Automated Enforcement of Receptive Safety Properties in Distributed Design*

Gilberto Matos
Siemens Corporate Research
Princeton,New Jersey
matos@scr.siemens.com
research done while at UMD

James Purtilo
University of Maryland
College Park, Maryland
purtilo@cs.umd.edu

Elizabeth White
George Mason University
Fairfax, Virginia
white@cs.gmu.edu

Abstract

Independent development of system components may cause integration problems if their interaction is faulty. This problem may be solved by enforcing required component interactions at the system level. We have developed a system that automatically integrates control–oriented components, to make them consistent with aggregate system behavior requirements. Our method is based on the automated synchronization method that modifies independently designed components to make them satisfy a set of user defined receptive safety properties. The automated synchronization allows us to design the components as independent controllers that satisfy their individual requirements and to compose a correct executable system by combining the components and enforcing their interaction constraints. This approach gives component designers the freedom to design independently, and produce a functional system by combining the components and specifying their interaction requirements.

Keywords: *receptive safety properties, automated synchronization, concurrent software, automated code generation, reliability.*

1 Introduction

Complexity of modern software systems often demands the decomposition of systems into components. These components may be designed separately and even independently, by geographically remote design teams. Development of complex systems is a difficult task, requiring the correct functioning of all components as well as correctness of their interaction in an integrated system. While the individual functionality of the components may be specified locally, the interaction is global and makes the components interdependent. This makes the system integration a critical and potentially expensive phase where the correctness of interaction must be assured.

In previous work, we developed a method for automated synchronization of concurrent systems that satisfy given sets of receptive safety properties[5, 6]. Systems are designed using independent components that only need to satisfy their functional requirements. The component interaction is specified using receptive safety properties, and the components can be modified to enforce system behavior that satisfies these properties. Figure 1 illustrates the process of creating a synchronized system from independent components and receptive safety properties that constraint the component interaction. The components are designed independently, without explicitely implementing their interaction. The specified safety properties are used to analyze the synchronization requirements of the system, and to produce modified components that interract correctly with respect to the given safety rules.

In this paper, we describe how this same method can be used to distribute the design of complex systems to independent, and possibly geographically remote teams. The teams can implement the individual components concentrating exclusively on their functional requirements, and the system can be integrated by producing automatically synchronized versions of the components. This approach is particularly suited for designing systems where the concurrent components are control dependent, meaning that their execution must be interleaved. The guarantee that the synchronized components will interact as specified simplifies the partition of the system, and the specification of the interfaces between components.

The format of this paper is as follows. In Section 2, we briefly describe our model of process control systems and our method for automated synchronization of the components of this type of system. We also define the receptive safety properties, and the implications of their automated enforcement. In section 3, we give an example system with significant component functionality, and with typically simple control interaction. We show how this system can be integrated using automated synchronization of independently specified components. We conclude by pointing out the advantages of this method compared to manual component synchronization.

*This research is Supported by the *Office of Naval Research* under contract ONR N000149410320 and by the *National Science Foundation* under contract CCR-96-25-202

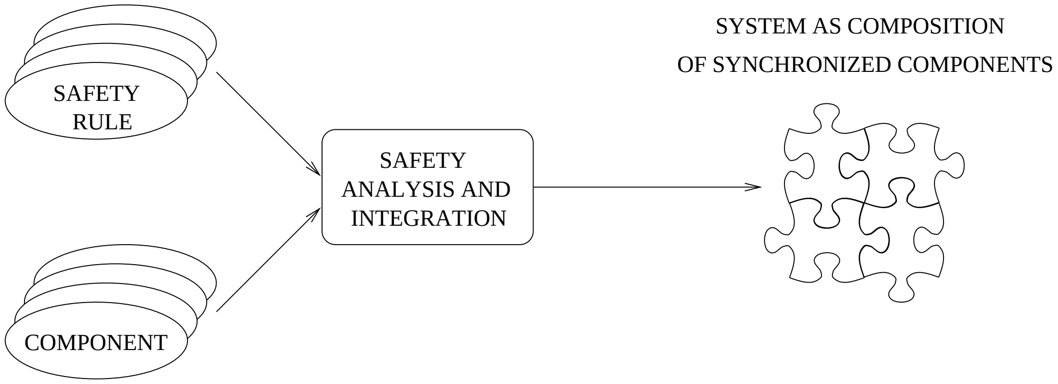

Figure 1: System integration for Safety Properties

2 Receptive Properties and Realizability

The behavior of a system is more complex then just the parallel execution of its components, and this additional complexity is a result of interaction. The interaction between components may occur in different ways, including data communication, control dependencies (RPCs for example) and synchronization mechanisms (such as locks and barriers). In systems that interact with the environment, components may also interact indirectly through environment reactions. The interaction between the components determines how the system as a whole satisfies its requirements. The consequences of incorrect interactions range from loss of performance to total and catastrophic system failures, while correct interaction is required in reliable and high performance systems. The correctness of interaction is application specific; while some interaction patterns may be correct in one system, they may cause failures in others.

The desired system behavior can be described by a set of safety and liveness properties that specify the aggregate behavior. A formal definition for both types of properties relates the safety and liveness properties to sets of acceptable executions. The fundamental difference between safety and liveness properties is in the finiteness of execution traces where those properties can be verified. If all violations of a given property are detectable on finite traces, that is a safety property, while the liveness properties accept every finite execution and may reject some infinite execution traces. This definition of safety is much broader than the notion of physical safety, and includes properties such as real–time behavior, fault tolerance [7], and data range constraints.

Safety violations are always caused by the occurrence of some violating event. Safety properties whose violations can only be caused by controlled actions have been studied by Abadi and Lamport [1]. They named these properties receptive based on an earlier definition of receptiveness of trace structures [3]. They also proved that receptive safety properties play an important role in the realizability and compositionality of complex systems. Receptiveness provides a tool for distinguishing between realizable and unrealizable safety properties. Any realizable safety property can be represented by a stronger receptive safety property; A system that satisfies a receptive safety property also satisfies all weaker safety properties implied by it.

Software development is increasingly reliant on component based technologies. Research in concurrent systems is often based on finite state components executing in parallel [4, 2]. This view of components is especially useful for embedded control systems where the main criterion for correctness is whether the system completes certain actions in the right order or at the appropriate time. We use this finite state machine (FSM) model for both the system components and the safety properties. Just as components can be modeled by finite state machines, many important system properties can be captured using similar notations. In our system the safety properties are defined as FSMs that react to component states, and signalize safety violations by reaching a predefined $REJECT$ state.

Violations of receptive safety properties are by definition a result of a controller action. By delaying the violating controller actions, we can prevent the occurrence of states that violate the specified receptive safety properties. In our execution model, the system controller comprises a number of concurrent components, and the controller actions correspond to specific combinations of component actions. By delaying the components whose actions cause the safety violation, the violation itself is delayed. While the components with potential for violations are delayed, others are allowed to proceed with their execution until the system eventually reaches a state where the delays are no longer necessary

2.1 Enforcement of Receptive Safety Properties

The synchronization mechanism in GenEx is based on the delaying of component transitions. Delayed transitions are implemented by introducing an additional state where the component FSM blocks as long as the completion of its transition may cause a safety violation. Figure 2(b) illustrates the implementation

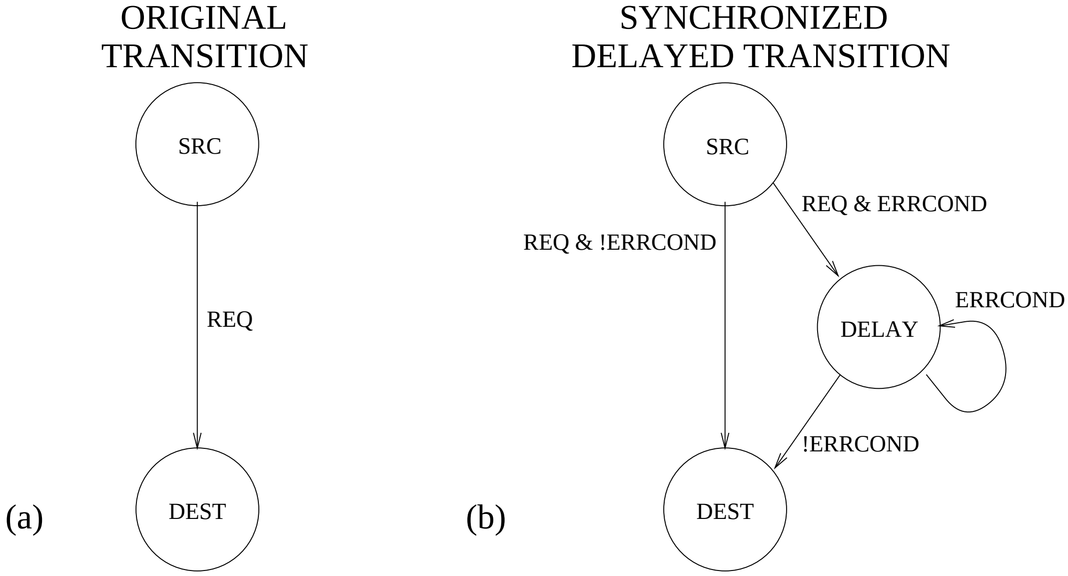

Figure 2: Overview of delayed transition implementation

of a delayed transition for the transition in Figure 2(a). If the safety analysis finds that the transition could lead to a violation, the delayed transition is added to block the component whenever the safety preconditions hold. The enabling condition of the original transition **REQ** is combined with a set of conditions **ERRCOND** that is the precondition of the safety violation, and the resulting conditions enable the delayed transition. The original transition will be allowed to proceed only when its enabling condition **REQ** is satisfied and the safety violation precondition **ERRCOND** is not. The transition from the delayed to the destination state will occur only when there is no potential for a safety violation.

GenEx uses this method to delay all component transitions that can cause a violation of some receptive safety property. The receptive safety properties describe the unacceptable system behaviors, and the receptiveness guarantees that some controlled component causes every violation. GenEx statically analyzes the safety rules and the components to find the components and their transitions that may cause the violations. All transitions that may cause safety violations are prevented using the delayed transition mechanism with the safety violation preconditions as enabling conditions.

2.2 Software Development with Automated Control Integration

Integration of complex systems from independently developed components can be a frustrating process if the components interaction is faulty. If component interaction can be corrected during system integration, by synchronizing their behavior and making it consistent with specified receptive safety rules, the component development becomes more independent. We have shown [5, 6] how automated control integration allows faster integration of complex systems with strong assurance that the final product satisfies all functional and safety requirements.

Our method supports a software development framework where the data dependencies are defined early in the project, but the control interactions can be left for the integration phases. During the development of concurrent components, the designers must cooperate on a smaller set of common interfaces, namely the data processing ones. This allows the separation of data processing and control interaction for different phases of system development. This separation fits well with the majority of modern programming languages where the data hierarchies are defined very early in the development process, while the control aspect of every component is largely independent.

By allowing the component behaviors to be synchronized late in the design process, in the integration phase, we give more independence to the designers in developing their respective components. The only restriction is that the components should not impose conflicting requirements on the ordering of system actions, because such components can never be integrated into a functional system. A simple guideline for component development is that they should not impose serialization on events unless the sequencing is required for their functionality. The serialization requirements of individual components should be verified early, only to ensure that no conflicts exist.

In the next section, we describe the reengineering of the control in an existing application. After describing the application itself, we briefly describe how the initial control integration was accomplished and then how we applied the techniques described above.

3 The AEGIS Tracking System

The AEGIS Weapons Systems, consisting of an array of sensors and weapons, is designed to defend a battle group against air, surface and subsurface threats. The system is responsible for tracking and categorizing objects in the system with respect to weapons doctrines, engageability regions and fixed regions. Parts of the system can be automated or semi-automated, resulting in a system with a great deal of complexity.

In 1993, the University of Maryland software interconnection lab prototyped a small part of the AEGIS system as part of a larger exercise to demonstrate various prototyping languages and techniques [8]. The difficult algorithms in the prototype are not dependant on synchronization and so could be implemented in a single component; however this component must be integrated with the rest of the system including information gathering, data, and graphical presentation components.

The achitecture of the system was based on a virtual shared memory module, implemented as a single component that communicated with the functional components. The functional components of the system included a loader, a spreadsheet, the tracker module, a display unit and a list unit. The initialization of the system was performed by the loader which required exclusive access to the shared memory module. After the loader completed its function, the spreadsheet and tracker were allowed to start executing and accessing the shared memory. The spreadsheet executed independently from the other components and had no further synchronization requirements. The spreadsheet acted as a producer of data, and all of its updates were atomic. The tracker component used the data produced by the spreadsheet, and computed the parameters to be used by the display and list components. The display and list only functioned when data was available from the tracker, so they had to synchronize with the tracker and access the shared memory after the tracker's accesses. These functional components interacted indirectly via ordered accesses to the shared memory module. In the initial implementation this ordering was built explicitly into the component implementation structure as extra read/write statements. For example, the tracker and spreadsheet functional components remain blocked on a read statement until the loader sends each a message telling them to proceed.

In the next section, we describe another way to build this system by enforcing control requirements automatically. When the control interfaces are enforced at the system level, designers can produce simpler functional components since they have less stringent behavior specifications. Assuming the components and interaction specifications are designed correctly, an executable implementation that satisfies all requirements can be automatically generated by synchronizing the system components.

4 System Specification

The AEGIS system is specified as a set of finite state components and receptive safety rules that describe their interactions. The control aspect of component behavior is almost trivial, as shown by the **tracker** component FSM description in Figure 3a). This component consists of three states, and the transitions between them are unconditional. This models a greedy component that attempts to process data as fast as it can, using all available processing resources. The control behavior of other components is very similar to the **tracker** and with similar complexity. Each component has an initialization, and an active and passive state. The active state is when the component is executing its function, and the idle state is when it is done, or waiting to be allowed to activate again.

We consider the components to have a finite state control behavior that roughly corresponds to the control structure of the component implementation in a sequential programming language. Part of this control structure is unrelated with the interaction between the component and the rest of the system, so it can be abstracted away in the representation whose goal is system synchronization. The abstracted part of of the component can be assumed to implement its data processing functionality. In the case of the **tracker** component, most of its functionality is executed as a part of the transition from *TR_IDLE* to *TR_ACTIVE*.

The data reading/writing requests are issued by the components as part of their actions. Actions represent the effect of specific component transition, and only the execution of the appropriate transition invokes the corresponding action. The actions are controlled by the component behavior, and the delayable nature of component transitions means their execution can be synchronized to prevent dangerous action sequences. Our synchronization method will analyze the interaction between the components, and modify the components control structure to include the necessary delays.

The global receptive safety rules are defined based on the system interaction requirements, derived from acceptable component interactions. A receptive safety rule for this system is shown in Figure 3b), and it specifies the valid initialization sequence for the **loader**, **spreadsheet** and the **tracker**. This safety property is receptive because its violations can only be caused by the actions of the controlled components. The **spreadsheet** and **tracker** will cause a safety violation if they access the shared memory before the loader sets its initial state. The **INIT_SEQUENCE** safety property specifies this as a violation, easily detectable by system behavior analysis. The components will be modified by adding conditional delays to their transitions to the active state, making them wait for the completion of system initialization. The lack of explicit delays in the components makes them easy to use in different component configurations, with other types of interaction constraints.

Automated synchronization using GenEx produces an integrated system where the components consult the safety rules' state data in determining their enabled transitions. The synchronization of the components works independently of the runtime organization of the systems. The same generated code can, depending on the runtime support library, execute in a sin-

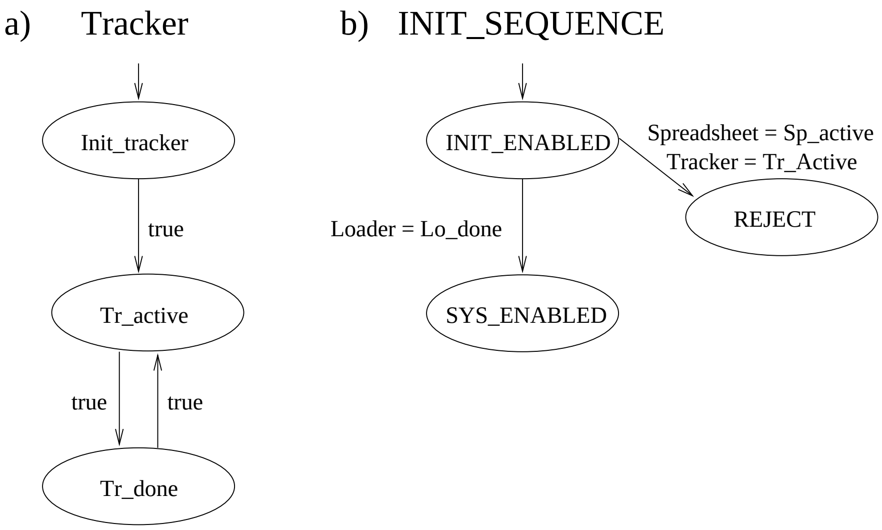

Figure 3: A component and a receptive safety rule for the AEGIS system

gle process form or as a collection of distributed processes comprising one or more components and safety rules. This flexibility makes the automatically generated aegis system portable to a variety of environments including those without support for multiprocess execution required by the original manual implementation.

5 Conclusion

Our method can be used to reduce the dependencies between components in complex concurrent systems. By eliminating control dependencies and enforcing the desired system control behavior automatically, we allow the designers to produce the control aspects of their software modules independently. The independence between design teams makes it possible for them to concentrate on implementing their assigned components for their specific functional requirements.

The use of receptive safety properties allows us to partition the analysis and integration, thus drastically reducing their computational complexity. This capability makes our automated synchronization method relevant for complex industrial systems.

References

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

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

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

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

[5] Gilberto Matos. *"Analysis and Applications of Receptive Safety Properties in Concurrent Systems"*. PhD thesis, University of Maryland, College Park, MD, January 98.

[6] Gilberto Matos, James Purtilo, and Elizabeth White. "Automated Computation of Decomposable Synchronization Conditions". In *Proceedings of the 2nd High Assurance Software Engineering Workshop*, 1997.

[7] Gilberto Matos and Elizabeth White. "Application of Dynamic Reconfiguration in the Design of Fault Tolerant Production Systems". In *Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, May 1998. to appear.

[8] J. Purtilo, C. Falkenberg, E. White, W. Andersen, and T. Ollove. "An exercise with prototying technology". unpublished, January 1994.