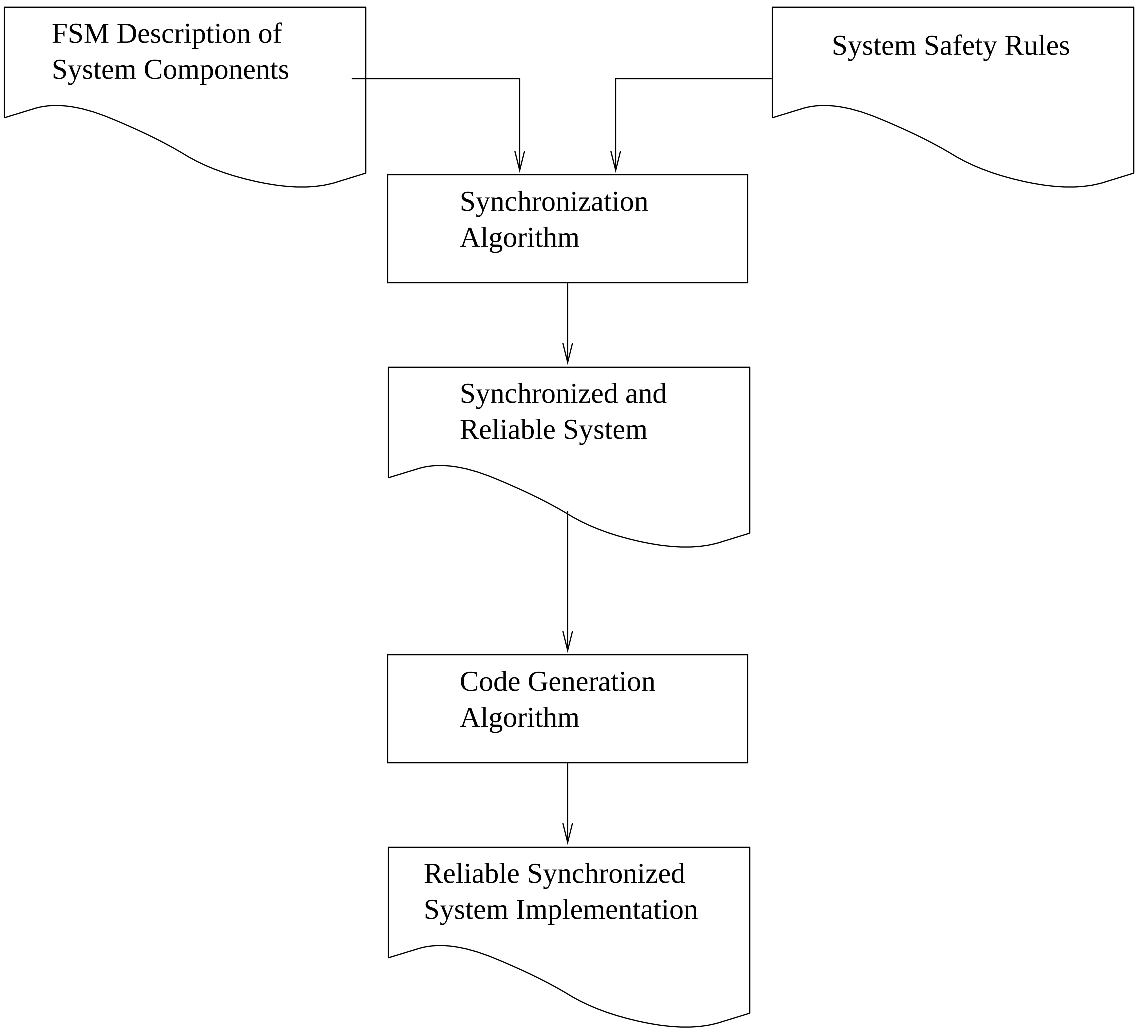

Figure 1: Automated Rule–Based Synchronization Process Overview

that can be monitored(external) or controlled(internal), and these variables are the inputs and outputs of the components or the environment. For a given group of components, controlled variables are those that contain the output from some of the components in the group, and all other variables are monitored. Only one component is allowed to control a given variable, by setting or resetting its value. In the client example shown in Figure 2, **req** and **done** are monitored variables and **accessing** is controlled by the component. Whenever a variable changes value, it produces an event, **@T(req)** when **req** becomes true and **@F(req)** when it becomes false. Events get enabled only immediately after a variable changes value, and get disabled when the variable's value stabilizes. The initial state of the system is defined by the initial states of all components and initial values of the variables.

Each component is specified as a stand–alone unit, and the system is specified by instantiating the components and the safety rules that apply to them. The specification of each component starts with its name and initial configuration. The initial configuration contains the initial state, and it may also contain some restrictions on controlled and monitored variables. The transition table for each state contains a list of transitions that may be activated in that state. The enabling condition, destination state and the effects define every transition. The enabling condition is a conjunction of event and variable values , or the keyword 'ELSE' for the default transition. The default transition is executed whenever none of the other transitions is enabled, and it makes the transition set complete. If the enabling conditions overlap for two transitions, their precedence is used to determine the priority when the component is executed. The effects associated with a given transition are changes in the value of controlled system variables, and calls to data processing segments.

GenEx uses the synchronous execution semantics similar to SMV [McM93], Esterel [BG92] and LUSTRE [CRR91]. When the transitions are executed, every component updates the values of its controlled variables, and shares that data with all other components. A transition is enabled when the associated combination of events and conditions is true. The monitored variables are sampled and all components receive their current value. Every transition takes a finite time to execute, and that is the sampling interval for the monitored variables.

The synchronous execution model simplifies the system analysis and the code generation for the synchronized system. The most important aspect of the synchronous execution is the immediate distribution of the global state information, that allows the components to make synchronization decisions locally, based only on the state data. While the system is analyzed assuming synchronous transitions in all components, this assumption is not practical for distributed systems. The synchronous execution assumption can be weakened for asynchronous and distributed environments while preserving the safety and reliability of the generated system.

A sample execution illustrates the semantics of GenEx in Figure 3. The time-line shows the changes in the state of a **client** component, and the values of the monitored and controlled variables leading to or resulting from the transitions. The event **@T(req)** causes the component to make the transition from the state **local** to **access**, and set the controlled variable **accessing** to the value **true**. As long as the component is in the **access** state, the value of the variable **req** and its events are irrelevant. When the monitored variable **done** becomes true, it enables the transition from the state **access** to **local**, that resets the value of **accessing** to **false**. All components execute one transition in parallel and share their state data and controlled variables before the next one.

The component specification in Figure 2a provides a description of a single component. System specification consists of a set of component specifications and safety rules. The a)

```
Component Client
Initial local, !req, !done, !accessing;
State local
        WHEN @T(req) access set(accessing);
        else local;
State access
        WHEN (done) local reset(accessing);
        else access;
```

b)

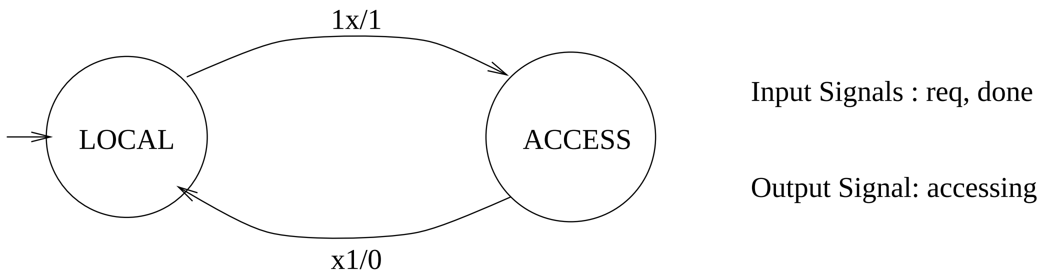

Figure 2: Textual and graphical FSM representation of a client component

safety rules describe the system behaviors that are undesirable in the final product, and restrict the domain of valid behaviors. Safety rules of interest for the client–server system deal mainly with the mutual exclusion and the desired ordering of the accesses. Mutual exclusion requires distinct clients to access the server during nonoverlapping intervals. The client accesses can be ordered using a specific access protocol such as priority or FIFO.

GenEx accepts the safety rules represented using finite state machines or regular expressions [JEH79]. Both representations are functionally equivalent but suitable for different types of safety rules. The FSM representation of safety rules is similar to the component representation with only minor distinctions. The FSM for safety rules have no interface to the data processing code, and they contain a state with the predefined name **reject** for the rejecting state. Rejection of an execution sequence by a safety rule FSM represents a safety violation. The FSM safety rules use the component's output and state variables, and monitored variables as their input, and produce their own state data as output. The state data from safety monitoring FSM is used by the synchronization algorithm to prevent safety violations.

The following regular expression–defined safety rule defines strict alternation of two clients' accesses, beginning with an access by the first client:

$$\text{AllPaths} \in (((\text{access}(1))^{+})((\text{access}(2))^{+}))^{*}$$

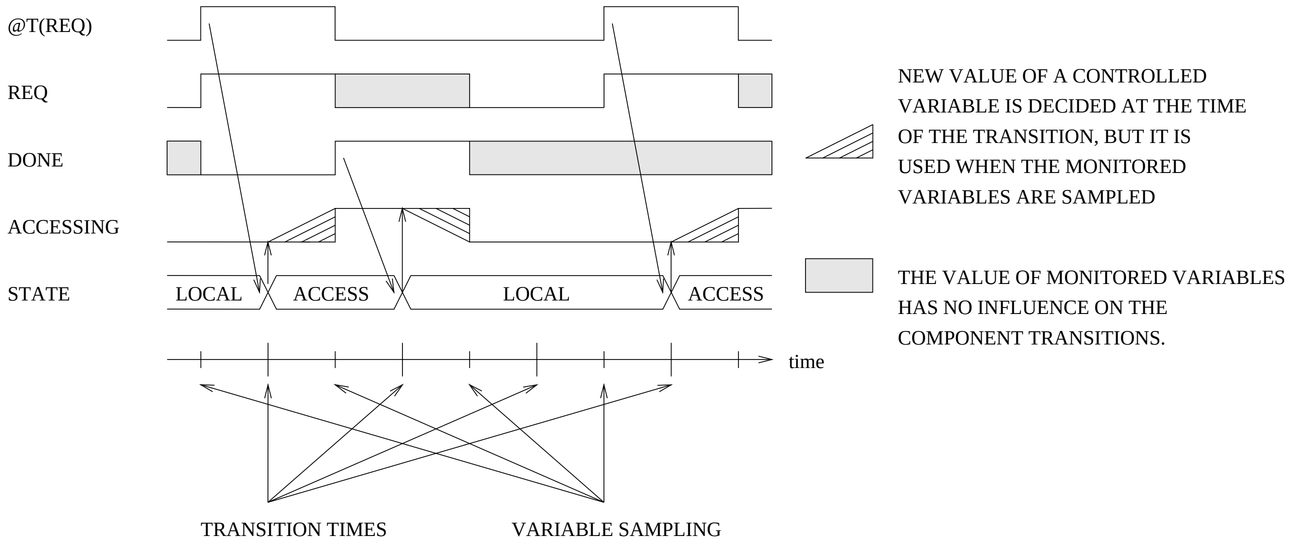

Figure 3: The execution semantics of the model

GenEx converts the regular expressions into the deterministic FSM representation before computing the synchronization conditions. Rejection of an execution sequence by a safety rule FSM represents a safety violation. The regular expression notation is more compact then the FSM notation, but its applicability is limited to simple rules like invariants and predefined sequences. For more complex rules the FSM notation offers better readability and simpler changes.

2.2 Client–Server System

The client–server system consists of a set of independent clients connected to a shared server. Depending on the nature of the interactions, as well as the capabilities of the server, the clients can access the server simultaneously, or be restricted to concurrent or exclusive access[1]. Different specialized subclasses of client–server systems can be derived by asserting different sets of safety rules, as well as new versions of existing systems when components or safety rules change. The main rules for the client synchronization deal with contol of simultaneous accesses, and with explicit ordering of accesses.

- **Mutual Exclusion**
  Some or all clients may be able to adversely influence each other when their accesses are concurrent, or the server may be unable to handle concurrent accesses. If this

[1]Simultaneous access applies when the server can accept several request simultaneously, while concurrent access refers to transaction–style accesses.

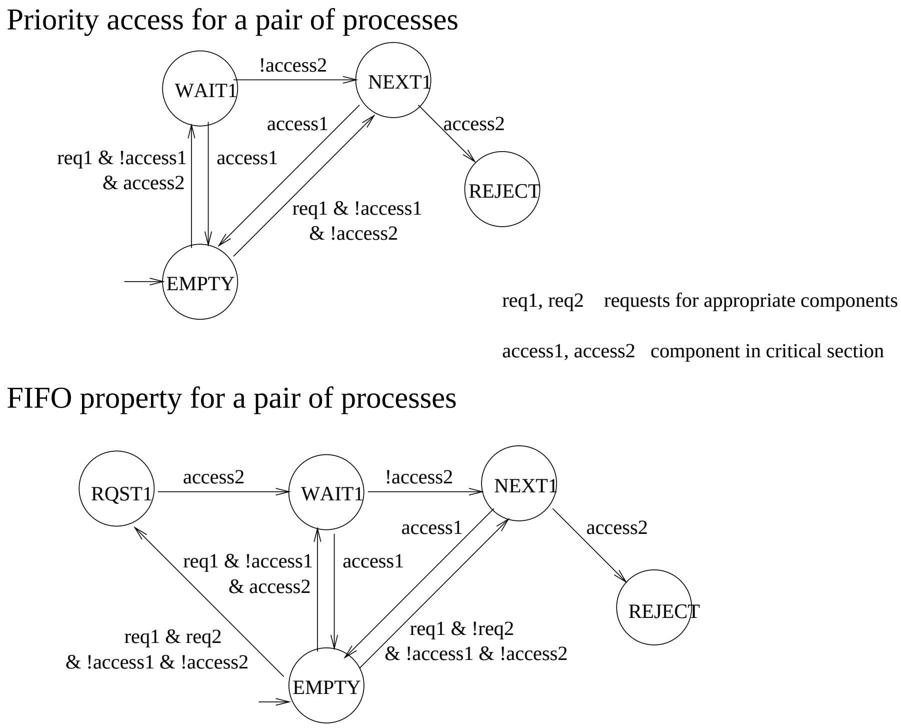

Figure 4: Graph representation of the FIFO and priority properties

is the case, some or all clients have to be restricted to mutually exclusive access. Mutual exclusion is a simple invariant safety rule, and it is defined by a single state FSM or by the regular expression below. The safety rule references the components' states and controlled variables to determine if the violation condition occurs.

- assert AllPaths $\in$ ($\neg$(access(2) & access(3)))*
  This is a singular mutual exclusion rule for clients 2 and 3
- forall($i, j \in 1..3, i < j$) assert AllPaths $\in$ ($\neg$(access(i) & access(j)))*
  Generalized rule for all pairs of clients in the system

- **Access Protocols**
  The order of client accesses may be a factor in the system performance and reliability. Access protocols allow the user to define the appropriate order for a given implementation, and make them a part of the system specifications. Priority and FIFO access protocols are illustrated by FSM in Figure 4. The priority access rule gives precedence to higher priority clients when lower priority clients are waiting for access, regardless of the order of requests.[2] The safety rule for priority access rejects

[2]In their original form, the clients make the transition to the access state immediately after receiving a request, but synchronization means that some accesses can be delayed. When components are delayed in

access by the lower priority component (**access2** in this case) while the higher priority component is waiting for access. The rule is not preemptive, and the higher priority requests will not force lower priority components to end or suspend their access. The FIFO access rule requires the accesses to the server to occur in order of their respective request signals. Figure 4b) shows the rule that rejects **access2** if **req1** was active before **req2** occured. This rule makes client2 respect FIFO and another similar rule is needed to make client1 delay on previous requests by client2.

- **Global Sequences**
  Some systems may require specific sequences of accesses, like round–robin access or interleaving of different groups of clients. Sequencing rules can synchronize the system into a desired global pattern of accesses. A simple sequencing rule requires the components client1, client2 and client3 to access the server in this order. This sequence may be necessary because their actions are dependent on the previous accesses by other components. The following rule requires this sequence of accesses.
  assert AllPaths $\in ((\text{access1})^{+}(\text{access2})^{+}(\text{access3})^{+})^{*}$

This system consists of a set of client processes, which get synchronized in their accesses to a shared server.[3] The safety rules in the system require mutual exclusion of client accesses, and FIFO policy for delayed accesses. GenEx processes this system specification and generates a synchronized and safe system implementation. The automatic synchronization allows simple modifications of the system, by adding or removing safety rules that restrict its behavior.

To illustrate the maintenance and reuse support, suppose the system is later modified to the priority access policy between two classes of clients. The system will be resynchronized according to the modified specifications without manual modifications to the components. A manually implemented client–server system can satisfy a fixed set of safety properties, but additions or modifications can require extensive design and verification rework. Formal reliability verification of the specifications is meaningless when the implementation or maintenance is done manually.

2.3 Synchronization of Clients

The two clients in Figure 5 access a shared server, and given the safety rules from the previous section, their simultaneous access is a mutual exclusion violation. Figure 5c) shows

accessing the server the FIFO policy limits their delay to the completion of the accesses that were initiated earlier.

[3]The server is not described here, but it handles only the data processing of the client requests, not their synchronization.

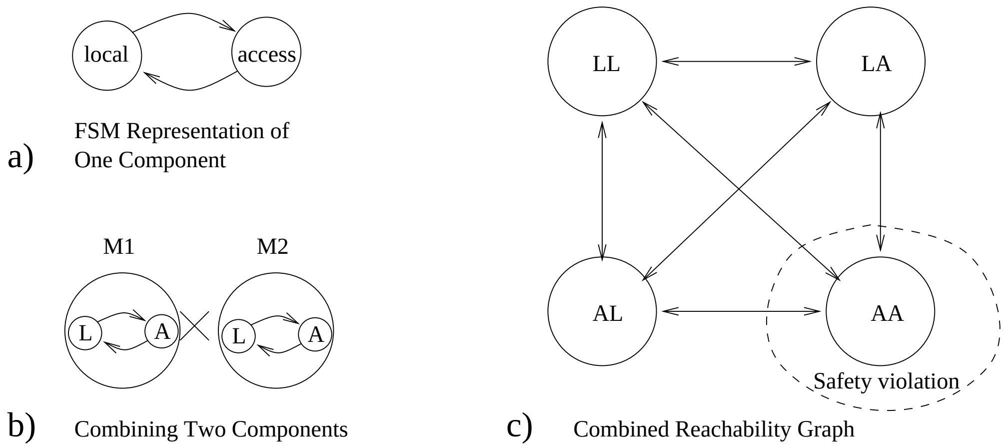

Figure 5: Conceptual representation of safety violation

the reachability graph that represents all execution sequences for this system, including some that violate the mutual exclusion rule. The state preceding the violation contains the information on the causes that lead to the violation, and this information can be extracted and used to prevent the safety violations from happening. In this case, the violation occurs when one component enters the **access** state while the other is already accessing the server, or when both components request access simultaneously. Using finite state models and model checking tools, these violations are easy to detect and analyze.

The concept of delayed execution is simple to capture in systems based on finite state machines. The transitions that can cause safety violations are modified by adding a delay state that is used to block the component until the conditions for proceeding to the destination state are met. The transition from **local** to **access** is the only transition that can cause the safety violation, and a delayed transition is shown in Figure 6a). The enabling condition for the original transition is combined with the safety violation preconditions to derive the enabling condition of the delayed transition. The delayed transition is not enabled unless the original transition is enabled and a safety violation is imminent. Figure 6c) shows how the safety violation state present in Figure 5 can be avoided by delaying the transition from **local** to **access** when it can violate the safety rule. One or more components are delayed when the completion of their transitions can cause safety violations. In the global reachability graph, the delayed transition of some component produces a redirected combined transition that leads to a safe state. This solution is very similar to using a semaphore or lock to guard the access, but it has the advantage of greater scalability and portability. The automatic synchronization is computed based on the given safety rules, and can be recomputed whenever the safety rules or components change.

Automatic synchronization of interacting processes, as outlined above, is conceptually simple, but the combinatorial complexity of the global interactions makes it computationally

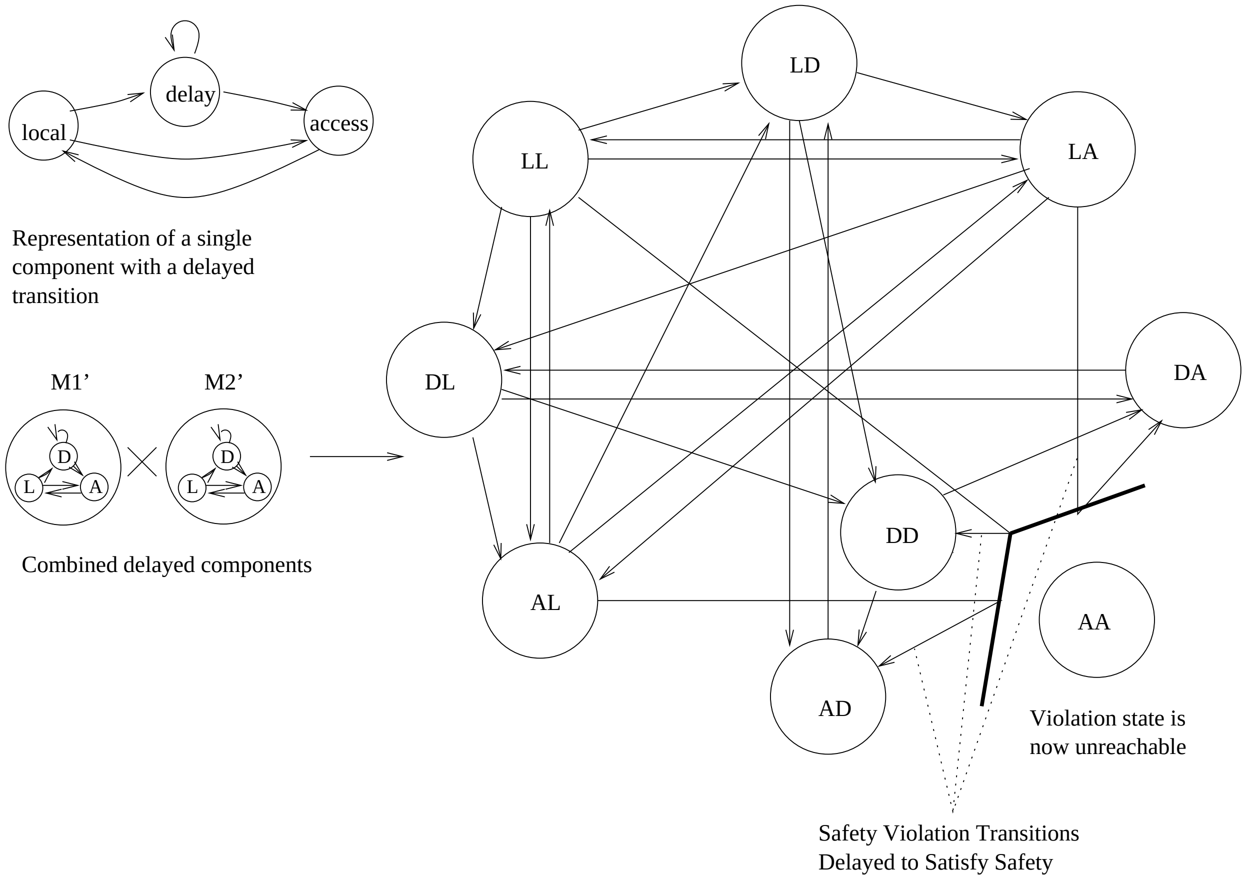

Figure 6: Delaying transition to preserve safety

intensive, with a potential for exponential growth. Figure 6 gives an example of the kind of modifications that are used to automatically synchronize systems. While the combined reachability graph for all components makes it simple to find safety violations, the size of the graph may make it impossible to construct. Our approach to automatic synchronization is oriented toward computing the synchronization conditions for each component without constructing the global reachability graph for the whole system. This is possible for a constrained but nontrivial set of safety rules.

GenEx uses only delays to synchronize the systems and make them satisfy the safety rules. The usage of delays obviously restricts the satisfiable safety rules to those that do not require real–time behavior. The non–real–time nature of satisfiable rules is a major factor in the decomposability of the synchronization process, because real–time requirements make the time a globally shared variable. A system can be synchronized to satisfy all given FSM–based non–real–time safety rules, and real–time rules can be detected and the user warned. Synchronizing the system for real–time rules leads to creation of deadlocks, as does the existence of conflicting safety rules. Even the synchronization for nonconflicting safety rules can sometimes lead to deadlocks, so the deadlock verification is an indispensable part of the development process.

3 Safety–Based Synchronization

The automated synchronization system computes the synchronization conditions for a system of user–defined components. The synchronized system consists of a synchronization skeleton and data processing code segments supplied by the user. Automated safety–based synchronization solves a conceptually simple but combinatorially very complex and time consuming problem, and helps produce a more reusable and maintainable system.

The user provides three distinct parts in the system definition: the component description, the safety rules and the system instance. The component description provides a formal model that defines the functionality of each component, and the interface linking the finite–state control to the separate data processing implementation. Every component is defined as a deterministic finite state machine, triggered by combinations of signal values. Every state in a component is associated with some component activity that can interact with other components or the environment [4]. All components are independent except for the interactions using controlled variables.

the local properties it has to satisfy, and that the

Another part of the system description is the definition of all safety rules that the system must satisfy. These safety rules are defined as finite state machines based on an alphabet that contains all component states and all monitored and controlled variables in the system. The safety rule for mutual exclusion in Figure 7a) shows how a safety rule is defined, and how it references system components and variables. The system instance specifies how the components are instantiated, and what safety rules are defined for them. The system description may also contain the equivalent description of the system when it is used as a component of a larger system. In our client–server example, all client modules behave in a similar way, and they can be described by instantiating the basic behavior and adding rules to achieve specific behaviors. Figure 7b) shows the instantiation of a system of clients where the safety rule for mutual exclusion is replicated for every pair of components, and priority access specializes the components into two priority classes.

The process of automatic synchronization, as illustrated in Figure 8, consists of several steps where the synchronization conditions are computed and decomposed to components,

[4] When a component is defined with more states then necessary, it increases the complexity of the analysis and the computation of synchronization conditions.

```
a)
Restriction mutex(i,j)
Reference client(i), client(j);
Initial state_OK & client(i)=local & client(j)=local
State state_OK
        WHEN (client(i)=access) & (client(j)=access) reject;
        else state_OK
end Restriction;
b)
System access_order;
Modules: client(i: i in 1 .. 4);
Restrictions:
forall(i,j in 1 .. 3 & i!=j) assert(mutex(i,j));
forall(i in 1 .. 2, j in 3 .. 4) assert(prio(i,j));
```

Figure 7: A FSM description of a safety rule and a sample system instance

and the executable code for them is generated. The automatic synchronization algorithm computes the component synchronization conditions that make the system comply with the safety rules. The code generation algorithm generates executable models of the synchronized system components, and links them together and with the data processing code.

Automatic Synchronization is based on delaying selected components in order to make safety violations unreachable. The delays are applied only on the violating transitions; i.e. those transitions that trigger the safety violations. The violating transitions are substituted by their delayed versions, where at least one component is delayed until its transition can be safely completed. A general assumption in this process is that every component is delayable and correct with respect to its requirements and that the goal of the synchronization is to satisfy the global safety rules. When all safety rules are satisfied, an additional check is performed to verify that the system satisfies its reachability and liveness requirements, as well as implicit deadlock freedom. This is necessary to verify whether the synchronization created any deadlocks. The satisfaction of these requirements guarantees that the refinements in control are not at the expense of the functional behavior of the system.

The computation of synchronization conditions requires the following steps:

1. **The System Expansion** step prepares the system for synchronization by enabling the components to detect when their transitions can cause a safety violation, and by providing them with an alternative execution that preserves safety. Delayed transitions are added to the components to make them synchronizable, and the global

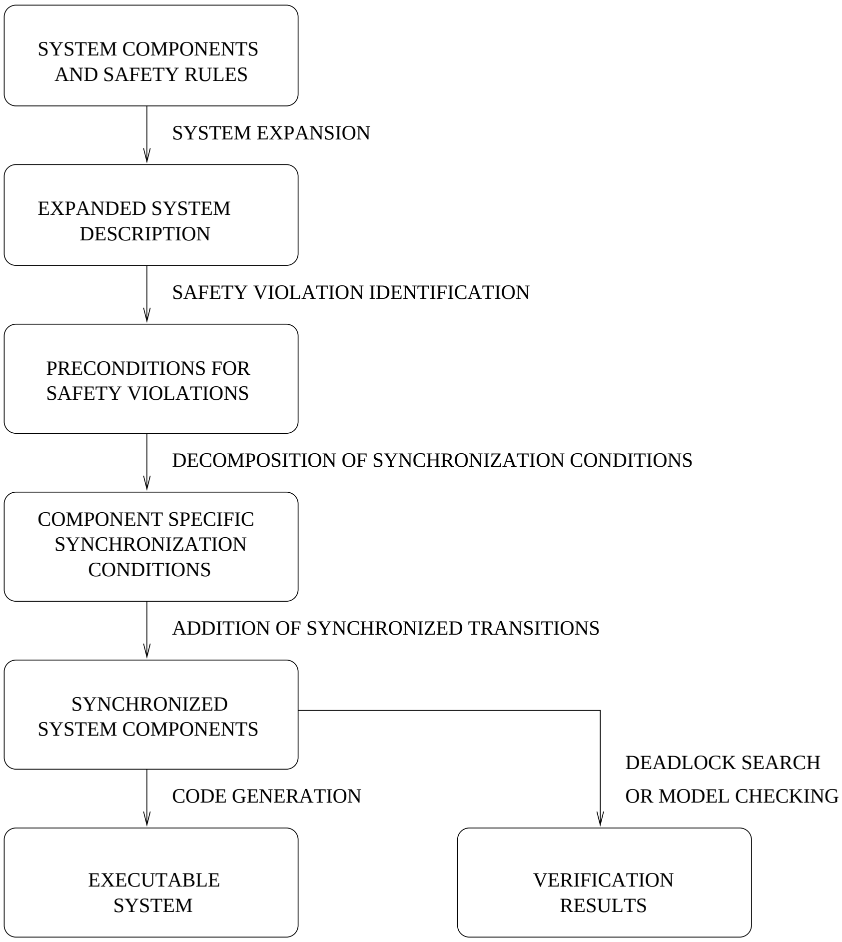

Figure 8: Process of Automatic Synchronization

system state is made accessible to every component that needs to detect potential safety violations.

The components with potential for safety violation are modified to delay rather then cause a safety violation. System components are defined as FSM that take the transitions between states when the system variables enable them. The synchronization mechanism in GenEx is based on the delaying of transitions when they lead to a safety violation. The concept of a delayed transition is implemented by introducing one additional state for every transition that might need to be delayed. The delayed version of the transition is given a lower priority than the original transition, so if no safety violations can occur, the transition is never delayed. Figure 9b illustrates the delayed transition implementation for the Figure 9a before the safety analysis. If the analysis finds that the transition could lead to a violation, the enabling condition of the delayed transition **REQ** will be combined with a set of conditions **ERRCOND**

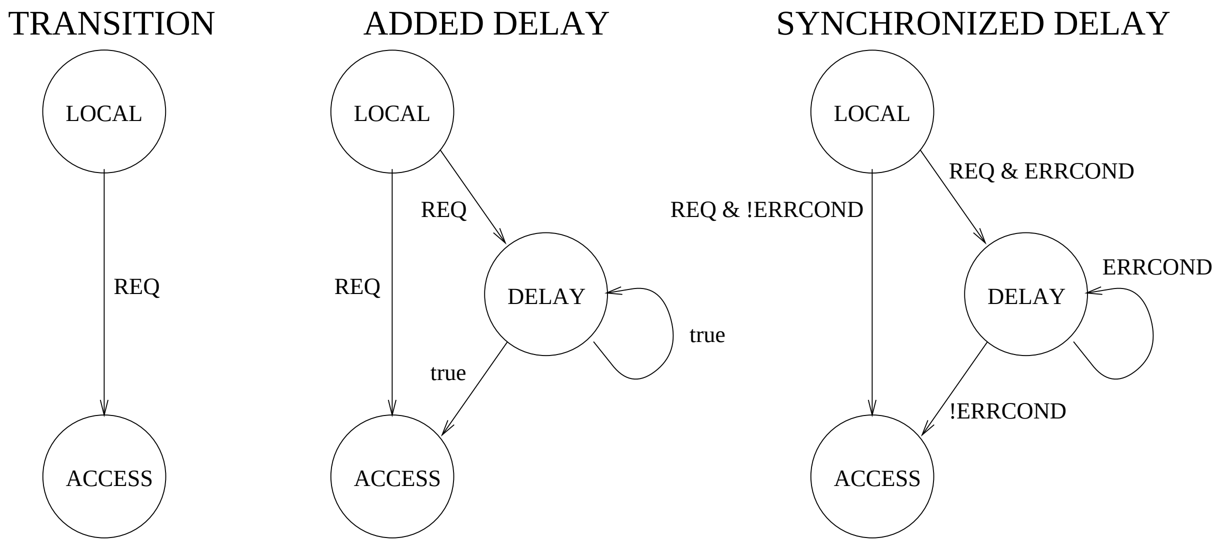

Figure 9: A simple transition, a blank delayed transition, and a delayed transition with added synchronization conditions

that is the precondition of the safety violation. This new version of the delayed transition will get a priority that is higher then that of the original transition, thus preempting its execution when a safety violation is imminent. The set of conditions **ERRCOND** is also used to substitute the **true** condition that activates the looping transition in the **delay** state, and its priority is raised above that of the transition to the destination state. The transition from the delayed to the destination state will occur only when there is no potential for a safety violation.

The system state variables are shared throughout the system to make components aware of each others' state. These extended variables contain information on the current states of the components, the priority of the components, and on the current states of safety monitoring rules. When components have the access to all this information in the decision on local transitions, the synchronization information can be effectively extracted during the execution. These signals are also the main interfacing mechanism between the components and the runtime support mechanisms. The priority variables are generated by the runtime support mechanism, to help the components decide on the transitions that are delayed. Other additional variables can be used for advanced synchronization of sets of simultaneous transitions. Added variables that encode the states of the components will not increase the complexity of the reachability graph because they represent state information in a condensed form. Priority variables will add to the complexity of the executable system, but they are used only when safety violations are corrected, and then they are added to the refined system without influencing the analysis process.

2. **The Safety Violation Identification** step, when the system behavior is checked with respect to the safety rules. Since the analysis is limited to non–real–time safety rules, all safety violations are detectable from the behavior of the referenced components only instead of the whole system. The preconditions of the detected safety violations provide the basis for the computation of the synchronization conditions. Several processes can detect the safety violations, with different levels of complexity and precision. GenEx currently detects safety violations by constructing the reachability graph for the subset of components that are referenced by the safety rule. This process has exponential complexity, but for a single safety rule with several referenced components it is within the reach of modern computers. A different approach to detecting safety violations is possible, but is not yet implemented. This static approach uses only the description of the components and safety rules to find potential safety violations, but it can identify unreachable violations, and the performance of the synchronized system may be reduced as a result.

   The combined reachability graph is constructed for components referenced by a safety rule, in order to determine if their interaction can cause a safety violation. The descriptions of components that are referenced by a safety rule are combined to generate a model of their behavior. The combined state space contains all reachable violations of the safety rule, allowing GenEx to identify them and correct them by delaying some of the components contributing to the violations. The components are combined with their respective delayed transitions, so the combined reachability graph contains the solutions to the violations. The delayed transitions are enabled by the same conditions that enable the original transitions, so they are treated as nondeterministic choices for the analysis purposes. By expanding the behavior that follows a delayed transition, GenEx analyzes all behaviors of the system, including those that will result from delaying the detected safety violations.

   Even if the reachability graph is too complex to construct, a static violation source graph (see Figure 10) can be created to identify potential rule violations. The static violation source graph is a graph that only contains violation states and all of their predecessor states, whether reachable or not. The complexity of this graph is very likely to be lower then that of the reachability graph, making it possible to analyze and synchronize more complex safety rules with more referenced components. The disadvantage of not creating the reachability graph is that deadlocked states may remain undetected, requiring further global checking.

   Consider the mutual exclusion rule for Client1 and Client2 and their combined reachability graph as in Figure 5c). Since the components Client1 and Client2 both have two states, the reachability graph can have at most four distinct states [5]. When

[5] In this case the size is bounded by the product of the component sizes, because the safety and controlled variables are dependent on the current state of the components. The safety rules like priority or FIFO, with multiple non–rejecting states, and controlled variables that contain information on previous states

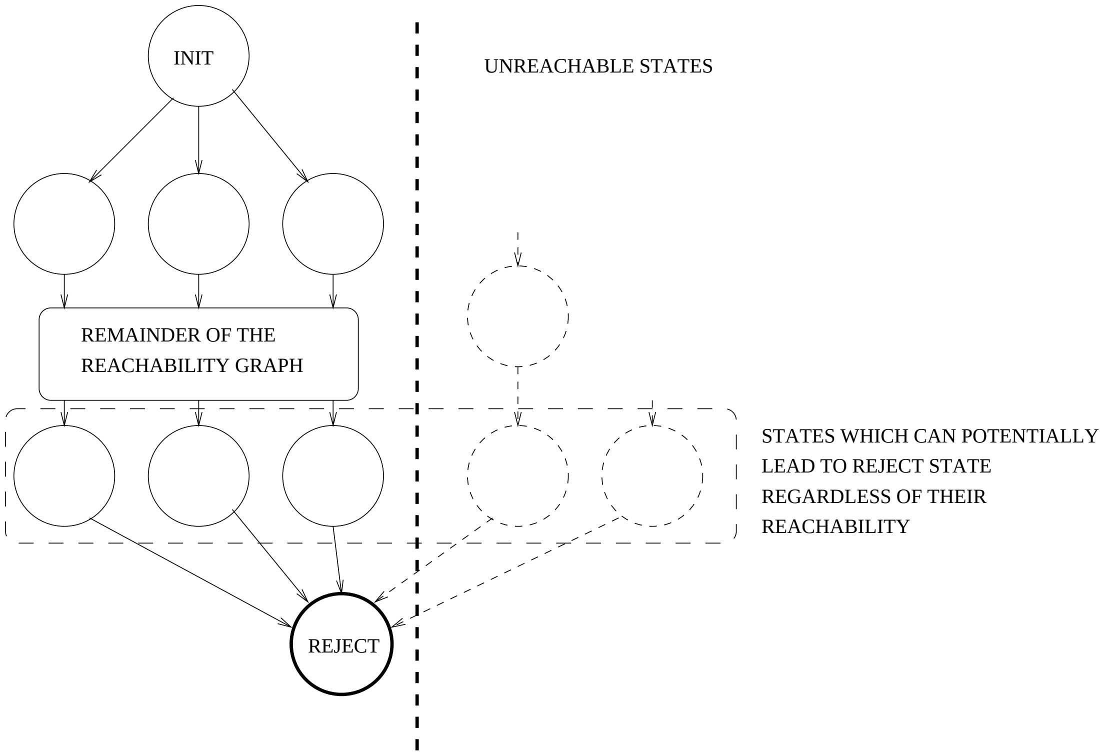

Figure 10: The violation precedence graph, contains all system configurations from which a violation would be possible.

the system is expanded with the delayed transitions, the reachability graph becomes similar to the one in Figure 6c), with the transitions to the safety violation state still active.

The Figure 11 shows the transition table for the initial state in the combined reachability graph. It shows the enabling conditions for transitions to different system states. Destination state is represented by a pair of component states, and the safety rule state. The state **¡access , access, reject¿**, reachable by the transition number six, represents a safety violation. All violations of the safety rules will appear as global states that include the rejecting state for some safety rule FSM. The enabling condition for the transition to the safety violation state also enables transitions 7, 8 and 9, to states <**access, delay**>, <**delay, access**> and <**delay, delay**>. Each one of those transitions can substitute the violating transition as necessary. The precondition for this safety violation is a conjunction of the current system state and the enabling condition for the violating transition; in this case it is **local1 & local2**

increase the potential complexity of the reachability graph in proportion with their complexity(number of states for safety rules or two for boolean controlled variables).

```
      Enabling                 <Client(1),Client(2),mutex12>
      Condition                Destination State

1     !req1 & !req2            <local  , local  , state\_OK>
2     !req1 &  req2            <local  , access , state\_OK>
3     !req1 &  req2            <local  , delay  , state\_OK>
4      req1 & !req2            <access , local  , state\_OK>
5      req1 & !req2            <delay  , local  , state\_OK>
6      req1 &  req2            <access , access , reject   >
7      req1 &  req2            <access , delay  , state\_OK>
8      req1 &  req2            <delay  , access , state\_OK>
9      req1 &  req2            <delay  , delay  , state\_OK>
```

Figure 11: Combined transitions from the state <**local,local**>

& req1 & req2.

3. **The Decomposition of Synchronization Conditions** is a process where the safety violation preconditions are partitioned into synchronization conditions for components. Every synchronization condition consists of the violation precondition and a selection part that nondeterministically chooses the delayed components. Whenever a combined transition leads to safety violation, a nondeterministic decision is added to choose some safe subset of transitions that are taken, while the rest are delayed. The transition leading to <access,access> in Figure 11 can be removed and substituted by the transitions to states <access,delay>, <delay,access> or <delay,delay> that have the same enabling conditions [6]. One of the components client1 or client2 has to be delayed to preserve safety, and this nondeterministic choice can be represented as relative execution priority. The variable prio12 represents this nondeterministic choice, and is randomly controlled by the runtime support environment. Synchronization condition for each client guarantees that the client will delay its transition only if violation preconditions occur, and the union of all synchronization conditions for a given violation is equal to the violation precondition, guaranteeing that the violation will never occur in the execution.

   The component client2(client1) is delayed when the violation precondition occurs and the variable **prio12** is **true**(**false**), meaning that client2(client1) has lower priority. The resulting combined transitions preserve safety, by redirecting the violating tran-

[6] The state <**delay,delay**> is reached only if both components are delayed, and this safety rule can be satisfied even with a single delayed component. Nevertheless, the transition to the state <**delay,delay**> might be required in an execution when a third client is in its **access** state, and also requires exclusive access.

```
        Enabling                 <C[1]   , C[2]  >
        Condition                Destination State

 !req1 & !req2                   <local  , local >
 !req1 &  req2                   <local  , access>
 !req1 &  req2                   <local  , delay >
  req1 & !req2                   <access , local >
  req1 & !req2                   <delay  , local >
  req1 &  req2 &  prio12         <access , delay >
  req1 &  req2 & !prio12         <delay  , access>
  req1 &  req2                   <delay  , delay >
```

Figure 12: Reachable states that preserve the safety rule

sition to a safe system state. Delaying both components is not necessary because a single delay is sufficient to avoid the safety violation. This means that no synchronization condition has to be generated for that case. The combined transition table for the state <local,local> after the removal of the mutual exclusion violation is shown in Figure 12.

4. **The Addition of Synchronized Transitions** is a process that uses the decomposed synchronization conditions to create new delayed transitions for components. This results in the creation of synchronized components that can be integrated without causing safety violations. These transitions are assigned a higher priority then the existing transitions in order to preempt their activation when the synchronization is necessary.

   To guarantee the preservation of the mutual exclusion rule starting from state $<$**local,local** $>$, we have to add one delayed transition to each of the two components. Adding a delayed transition to **client2** with the enabling condition $\mathbf{local1}\wedge\mathbf{local2}\wedge\mathbf{req1}\wedge\mathbf{req2}\wedge\mathbf{prio12}$, and giving that transition a priority that is higher than the original transition to the **access** state guarantees the mutual exclusion when **prio12** holds, i.e. when **client1** has priority. No changes are necessary in **client1** when it has priority because the delay of the other component is sufficient to satisfy the safety. The enabling condition $\mathbf{local1}\wedge\mathbf{local2}\wedge\mathbf{req1}\wedge\mathbf{req2}\wedge\neg\mathbf{prio12}$ for the delayed transition of the component **client1** guarantees that the mutual exclusion is satisfied when the component **client2** has priority. This synchronization step is executed for every combined state that can lead to a safety violation, and thus more than one version of the delayed transition may be necessary. For example, the component **client1** requires another delayed transition with the enabling condition $\mathbf{local1}\wedge\mathbf{access2}\wedge\mathbf{req1}$ to guarantee that it delays until the component **client2** exits

its access state.

5. **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 will result in the existence of deadlocked states. Another source of deadlocks is the circular dependence between synchronized components. GenEx generates a model of the synchronized system in the SMV [McM93] notation that allows symbolic checking of very complex systems. The 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.

   The 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. This static method is currently not a part of GenEx, but it is a planned addition, and will be discussed in more depth in the proposed work section.

   Together with deadlock verification, model checking tools can verify that the synchronized system satisfies some reachability, liveness or real–time specifications. These classes of properties either propagate dependencies between parts of the system, or are unenforceable because they require control of the input from the environment. These properties are very important for the correctness of a system, so even if enforcing them is not an option, their preservation can be formally verified. The 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. [7] The model checking can be done with the original components, and all the properties that can be satisfied without violating the safety will be preserved in the synchronized system.

6. **Code Generation** produces the executable versions of all synchronized components, as well as the interfaces to the runtime support environment and links to the data processing code. The code is generated separately for every component and includes its delayed transitions. The components can be grouped for execution in arbitrary ways, in a variety of execution environments. The runtime support is currently

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

available only for centralized execution, and more general distributed versions are planned.

The generated code is equivalent to the synchronized components, and the whole system is guaranteed to satisfy the safety rules that it was synchronized for. Links to the external data processing code are also generated according to the component specifications. The priority signals are generated by the runtime support, and they can be randomized to guarantee the fairness of the system. The fairness in the selection of delayed components guarantees the preservation of the liveness properties in the synchronized system.

4 Detailed Algorithm Description

In this section we will concentrate on the details of the automated computation of the synchronization conditions. The synchronization process consists of the following phases: single rule synchronization, global deadlock cleanup, and liveness checking. Two distinct algorithms can be used to compute the necessary synchronization between the components, and the choice depends on the number and complexity of the safety rules and components that influence them. The deadlock prediction algorithm is global and is intended to prevent deadlocks between components whose combined reachability graphs were not constructed in the synchronization phase. Additional liveness and reachability checking can verify if the synchronization prevents the system from completing its functional tasks.

4.1 Single Rule Synchronization Using the Reachability Graph

Combining related components is the first step in the reachability–based analysis of system compliance with the safety rules. Only the components which are relevant to the property are combined, thus limiting the complexity of the combined state space. This complexity is a decisive factor in the practical applicability of the reachability–based analysis. The analysis can be limited to the referenced components because the system is being synchronized to satisfy safety rules. Safety rules depend only on the referenced components, so the synchronization that satisfies them can ignore other components.

Since all components are represented by finite state machines, the combined state space is a finite state machine. The safety rules are represented by FSM, and they have to be included in the generation of the combined FSM, because some safety violations depend on the previous safety related states of the system. The construction of the combined reachability graph starts with the initial combined state **s0**, that is the combination of initial states for all combined components, and the initial state for the safety rule. The transitions in the combined reachability graph contain a set of component transitions, and a transition for the safety rule. The component transitions **t1** of component **c1** and **t2** of **c2** can be combined if they fall in one of the following categories for **every** system variable **v**.

- Both **t1** and **t2** require **v** to be **true** or **@true**.
- Both **t1** and **t2** require **v** to be **false** or **@false**.
- The enabling condition of at least one of the transitions **t1** and **t2** is independent from **v**.

The combined reachability also requires the consistency of the transitions with the preceding combined system state. Transition **t1** of component **c1** can take place in state **s** if the state of **c1** in **s** is the source state of **t1**, and one of the following conditions holds for **every** system variable **v**.

- Transition **t1** requires **v** to be **true**(**false**) and the present value of **v** in state **s** is **true**(**false**).
- Transition **t1** requires the event **@true**(**@false**) to hold for **v**, and the present value of **v** in state **s** is **true**(**false**), and previous value of **v** is **false**(**true**).
- The enabling condition of the transition **t1** is independent from **v**.
- Previous and present value of **v** are unrestricted in **s**. The value for a monitored variable in a state is unrestricted when the combined transition to that state is enabled regardless of the value of the variable.

The transition of the safety rule FSM also has to be consistent with the component transitions in the combined transitions. The safety rule transition is enabled by the destination states of the component transitions and by the new values of the controlled variables. In other words, the transition of the safety rule is selected based on the result of the combined transitions of all components. The transition computation for the reachability graph has two phases: the composition of the component transitions, and the safety rule transition. When the component transitions are combined and verified for consistency in the source state, the resulting state is encoded into the expanded state variables. The enabling condition for the safety rule is computed from the expanded state variables. If the enabled transition for the safety rule leads to the **reject** state, the combined destination state is a safety violation state, and the conditions that enable the combined transition are the safety violation precondition. If any other state is the destination state for the safety rule transition, the combined destination state is safe.

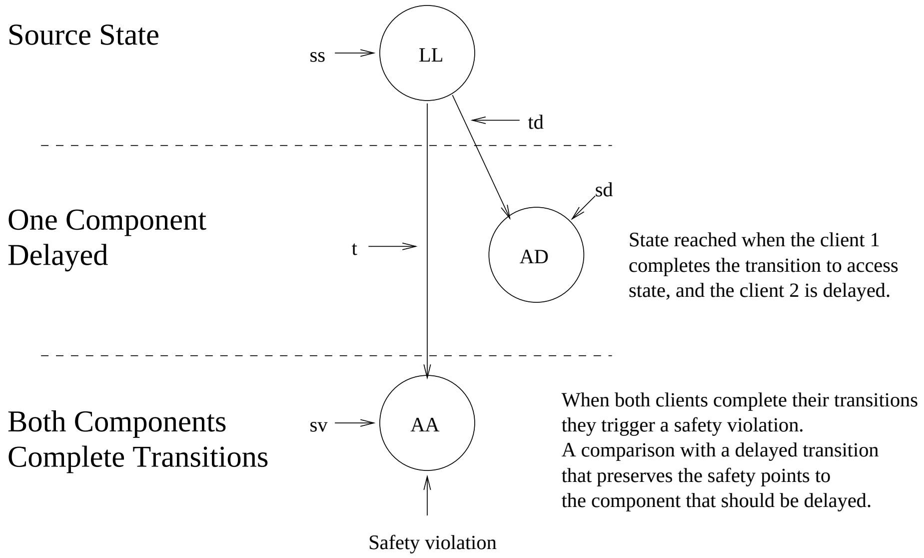

Figure 13: Selection of delayed components and delay conditions

Every consistent combined transition from a state in the combined reachability graph leads to another state in the graph. The graph is constructed by exhaustive expansion of every new state to find its successors until all transitions lead to some already explored state. Only transitions from the safe states are constructed and their destination states added to the graph. The safety violation states will be made unreachable after the synchronization, and that will make their successors unreachable as well.

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. The identification of safety violation states is a part of the reachability graph construction. This is done to limit the construction of the reachability graph to only those states that can be reached without safety violations. This way, the states that are only reachable after safety violation states will not be included in the reachability graph. The selection of delayed components is based on their contribution to the safety violation. A component has to be delayed only when the completion of its transition leads to a safety violation that is avoided when the transition is delayed.

For every pair of states **ss** and **sv**, connected by a combined transition **t**, where **sv** is a safety violation state, the delay of the combined transition is propagated to the components that contributed to the violation. Analysis of the differences between the states **ss** and **sv** identifies the components that changed their state or controlled variables in the transition **t**. To identify the components that caused the violation, the analysis looks at all states **sd** reachable by a delayed transition **td** that satisfy the safety rules. The difference between the delayed transition **td** and the violating transition **t** determines the condition that is propagated to the component to trigger the delay.

Figure 13 shows the three system states used for the computation of synchronization conditions. When a delayed transition **td** satisfies the safety rule violated by the transition **t** and the combined transitions differ only by the transition of a component **C**, then that component can be delayed to make the safety violation unreachable. The enabling condition for the delayed transition is the enabling condition of **t** combined with the priority mask condition that contains the information on the components that have higher priority than **C**. The Figure 13 shows the states **LL** with both clients in their **local** state, **AA** with both clients in the **access** state, and **AD** with client 1 in the **access** state and client 2 in the **delay** state. The following algorithm finds the components that need to be delayed, and computes the delay conditions.

- For all states **sd** that satisfy the safety rules, with a transition **td** from **ss** to **sd** do loop:
- If the transition **td** from **ss** to **sd** is not enabled by the same condition **cv** as the transition **t** then break.
- If some component transition in **t** is a delayed version of the transition by the same component in **td** then break.
- If **td** differs from by more then one component transition then break.
- Identify the component **C** that is delayed in **td** and completes the transition in **t**. This component causes the safety violation, and has to be delayed.
- Identify states **scs**, **sc** and **scd** where the component **C** makes a transition from **scs** to **sc** in **t**, and from **scs** to **scd** in **td**.
- Assign the set of components that complete their transitions in both **t** and **td** to the priority mask condition **cp**
- Assign the conjunction of the enabling condition **cv** for **t** and the priority mask **cp** to delay enabling condition **cd**.
- Generate a delayed transition from **scs** to **scd** in component **C**, enabled by the condition **cd**.

For behavior rules which are specified as regular expressions, the analysis is based on the combination of the related components and the finite state representation of the rule. The analysis then consists of checking if any of the rejecting states in the rule representation is included in some reachable global state. The way to improve the system in those cases is based on simple elimination of the offending states from the combined system description.

5 Related Work

This section shows some related approaches in the design and verification of concurrent systems. The emphasis of most of these systems is on the formal assurance that the critical system requirements are satisfied. The main distinction between GenEx and these verification systems is in the programming paradigm they use. The verification systems take a description of the system and its requirements, and verify the consistency between them. The system development using GenEx is based on the automatic generation of the system synchronization code that will guarantee its consistency with the safety requirements. GenEx and other design–oriented systems generally support a subset of properties that can be handled by the verification systems, but their use generally results in faster development.

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. PAL [YY91], [YY93] is a compositional reachability analysis tool that uses algebraic methods to reduce the state space of the problem. It allows the checking of more complex problems, but some examples can force it into searches of exponential size.

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 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 [CG95] requires additional information related to the abstract 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. The 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.

The concept of product state machines, as described in [Lim93], [Lim96] is conceptually very similar to GenEx, and the main difference between them is the scalability. The approach in Lim's system is that the global reachability graph is actually constructed and restricted to eliminate violation states, and the restricted graph is used in the execution. As discussed before, this approach does not scale to the level of realistic software systems. The situation is different in hardware based or manufacturing systems where the complexity generally has to be such that it can be completely captured. 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.

Automatic code generation is a technique that has been used in many areas of software engineering, and it greatly increases the consistency and the reliability of the generated systems. Lex and Yacc generate code that parses a given language, based on the grammar and operator priorities in the language. The code is generated for a modified version of the grammar that is obtained by eliminating the ambiguities from the original grammar according to the specified priorities. Polylith and other configuration languages automate the interfacing between application components based on the global connection descriptions. GenEx extends this automatic interfacing concept by using safety properties as rules that guide the synchronization process. The performance of the generated code is potentially very high, because our descriptive model is based on deterministic finite state machines that can be naturally mapped into efficient executable code.

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 synchronized model of computation, making them simple to analyze and generate code for. Both LUSTRE and Esterel support the verification of given system properties versus the system behavior. The main difference about GenEx is that instead of verifying that the specified system satisfies the given properties, it actually computes the necessary synchronization of the components that makes the system consistent with the given safety properties. This is a fundamental difference 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.

TRACTA is another compositional verification system with code generation support. The execution model in TRACTA uses an even stronger version of synchronization, embodied by the labelled transition system(LTS). The synchronization in LTS is based on the synchronous transitions by all components that use a given label. This synchronization is context–based, unlike simple synchronous execution of all components used in GenEx , LUSTRE and Esterel. The expressiveness of LTS makes it possible to design very elegant systems, using similar decomposition to that used in GenEx. The main drawbacks of TRACTA come from the strong synchronization provided by LTS. The LTS models are hard to design without deadlocks, and their implementation on distributed systems is of questionable efficacy. Although TRACTA reduces the analysis space using compositional verification, the complexity can be an exponential function of the number of components in the system

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 orientation towards centralization, that is useful in its domain of safety enforcement, but not really applicable to the concurrent and potentially distributed systems. The inapplicability is due to the notion of a centralized safety kernel that controls all accesses, while the concurrent and distributed systems require the maximum possible decentralization. Despite this shortcoming, the system is an example of how simple methods can solve complex problems, given the right domain.

The decomposition aspect of our method is based on some features introduced by the configuration languages such as Polylith [Pur94] and Darwin [JM]. Separation of conceptual description from the implementation is a powerful concept that provides these languages with great flexibility in porting applications between different environments, due to their support for mapping the concepts to a given physical structure. Unlike configuration languages, GenEx supports extracting behavior information to the conceptual level, where it can be used in the refinement of the system to comply with the given set of rules. While the goal of GenEx is similar to the configuration languages, to simplify the interaction between system components, the domains where they operate are clearly distinct and they can be combined in the construction of a concurrent system where GenEx would generate the synchronization code, and Polylith can provide the data communication between the components, as well as the underlying communication for the GenEx runtime kernel.

6 Proposed Research and Future Work

The goal of automatic rule–based synchronization is to integrate the independently developed components into a concurrent or distributed system whose behavior satisfies the given safety rules. The basic assumption is that each component is "delayable", i.e correctness of its execution is time–independent. Given a set of delayable components, the global behavior of the system can be integrated using local analysis and refinement, without the need for the computationally costly global analysis. This approach helps to reduce development time and, even more importantly, maintenance time for complex concurrent systems.

The application of automatic synchronization in software design provides the users with a synchronization skeleton that is created based on the given set of safety rules. The generated code also has a simple interface to link it with the data processing code, whether it is implemented manually or generated by other tools. The generated code is very efficient because it consists of a single table lookup for every component in the system. Even more importantly, the performance is predictable because the overhead of the table lookup can be accurately estimated.

The remaining work for the completion of the GenEx automatic synchronization system involves the completion, integration and assessment of achieved results. The first priority is the completion of the missing functional units for the system, such as the static safety violation prevention, and the global deadlock detection and removal algorithm. Another aspect of system completion is the necessary optimization of some computationally intensive functional units. The optimization is necessary to improve the performance of the analysis tools that are currently written in prolog, and therefore very inefficient, and the optimization strategies range from reordering data fields within prolog predicates to rewriting the analysis tool in a compiled programming language such as C.

The next task is the integration of all these parts into a functional prototype that implements the automatic synchronization method. The integration of the system requires a front–end interface that analyzes the safety rules and generates a script with the necessary sequence of automatic synchronization operations. Other integration aspects include the interfaces to temporal logic verification tools such as SMV [McM93].

6.1 Static Violation Detection

The automatic synchronization based on the construction of combined reachability graphs has one important drawback. The computational complexity of the graph can exceed the available processing capacity or, the time required to compute the synchronization changes can become too large to be acceptable in practice. Both of these problems are direct results of the combinatorial nature of the global system behavior, and they can be alleviated by different analysis approaches, but it is improbable that they can be solved completely.

An alternative to combined reachability graph construction is the static analysis method that is based on the detection of transition combinations that can lead to safety violations. Every potential safety violation is identified, and all possible delayed versions of the transition are added to the system as alternatives that preserve the safety. This approach has the advantage that it only requires the construction of a single–layer precedence graph for the violation states, as opposed to the whole combined reachability graph. Since the reachability graph is not constructed, some of the detected violations may be unreachable, and their delayed versions may never be executed. The addition of transitions that will never be taken does not change the behavior of the system, and only imposes some run-time overhead in the transition selection phase. This overhead should be an acceptable tradeoff to preserve the automatic synchronization capability when the complexity of the exhaustive reachability graph approach is unacceptably high. It is specially useful in the early design phases, when changes in the system occur more often, and the performance requirements are less strict. Multi–layered violation precedence graphs can be used to detect more unreachable transitions, and eliminate them from consideration.

The single–layer violation precedence graph has a lower complexity than the reachability graph, but its size is still an exponential function. Instead of being a function of the number of states per component, the complexity of the violation precedence graph is a function of the number of transitions that may lead to a violation in each component. This should result in a great complexity reduction when complex components and safety properties[8] are involved.

[8] Complex here refers to the representations of the components and rules having larger number of states. For example, invariant properties are considered simple, and their representation consists of only two states, one of which is a rejecting state. On the other hand, complex properties could be those with 2 or more non–rejecting states, because each such property at least doubles the complexity of the reachability graph. Priority, fifo access, sequence scheduling are examples of such complex properties.

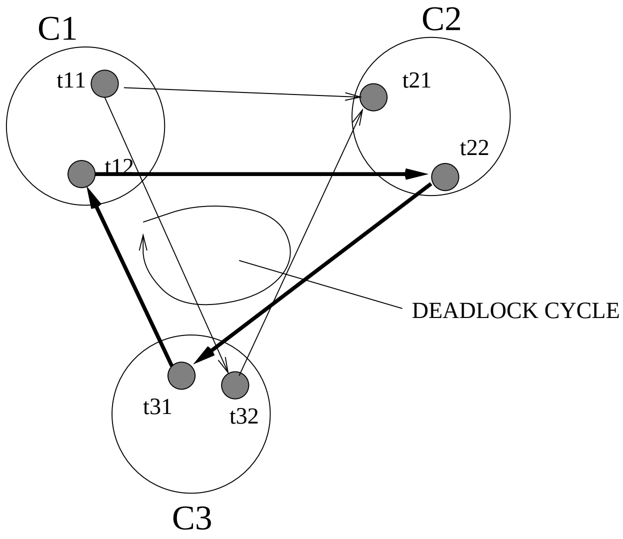

Figure 14: Blocking graph cycle detection of deadlocks. The directed edges connect a blocked component with a component that is blocking it.

6.2 Static Deadlock Detection

A well known effect of synchronization by blocking processes is the appearance of deadlocks. While mutual exclusion, and priority violations that the user specifies are domain dependent safety violations, the deadlock is a systemic fault that appears as a potential result of the synchronization when some components are required to wait for events in the system that allow them to continue the execution. Since deadlocks are an implicit byproduct of the blocking synchronization, they have to be identified and removed from the system just as the user–specified safety violations are.

Deadlocks can be a result of the synchronization for a single safety rule, or of multiple safety rules defined on overlapping sets of components. If a single safety rule produces the deadlock, and the rule is satisfied using the combined reachability graph, the deadlocked state is easy to detect and remove like other safety violations in the graph. In the case of static analysis or when multiple rules lead to a deadlock, a cycle detection algorithm is used to identify the deadlocked components and states. The delayed transitions of all components make the nodes of the deadlock detection graph, and the edges are directed between transitions that may block one another. Figure 14 illustrates how a cyclic blocking may exist between components of a system, and how it differs from acyclic blocking that exists between correctly synchronized components.

While the deadlocked states in reachability graphs can be removed immediately, the deadlocks which are identified by the cycle method should not be treated as new safety violations. This is because the synchronization for the additional rules would simply generate new deadlocks, and the complexity of the analysis would increase due to the increased number of components referenced by the safety rule. The alternative approach to solving these deadlocks is the relaxation of the state correspondence for the delayed transitions. Instead of having the delayed state represent the source state, with the capability to continue the actions started in the source state, a relaxed delayed state becomes a new, do–nothing state that can not interact with any other component in any way.

The relaxation produces idle states in components, and its application should be kept to a minimum. The components to be relaxed should be chosen by the number of different deadlocks they prevent, and by the importance of their activity in the system. User should be able to direct this process by declaring the critical components or transitions that should not be relaxed.

6.3 Synchronization Case Studies

The practical applicability of the system has to be verified on different types of problems, to assess its performance, reliability and suitability for problems of realistic size. The most important use of the automatic synchronization method is in the systems that comply with potentially evolving safety requirements. The verification of the suitability of automatic synchronization for this type of system involves two factors: the resynchronization capability, and the preservation of the interface compatibility between the generated code and the user–supplied data processing code. These capabilities will be verified on the client–server example by modifying the types of safety rules and protocols required from the system.

The capability to handle concurrent systems of non–trivial complexity is the most important requirement for the practical applicability of automatic synchronization. Verification of the scalability of the automatic synchronization method will be based on concurrent systems whose complexity is a result of the number rather then the complexity of the components and synchronization rules. The client–server example with incremental increases in the number of components will be used to verify the scalability of the analysis of this class of systems. Successful synchronization of complex systems supports the claim that the synchronization complexity does not depend on the complexity of the whole system, but on the complexity of the individual components and safety rules.

Verification of the performance and reliability of the system will be based on standard synchronization problems such as the dining philosophers or cruise control. This type of synchronization problems provides a benchmark to compare the code generated by automatic synchronization with manually generated applications or with code generated by other automatic methods.

Additional case–studies will address the issues of non–finite state systems such as stacks/buffers and event counting, and other aspects of interaction that the automatic synchronization does not satisfy directly, such as real–time behavior, asynchronous execution and delays in the sharing on global state information. Even if the complexity of the analysis for these properties is unacceptably high, the automatically synchronized systems can be used to address them. The adaptations usually involve some manual changes to the architecture of the components or the global shared data, and some of that could even be automated.

The benefits of using the automatic synchronization method have to be compared to those of other methods for producing concurrent systems. I am currently aware of only two systems that consider the safety requirements in an abstract form and implement them in the executable system, The Safety Kernel concept [WK95], and the State Combination approach by Lim [Lim93]. The system will also be compared to systems that are based on the verification of user generated concurrent systems such as Esterel [BG92] or SMV [McM93].

7 Conclusion

As described in the previous sections, the process of automatic rule–based synchronization takes a set of system components and system safety requirements, and produces an integrated system consistent with the given requirements. Thanks to the embedded interfaces, the system is readily linkable with data processing code that is manually developed or produced using other software generation tools. This method allows very quick development of high reliability concurrent applications, and increases its reusability and maintainability by supporting automatic resynchronization whenever the components or the safety rules change. The synchronization process itself is organized in a way that limits the computational complexity of the analysis, and guarantees that the system synchronization will be successful even for very complex systems.

References

[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*, 1992.

[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.

[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.

[CG95] M. Chechik and J. Gannon. "Automatic Analysis of Consistency between Implementations and Requirements". Technical report CS-TR-3394, Dept. of CS, University of Maryland, College Park, January 1995. (in preparation).

[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.

[dLSA95] Rogerio de Lemos, Amer Saeed, and Tom Anderson. "Analyzing Safety Requirements for Process-Control Systems". *IEEE Software*, 12(3), May 1995.

[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.

[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.

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

[JM] Jeff Kramer Jeff Magee, Narankar Dulay. " A constructive Development Environment for Parallel and Distributed Programs ".

[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.

[Man96] Toni Mandrioli. COMPASS 1996 Keynote Address, 1996.

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

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

[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.

[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.

[YY93] Michal Young and Wei Jen Yeh. " Compositional reachability analysis of Ada Programs Using Process Algebra". July 1993.

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

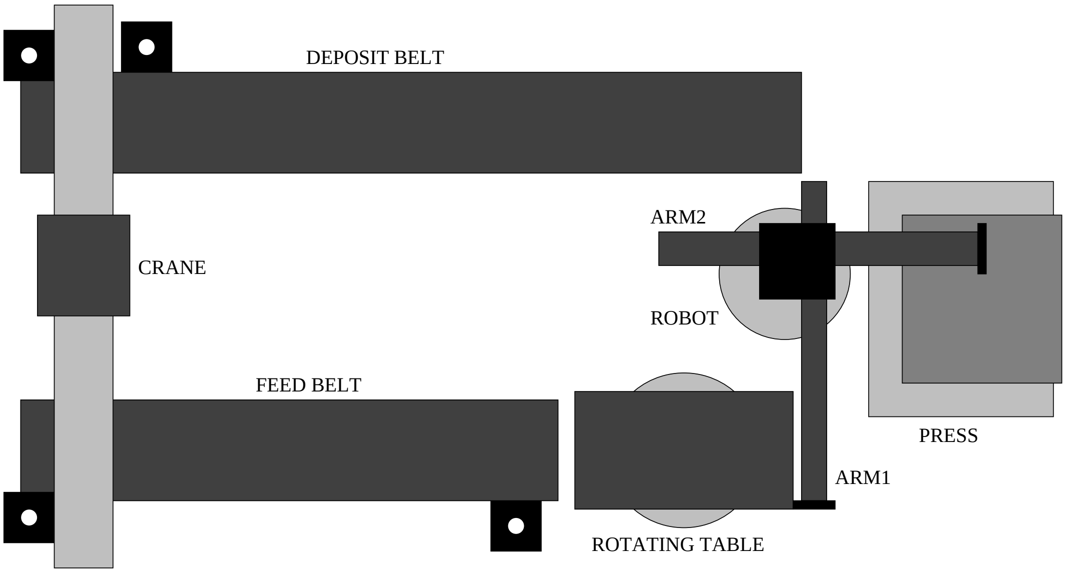

Figure 15: The Production Cell system

A Production Cell Controller

Our method automates the integration of FSM–based concurrent and distributed systems. Many other methods work with the same kind of systems, and simplify the design using a variety of integration, verification and analysis tools. To assess the quality and the potential of our method, we will analyze its effects in the development of an industrial safety critical control system, where many other methods have been used with the same goal. We will attempt to design a controller for the production cell system [LL95] used for a case study of over a dozen formal design methods for concurrent systems.

The production cell system in Figure 15 is an example of a controlled concurrent system with decomposable interaction between the components. The production cell contains five independent devices that have to synchronize with each other to achieve correct behavior. The devices are: a press that processes metal blanks, a rotating robot with two extendable hands that places blanks in the press, and takes them out after they are processed, a feeding belt that brings the blanks , a rotating table where the blanks are deposited to be picked by the robot, and a deposit belt where the robot deposits the processed blanks. A crane between the feeding and deposit belt maintains a continuous stream of blanks coming into the system.

Each component of this system has some individual restrictions on their mobility, and the interaction between them has to occur within predefined parameters to preserve the safety of the system. The safety of the system is violated by collisions between components, and by the inappropriate handling that results in blanks being dropped. The 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 transfering them.

The goal of the controller is to synchronize the machines to accept the blanks, press them and deposit the pressed blanks on the deposit belt. The component behavior is outlined by the authors, in the form of finite state machines, and we use those descriptions to create our own controller for the system. Our controller consists of the component controllers, taken from the system specification, and synchronization and safety rules that specify the restrictions on the interaction between the machines. The synchronization and safety rules are also in the form of FSM, and they monitor the behavior of the components, signalling safety violations when the components interact in undesirable ways. The integration analyzes the behavior of the components and uses the detected safety violations to delay the transitions that generate those violations. The components can complete their delayed transitions only when they no longer endanger the system safety. This way, the safety of the system is guaranteed by design, and the generated controller enforces it.

A.1 Components

The simplest components of the production cell system is the feed belt. The belt is equiped with a sensor that detects when a metal blank reaches the end of the belt, and it accepts one controlling signal specifying whether to move or stop. The behavior of the feed belt is simple: it should move until a blank is detected at the end of the belt, then it should stop and restart when the rotating table is ready to receive the blank. This behavior is formally defined in Figure 16a).

The deposit belt is similar to the feed belt, but its behavior is somewhat different because it has to wait for the crane to take the blank off the belt rather then just drop the blank. The behavior of the deposit belt is the following: it moves until a blank is detected by the sensor, and passes by it; when the sensor no longer detects a blank, it is in position to be lifted by the crane. After the crane lifts the blank, the deposit belt can restart. This behavior is defined in Figure 16b).

The rotating table can move vertically, adjusting its height to receive the blank from the feed belt, and to allow the robot to pick the blank with the magnet on its arm. The table also rotates to position the blank in the correct position to be picked up by the robot. The behavior of the rotating table follows: it starts in the low inline position, and then rises and rotates clockwise until it gets to the high and diagonal position. From the high diagonal position the table lowers and rotates counterclockwise until it gets to the low vertical position. This behavior is defined in Figure 16c).

The press moves only vertically, and has three height sensors that detect its position. The sensors can detect the high position when the press closes and processes the metal blank, the medium position when the press is ready to accept a new blank from the robot, and the low position when the processed blank can be picked up by the robot. The press receives a new blank in the medium position, and then presses it until it reaches the high position. After the blank is pressed the press opens until it reaches the low position where the blank can be picked up by the robot. Then the press closes until it reaches the medium position where a new blank can be received.This behavior is defined in Figure 16d).

The robot performs two functions in the system, its first arm carries blanks from the rotating table to the press, and its second arm takes processed blanks from the press and places them on the deposit belt. The robot can rotate as a whole, extend and retract both arms individually, and pick blanks by activating magnets on the arms. The robot also senses its rotation angle, and the extension of the arms to help it in the correct positioning. The detailed behavior of the robot is as follows: it picks a blank from the table, then rotates and extends arm two to pick a processed blank from the press. Next, it rotates counterclockwise and retracts the second arm to drop the processed blank on the deposit belt and then rotates more to drop the new blank from first arm onto the press. After that it retracts both arms and rotates clockwise until the first arm points to the rotating table and extends it to pick a new blank.

The crane takes blanks form the deposit belt and places them on the feed belt to maintain continuos operation. The crane can move horizontally between the two belts, and vertically to adjust its height to the level of the belts. The crane is equiped with three sensors, two signal when the crane is in the correct horizontal position over the belts, and the third provides the information on the crane elevation. The crane moves to the deposit belt, and lowers to its level to pick a processed blank. Then the crane lifts the blank, and travels to the feed belt where the blank is lowered and dropped when the crane magnet is turned off.

The common denominator of all described components is that they are all independent from each other. Each component is described only in terms of its position without regard to the state of other components. But it is clear from the description of the system that these components have to be synchronized to pass the metal blanks. The rotating table has to wait for the blank from the feed belt before starting to raise and rotate, and the feed belt has to wait for the rotating table to come to the low inline position before starting to unload the blank. This informal description of the desired interaction between the components is easily formalized into a safety rule that requires that behavior.

The safety rule **FEED_TABLE** in Figure 17a) specifies the interactions between the feed belt and the rotating table that should be made unreachable in the executable version of the controller. The safety violations are defined as transitions to the state *reject* of the safety rule. The safety rule prohibits the feed belt from entering the *UNLOADING* state

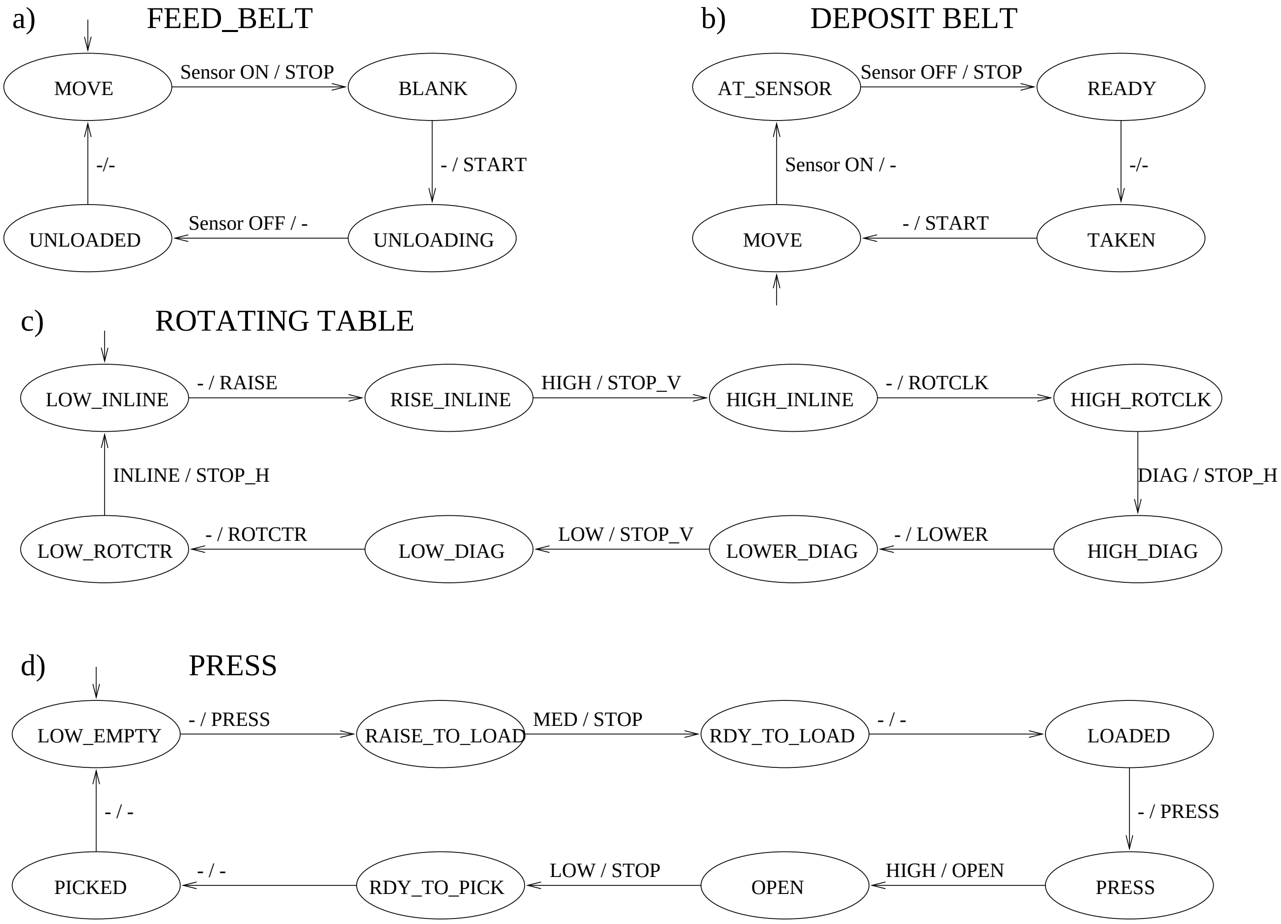

Figure 16: Components of the Production Cell system

until the rotating table gets to the *LOW_INLINE* state where it can accept a new blank. Once the table is in the state where it can accept the new blank, it is not allowed to start rising until the feed belt unloads a blank on it. After the new blank is passed to the rotating table, it is allowed to start rising and the safety rule prohibits the feed belt from unloading until the rotating table returns to the low inline position.

This safety rule enforces what is basically a handshaking algorithm for the feed belt and the rotating table. It requires a specific interleaving of transitions by the two components, insuring that the feed belt unloads the metal blank on the rotating table. Although the components are defined independently, this rule references both of them, and enables the integration process 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 safety rule **ARM1_TABLE** in Figure 17b) specifies the interaction between the robot

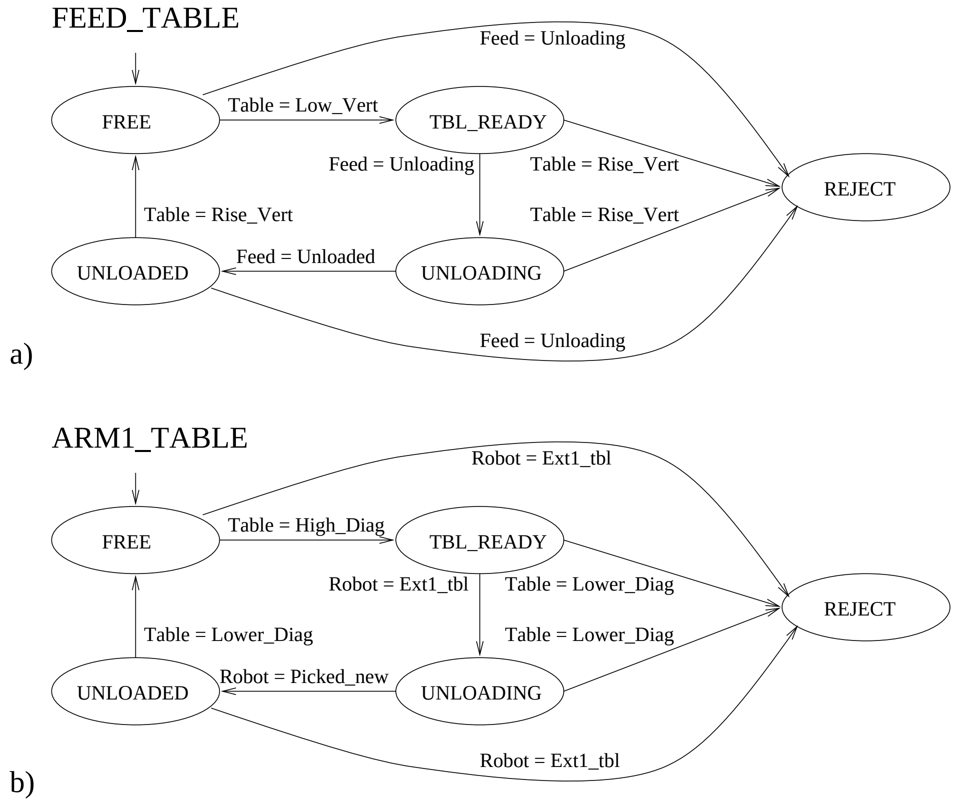

Figure 17: Some safety rules for the Production Cell

and the rotating table when the first robot arm picks a new blank from the table. This safety rule requires the robot to wait for the rotating table to get to the high diagonal position before the robot starts extending its arm to pick the blank, and requires the rotating table to remain in the high diagonal position until the robot succeeds in picking the blank. The two FSMs are isomorphic, and the same applies to all other safety rules for this system. In every case one component has to wait for the other to complete some action that transfers the metal blank between them. The simplicity and uniformity of the safety rules makes this system description very compact and intuitive. The independence between the components simplifies design and maintenance and promotes component reuse for similar systems.

A.2 System Integration

The integration of our production cell controller starts with a safety analysis of the component behavior.

The integration of our production cell controller consists of several phases where the components and safety rules are analyzed, modified and finally combined into an executable model. The analysis phase tries to identify all possible ways the safety rules might be violated by the components, and these violations are memorized. For every safety violation one or more components are identified for causing it; these components will be delayed to preserve the safety. The delayed transitions substitute the original transitions whenever the preconditions for the safety violation are satisfied. The components are modified to include the delayed transitions in their description, and the code is generated from the modified components and safety rule descriptions.

We will show the main phases of this process in more detail on the analysis and integration of the feed belt and the rotating table using the safety rule **FEED_TABLE**. This safety rule references only the feed belt and rotating table, and thus only these components can lead it to safety violations. We limit the analysis to those components and thus significantly reduce its complexity. The highest possible complexity of the combined behavior of these two components and their safety rule is the product of the numbers of their states. In this case that is $4 * 8 * 5 = 160$ states, and that complexity is easy to analyze automatically. Since only one state per component can cause a safety violation, there are 4 possible violations: When the table starts to rise while the safety rule is in the state *TBL_READY* or *UNLOADING*, or when the feed belt starts unloading while the safety rule is in the state *FREE* or *UNLOADED*. We can use these potential safety violations for component delays without analyzing their reachability, thus reducing the complexity even further.

The first phase of the integration is the addition of delayed states and the creation of a shared global variable pool where the system state data resides. The delayed states are the states where the components remain while their transitions are delayed, and one state is necessary for every delayed transition. In this system only transitions from *BLANK* to *UNLOADING* in the feed belt, and from *LOW_INLINE* to *RISE_INLINE* may need to be delayed to satisfy the **FEED_TABLE** safety rule, because only those transitions lead to a state that may violate the safety [9]. After adding the delayed states to the components, we generate the shared state variables. These consist of: the environment variables, both the monitored (sensor inputs) and controlled ones (control signals), the component and safety rule state variables, and the nondeterministic priority variables. The state variables encode the global state of the system, and they provide the information on the violation preconditions for the detected safety violations. The priority variables have a purpose when resource utilization is specified using safety rules, but in this system all safety rules require only action sequencing and the priority variables are redundant.

The analysis of the reachability graph for this safety rule requires a graph with 100 states, and detects the safety violations described earier. The safety violations occur when one of

[9]Other delayed states are added to these components because other transitions may violate other safety rules, but these are the only ones necessary for the enforcement of the **FEED_TABLE** rule.

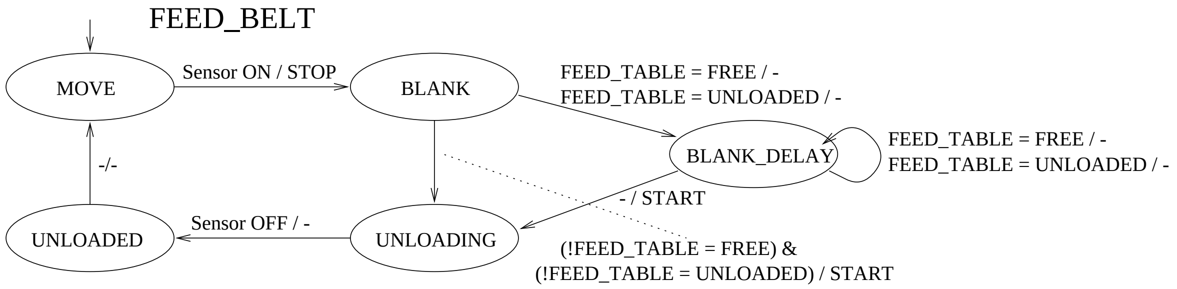

Figure 18: The modified feed belt component

the following preconditions is satisfied:

TABLE = LOW_INLINE & FEED_TABLE = TBL_READY
TABLE = LOW_INLINE & FEED_TABLE = UNLOADING
FEED = BLANK & FEED_TABLE = FREE
FEED = BLANK & FEED_TABLE = UNLOADED

The feed table causes the safety violation when it rises while the **FEED_TABLE** is in states *TBL_READY* or *UNLOADING*. That means that the transition from *LOW_INLINE* to *RISE_INLINE* has to be delayed until a blank is unloaded. The enabling conditions for the delayed transitions are taken directly from the violation preconditions. The same conditions that select the delayed transition, also enable the looping transition in the delayed state, and prevent the component from completing the transition before the safety preservation is guaranteed. The transitions from *LOW_INLINE* to *LOW_INLINE_DELAY* are enabled when one of the following conditions holds:

FEED_TABLE = TBL_READY
FEED_TABLE = UNLOADING

The delayed transitions are given a higher priority then the original transitions, thus preventing the original transitions from being enabled on a subset of its original enabling conditions. The subset of the original enabling condition where the delayed transition is selected corresponds to the intersection of the enabling condition and the safety violation precondition. The same process will generate delayed transitions from *BLANK* and looping transitions in the delayed state *BLANK_DELAY* for the feed belt with the following enabling conditions:

FEED_TABLE = FREE
FEED_TABLE = UNLOADED

The resulting component **FEED_BELT** is in Figure 18. The delayed state *BLANK_DELAY* is reachable when the safety rule **FEED_TABLE** is in the states *FREE* or *UNLOADED*, and the same conditions keep the component in the same state. The original transition from *BLANK* to *UNLOADING* is enabled only when the delayed transition is not, so this coomponent can no longer cause safety violations for the safety rule **FEED_TABLE**.

This process is repeated for every safety rule and its referenced components, and results in a set of delayed transitions enabled by the safety violation preconditions. The delayed transtions for a given component are combined without interference because they all apply to single states. After analyzing all safety rules and adding delayed transitions to all components that need them, the controller is constructed by combining the components and the safety monitors derived from the safety rules. The safety monitors model the behavior of the safety rules used in the analysis, and provide the components with the information on their state needed to evaluate the enabling conditions for the delayed transitions.

A.3 System Verification

The requirements for this system include a number of safety properties and one liveness property. The main safety properties define the limits of machine mobility, the machine collisions and the conditions when the metal blanks are dropped outside safe areas. The machine mobility properties specify the safe ranges for the operation of individual machines, that may be damaged if the limits are not respected. Since these properties are individual for each machine, the controller components implement them directly. An example is the rotating table component in Figure 16c) that monitors its height and rotation angle and stops immediately when the table reaches the desired positions within the safe limits. It is obvious from the FSM that the rotating table respects the machine mobility requirements in its original form.

Since the components are modified by the addition of delayed transitions, it is conceivable that some transition whose effect is to stop the machine could be delayed, thus leaving the machine to move without control. The mobility limits are preserved if the transitions that stop the machines when they reach a desired position are never delayed. This condition holds in our system because the safety rules only reject the states when the machines initiate a new movement. This is an informal proof of preservation of machine mobility limits. The same properties can be proved formally using the symbolic model checker SMV, on a partial model of the controller generated by GneEx. To verify the mobility limits safety, the partial model only needs to include the component itself, and the complexity of the analysis is limited to the size of its FSM. The machine mobility limits are represented by CTL formulae that specify that the machine stops immediately when it reaches its destination position. The properties that verify the limits on the table rotation and raising are given below:

```
AG(((tbl_high_pos = 1) & (tbl_raise = 1)) -> AX(tbl_raise = 0))
AG(((tbl_diag_pos = 1) & (tbl_rotclk = 1)) -> AX(tbl_rotclk = 0))
```

There are various reasons for machine collisions, one of them being the violations of mobility limits. Namely, if the rotating table rotates counterclockwise past the position in line with the feed belt, it will collide with the belt. Other collisions may occur if the robot arms are too extended during rotation or if the crane is too low during horizontal movement. These components are designed to avoid those collisions by retracting the robot arms and lifting the crane during movement, according to the constants given in the specifications. Another cause of collisions is wrong synchronization when the blanks are being passed between machines. These collisions are prevented by the safety rules integrated with the system. The safety rules only allow the components to approach when their states are compatible, and collisions are impossible.

A related type of problem is the collision between blanks, it happens when a new blank is unloaded on the rotating table or press while the previous one is still there. This type of collision is also prevented by the safety rules integrated in the controller, because they require both components to exit the state where the transfer is possible before allowing them to initiate another transfer. When the components leave a state where they receive a blank, they have to go through a state where the robot picks the blank before being able to accept another blank. This shows how the handshaking algorithm implemented by the safety rules forces the components to synchronize for the transfer of blanks, and leave those transfer states to allow the transfer of blanks to other machines.

The last type of safety requirement in the production cell system are the rules governing where the metal blanks may be dropped. These rules specify that the feed belt may only unload a blank to the rotating table in its low position and in line with the feed belt, or that the robot may only drop a new blank on the press, meaning that its first arm points to it and is sufficiently extended. These rules are also enforced by the integrated safety rules that synchronize the behavior of the components. Some of these rules are given below using CTL notation for invariant properties.

```
  Table in low position when feed belt is unloading
AG((feed = unloading) -> (tbl_pos_low = 1))

  Arm pointing to press and extended when dropping new blank
AG((robot =  drop_new) -> ((rbt_one_to_prs = 1) & (rbt_one_over_prs = 1)))

  Table in high diagonal position when robot picks the new blank
AG((robot = pick_one) -> ((tbl_pos_high = 1) & (tbl_pos_diag = 1)))
```

A.4 Executable Production Cell Controller

The integrated controller model contains all necessary functionality to control the production cell system, and GenEx derives its executable model using the C programming language. The generated code contains the representation of each component and safety monitor, and is machine and architecture independent. Depending on the runtime support used it can be executed on a single processor or a distributed network. The following gives a flawor of the generated code for the feed_belt and its state *BLANK*.

```
SMG_feed_belt()
{
  newstate= -1;
  trans=0;
if(currentstate[_MD_feed_belt]==_ST_feed_belt__in__f_nothing)
        {SMG_feed_belt__in__f_nothing(-1);}
if(currentstate[_MD_feed_belt]==_ST_feed_belt__in__f_plate)
        {SMG_feed_belt__in__f_plate(-1);}
if(currentstate[_MD_feed_belt]==_ST_feed_belt__f_plate__f_unload__delay__1)
        {SMG_feed_belt__f_plate__f_unload__delay__1(-1);}
if(currentstate[_MD_feed_belt]==_ST_feed_belt__in__f_unload)
        {SMG_feed_belt__in__f_unload(-1);}
if(currentstate[_MD_feed_belt]==_ST_feed_belt__in__f_move)
        {SMG_feed_belt__in__f_move(-1);}
if(newstate!= -1){currentstate[_MD_feed_belt]=newstate; change=1; }
}

SMG_feed_belt__in__f_plate(num)
int num;
{
 if((sig1[_SG_tbl_inline_pos]==0)&&(sig1[_SG_feed_belt__f_blank]==1)&&
    (sig1[_SG_rot_table__rotctr]==1)&&(sig1[_SG_feed_table__ft_nothing]==1)&&
        ((num==0)||((num== -1)&&(newstate== -1))||(num==1)))
  {newstate=_ST_feed_belt__f_blank__f_unload__delay__1; SMG_action__80(); }
         }
 else
 if(
        ((num==0)||((num== -1)&&(newstate== -1))||(num==22)))
  {newstate=_ST_feed_belt__in__f_unload; SMG_action__81(); }
         }
}
```

The components are executed in parallel, and their results are propagated to the shared state variables. The safety monitors are executed in the second phase, using the new component state information. This makes the code match the simulation structure used in the safety analysis and integration. This structure speeds the detection of safety violations in the analysis and provides precise preconditions for synchronization. The code that integrates the execution of components is given below.

```
dotransitions1()
{
 SMG_robot();
 SMG_press();
 SMG_crane();
 SMG_dep_belt();
 SMG_feed_belt();
 SMG_rot_table();
 getsigs();
}

dotransitions2()
{

 SMG_robot1_press();
 SMG_robot2_press();
 SMG_robot_belt();
 SMG_robot_table();
 SMG_dep_crane();
 SMG_feed_crane();
 SMG_feed_table();
 propagation();
 getsigs();
}

inputsignals()
{
input_signals_fn();
}

outputsignals()
{
output_signals_fn();
}
```

The *inputsignals* and *outputsignals* functions are used to communicate with the environment, receiving monitored variables and sending control signals to the machines. The *inputsignals* function is called before the components are executed in every phase, and the *outputsignals* at the end of the phase when all new signals are computed. The user supplies these body for these functions, and it is preferably in the form of external functions that don't have to be typed in whenever the code is regenerated. Parts of the user supplied functions are given below.

```
input_signals_fn(){
  int inum[15];
  float num[15];
    scanf("%d",&inum[0]);
    if(inum[0] == 1)        sig1[_SG_press_low_pos] = 1;
      else                  sig1[_SG_press_low_pos] = 0;
    scanf("%d",&inum[1]);
    if(inum[1] == 1)        sig1[_SG_press_mid_pos] = 1;
      else                  sig1[_SG_press_mid_pos] = 0;
    scanf("%d",&inum[2]);
    if(inum[2] == 1)        sig1[_SG_press_up_pos] = 1;
      else                  sig1[_SG_press_up_pos] = 0;
    scanf("%f",&num[3]);
    if(num[3] >= 0.5208)    sig1[_SG_rbt_ext_to_tbl] = 1;
      else                  sig1[_SG_rbt_ext_to_tbl] = 0;
    if(num[3] >= 0.6458)    sig1[_SG_rbt_one_over_prs] = 1;
      else                  sig1[_SG_rbt_one_over_prs] = 0;
    if(num[3] <= 0.3708)    sig1[_SG_rbt_one_retracted] = 1;
      else                  sig1[_SG_rbt_one_retracted] = 0;
}

output_signals_fn(){
  if (sig1[_SG_tbl_raise] == 1)       printf("table_upward\n");
  if (sig1[_SG_tbl_lower] == 1)       printf("table_downward\n");
  if ((sig1[_SG_tbl_raise] == 0) && (sig1[_SG_tbl_lower] == 0))
                                      printf("table_stop_v\n");
  if (sig1[_SG_tbl_rot_clk] == 1)     printf("table_right\n");
  if (sig1[_SG_tbl_rot_ctr] == 1)     printf("table_left\n");
  if ((sig1[_SG_tbl_rot_clk] == 0) && (sig1[_SG_tbl_rot_ctr] == 0))
                                      printf("table_stop_h\n");
  if (sig1[_SG_feed_move] == 1)       printf("belt1_start\n");
    else                              printf("belt1_stop\n");
}
```

These functions work with the standard input and standard output, because that is the interface that the production cell simulator uses. A different interface to the simulator or system would be simple to design using appropriate control or message passing mechanism instead of the *scanf* and *printf* function calls.

The informal verification of system functioning requires a simulated or live execution, and we did that using the production cell simulator produced by the study authors. The simulated execution controlled a production cell system with three blanks simultaneously in the system.

A.5 Conclusion

This example shows that using safety rules to specify the behavior of concurrent and distributed systems can significantly simplify the system integration process. Our specification uses very simple specification for both components and interaction rules, and produces a correct and verifiable controller. The intuitive nature and simplicity of the components and synchronization rules makes them easily reusable in other similar systems. The capabilities of our controller are comparable to those produced using other systems, and it handles the maximum number of blanks that the simulator supports simultaneously in the system. The time required to design our system is in line with the best of the systems surveyed in the study.

The formal nature of the integration process guarantees that the produced system will enforce the desired safety properties. The integrated model generation facility produces partial models where the correctness of the system can be independently verified. Finally, the generated code implements the verified behavior as a safe and reliable executable system.