Technical Report
CMU/SEI-96-TR-006
ESC-TR-96-006

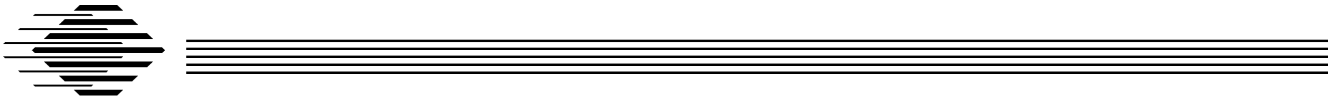
An Architectural Description of the Simplex Architecture

José Germán Rivera
Alejandro Andrés Danylyszyn
Charles B. Weinstock
Lui R. Sha
Michael J. Gagliardi

March 1996

Technical Report
CMU/SEI-96-TR-006
ESC-TR-96-006
March 1996

An Architectural Description of the Simplex Architecture



José Germán Rivera
Alejandro Andrés Danylyszyn
Charles B. Weinstock
Lui R. Sha
Michael J. Gagliardi

Dependable Real-Time Systems

Unlimited distribution subject to the copyright.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER
(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 5/10/96 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTIBILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1 Introduction 1

2 Requirements for Motion Control Simplex 3
2.1 Top Level Requirements 3
2.2 Requirements Analysis 3
2.2.1 Making Changes Safe 3
2.2.2 Making Changes Easy 5
2.2.3 Making Development Faster 7
2.2.4 Using Standardized and COTS Components 7
2.3 A Uni-Processor Example 7

3 System Behavior 9
3.1 Assumptions 9
3.2 Abstraction Techniques 11
3.3 FDR Model Checking Guidelines 13
3.4 Definitions 13
3.5 Preliminary Model 15
3.6 Final Model 18
3.6.1 Process Definitions 18
3.6.2 Properties Verified 21
3.6.3 FDR Verification Results 23
3.6.4 Corrections to the Model 24

4 Software Architecture 27
4.1 Graphical Representation 27
4.2 Wright Specification 28
4.2.1 Process Types 30
4.2.2 DistributionTag Connector 31
4.2.3 Procedure Call Connector 33
4.2.4 Upgrade Manager Component 33
4.2.5 Decision Component 35
4.2.6 Safety Controller Component 37
4.2.7 Untrusted Controller Component 38
4.2.8 Physical I/O Component 38

5 Conclusions 41

6 Acknowledgments 43

Appendix A CSP Models of Simplex **45**
A.1 CSP Model Without Upgrade Manager 45
A.2 First Attempt at a CSP Model with Upgrade Manager 49
A.3 Final Attempt at a CSP Model with Upgrade Manager 55

List of Figures

Figure 1: The Software Hazard Space 4
Figure 2: Simplex Architecture Unit Relationships 8
Figure 3: Simplex Context Diagram 9
Figure 4: Preliminary Model 16
Figure 5: Final Model 18
Figure 6: Software Architecture Excluding Connections to the Upgrade Manager 27
Figure 7: Software Architecture Showing Connections to the Upgrade Manager 28

List of Tables

Table 1: Simplex Events 13
Table 2: Channel Alphabet 15

An Architectural Description of the Simplex Architecture

Abstract: Simplex is a software architecture for dependable and evolvable process-control systems developed by the Software Engineering Institute. Our project consisted of creating a formal specification of this architecture, and analyzing its safety and liveness properties. We developed a Communicating Sequential Processes (CSP) model to describe the overall dynamic behavior of the Simplex architecture, which we verified using the Failure-Divergence-Refinement (FDR) model checker. As a result, we discovered interesting things about the use of FDR that revealed subtle points in the Simplex architecture. We also developed a WRIGHT specification of this architecture to characterize precisely the connections between its components at the architectural level. The specification was based on the latest version of the CSP model.

1 Introduction

The Simplex architecture is a family of high level application development platforms (middleware) that has been designed to support the online evolution of software intensive systems specialized to a specific domain. From the perspective of application developers, it is a collection of online software modification facilities, real time process management and communication facilities, and fault tolerant facilities. In addition, there is a set of application program interfaces (API) that users follow in order to achieve the benefits of easier and safer online software evolution. The Simplex architecture is a technology that is still being matured. Three prototypes of Simplex architecture have been built: a uni-processor motion control prototype, a fault tolerant group motion control prototype, and a radar tracking prototype. Currently, a fourth prototype that supports motion coordination and multi-media communication over local area networks is being designed.

The ideas embodied in the Simplex architecture are likely to be applicable to other domains, though the specifics of implementation would change. A more precise definition of the Simplex architecture will make it easier to understand its properties and for others to adopt it.

One promising technique for precisely describing the architecture is through the use of an Architectural Description Language (ADL). This paper reports the results of an attempt to use Communicating Sequential Processes (CSP) and the Wright ADL to describe the relatively simple uni-processor Simplex prototype. The attempt was largely successful, although a few problems arose because of the real-time requirements of Simplex.

This report is organized as follows. In Section 2, we examine the basic requirements of the Simplex architecture for motion control. Sections 3 and 4 present the Architectural descriptions of the Simplex architecture. In Section 3 we present two models of the system behavior and the properties verified. The abstraction techniques and assumptions are also detailed so

the reader can understand the models and the scope of this work. In Section 4 we present an architectural model of Simplex. In Section 5 we review both the successes and failures of this effort. In particular we examine the difficulties of dealing with real-time fault-tolerant systems using the Wright ADL and CSP.

2 Requirements for Motion Control Simplex

Before we attempt to use an ADL to describe the Simplex architecture, it is helpful to discuss the requirements that an instance of the Simplex architecture must meet. The Simplex architecture is designed to make software changes safer, easier, and cheaper. As mentioned in the introduction, the ideas behind Simplex can be applied to many domains. Our initial domain of application was in the area of motion control. In this section we discuss the requirements for using the Simplex architecture in a motion control environment.

2.1 Top Level Requirements

The Simplex architecture was developed to offer a safe and easy way of upgrading a system while it is in operation. The prototype illustrates its use in providing the ability to perform a safe upgrade of an unstable motion system without shutdown. The requirements that led to the development of the Simplex architecture are:

1. To make changes safer, i.e., to allow the control system to be changed online safely.
2. To make changes easier, i.e., to provide developers with replaceable units that can be modified and replaced online and with facilities to make the change simple.
3. To make the system development faster by providing developers with real-time scheduling and communication facility, fault tolerance facility, and generic motion control utilities such as sequencing control and transaction facilities for ordering and coordinating actions.
4. To lower the system cost by using standard components and popular commercial-of-the-shelf (COTS) components when applicable.

2.2 Requirements Analysis

From the viewpoint of a designer, an architecture specifies an envelope of alternative designs. Each of these designs will have its own characteristics and can meet the requirements. To understand how the requirements constrain the design space, we examine each requirement in turn, and consider its effects on design decisions such as the fault tolerance mechanisms employed, the selection of development platforms, the type of communication facilities required, scheduling disciplines, etc.

2.2.1 Making Changes Safe

To ensure safety and an acceptable level of performance in spite of errors introduced during changes to application software, we need to ensure the timely execution of monitoring and fall back software and protect it from being corrupted. To this end, we need to defend against software faults that could compromise the execution of monitoring and fall back software. The

technique we use to achieve this requirement is known as *analytic redundancy*. This technique uses a proven simple and reliable safety controller as backup for the newly installed or modified and yet-to-be-proven controller. In the event that the new controller fails, the safety controller is ready and able to take control before a system failure occurs.

In the context of motion control, this limits the domain of application to those in which

- The safety and performance of the system is measurable and analyzable at runtime in a timely manner. This gives the safety controller enough time to take over in the event of a failure.
- It is possible to design such a simple and robust safety controller.

Note that the safety controller is not strictly necessary if the system is inherently fail-safe (e.g., gravity may make certain systems end up in a safe state in the event of a controller failure.)

Generally, the advanced controller is expected to be able to follow a reference signal (command) faster and with a higher degree of precision (higher control performance) than the safety controller. However, the trajectories of the advanced controller must be within the controllable states of the safety controller. In other words, the state space of the advanced controller must be a subset of the safety controller's controllable states, although the two controllers may have very different behaviors in response to a command.

In addition to analytic redundancy, the Simplex architecture makes use of Generalized Rate Monotonic Scheduling to avoid resource utilization hazards, and separate (operating system enforced) address spaces for all processes to avoid resource corruption hazards. Communication between processes are conducted via a message passing protocol.

The hazards and how we deal with them are shown pictorially in Figure 1.

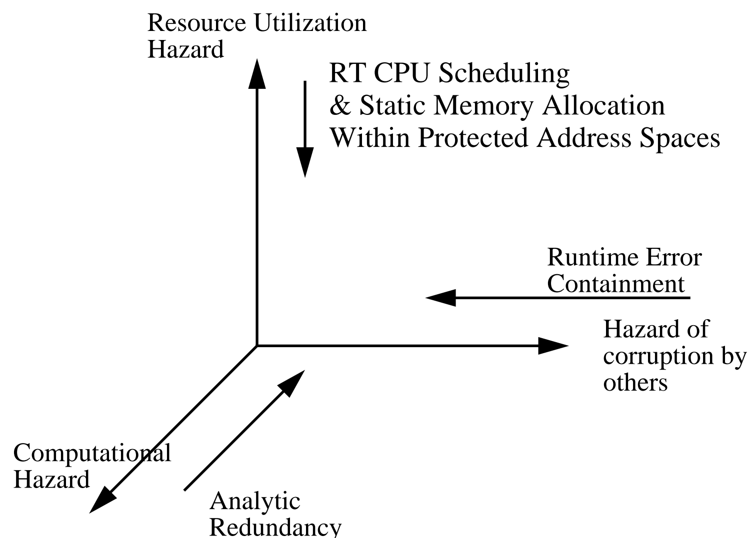


Figure 1: The Software Hazard Space

2.2.2 Making Changes Easy

Support for online software changes is another key requirement. To this end, we need to provide application developers a software packaging construction that can be modified and replaced online. The basic building block of the Simplex Architecture is the *replacement unit*. A replacement unit is a process with a communication template that facilitates its replacement with another replacement unit, online. The new replacement unit might be an improved (or repaired) version of the old one, or have entirely different functionality. Replacement units are designed in such a way that they can be added, deleted, merged, or split online by a set of standardized upgrade transactions. Using the replacement unit as the basic building block allows a uniform approach to support not only the evolution of the application architecture but also that of the Simplex Architecture itself.

Replacement units are specialized into application units and supervisor units. Application units are used to provided functionalities required by the applications. Supervisor units are used to implement process and communication management functions that are independent of the application semantics. In the uni-processor implementation of the Simplex Architecture, application replacement units can be freely replaced. In multi-processor implementations it is also possible to do online replacement of the supervisor units.

Specialized replacement units are assembled into subsystem modules. The system may be built out of one or several subsystem modules. The uni-process Simplex Architecture modeled in this paper consists of one subsystem module. A particular subsystem module is used to implement a distinct application function such as set point control, trajectory generation, motion coordination, and user interfaces. A typical subsystem module consists of a module management unit (often referred to as the Upgrade Manager), one or more application units, and an optional safety unit. The safety unit is intended to implement a safety controller and system performance and safety monitoring functions. Device I/O can reside within the optional safety unit or in a separate application unit. However, it cannot reside within application units whose correctness is questionable since device I/O is always critical to the control of the system.

The upgrade manager is a replacement unit with functions that are designed to support process management, the upgrade operation, and the handling of software faults in an application unit. Structurally, both the safety unit and the application units may be child processes of the management unit, with the safety unit being a trusted and privileged application unit.

Each subsystem module also acts as a software fault-containment unit and normally runs in its own set of address spaces (typically one per replacement unit). This provides protection against resource corruption hazards. Resource utilization hazards also need to be considered. For example, an application unit can burn more CPU cycles than expected because of some error condition. The protection against timing faults can be provided in one of two ways. First, one can keep track of the CPU cycles used by the application units and compare the count

with an expected value. This requires OS support and consumes CPU in the form of some scheduling overhead. Secondly, one can assign to the safety units a higher priority than those of the application units. This approach can be implemented in an OS that supports fixed priority scheduling and if the CPU cost is in the form of bounded priority inversion to all safety units.

Because the creation and destruction of processes are resource-intensive it must occur at a lower priority than those of the units controlling the system. For this reason, when the Upgrade Manager creates a process it assigns a low priority to it. Once the new process has gained all necessary resources in the background, the Upgrade Manager raises its priority to the value required to run normally. To kill a process, the Upgrade Manager first has to lower its priority.

The fundamental operation provided by the Simplex Architecture to support system evolution is the replacement transaction, where one replacement unit is replaced by another. This typically involves moving from a running application to a new (and hopefully improved) version of the application. During this replacement transaction, state information (e.g. those relating to controllers or filters) may need to be transferred from the original unit to the new replacement unit. Alternatively, the new unit may capture the dynamic state information of physical systems through input devices. Without state information, there may be undesirable transients in the behavior of the new replacement unit when it comes online. Hence, the replacement transaction of a single replacement unit is carried out in stages:

1. The new replacement unit is created.
2. New input and any state information is provided to the new replacement unit when it is ready. The new unit begins computations based on the data. The output of the unit is monitored but not used.
3. The upgrade manager waits for the output of the new unit to synchronize or converge to a stable point.
4. Finally, the output of the old unit is turned off and the new unit is turned on. (In practice this means that the outputs from the old unit are ignored in favor of those from the new unit.) The old unit can now be destroyed.

A two-phase protocol can be used when multiple replacement units are to be replaced simultaneously. The first phase is to wait for all the new replacement units to reach a steady state (step 3 above). The second phase is a distributed action that simultaneously switches on all the new replacement units and switches off all the old replacement units. The granularity of “simultaneity” is subject to the accuracy of clock synchronization in a distributed system (but we are not dealing with a distributed version). If the switching is successful, the old replacement units can be destroyed. If the switching of any unit is not successful, the system can automatically switch back to the old replacement units and the replacement transaction can be aborted.

2.2.3 Making Development Faster

Another important requirement is speeding the development process. We achieve this requirement by providing several facilities detailed below.

Fault Tolerance Facility: Fault tolerance approaches have a very significant impact on the system architecture because they dictate the protocols for interactions between members of a fault tolerant group. The approach commonly used is replication. Unfortunately, this approach is not compatible with requirements 1 and 2, safe and easy online evolution. Replication offers no defense against logical errors that could be introduced in a software upgrade. Furthermore, in a replicated system, if only a minority of a fault tolerant group is upgraded, there will be no effect on the system behavior. This is because the majority will see that the minority does not agree and will take corrective action. On the other hand, if one upgrades a majority of the fault tolerant group, there is a real likelihood that the system will fail before it stabilizes. To escape from this upgrade paradox, a new approach must be used. Analytic redundancy, already described in Section 2.2.1, allows well formed diversity among components of a fault tolerant group in the following sense: it permits the replacement of an existing component with a new one that improves the performance while observing the safety constraints. Furthermore, it allows this improved component to gain control; yet it is able to regain control from an active controller who is either non-performing or violates the safety constraint. This allows developers to experiment with new controllers without fear of crashing the system.

Real Time Scheduling and Communication Facility: A real-time scheduling and communication facility is used to perform the creation, replacement, scheduling, and destruction of processes and threads in real time. Communications between software components is indirect through this facility, which makes online replacement possible. The communication facility is able to meet timing and reliability requirements imposed by motion control applications.

2.2.4 Using Standardized and COTS Components

The Simplex architecture makes use of both standardized and COTS components. The Uniprocessor Simplex is built upon a standard real-time POSIX compliant operating system running on a COTS IBM-PC compatible computer. X-Windows is employed for the user interface.

2.3 A Uni-Processor Example

The uni-processor version of the Simplex architecture has six units: *Physical I/O, Decision Unit, Safety Controller, Baseline Controller, Complex Controller,* and *Upgrade Manager*. They can be grouped into *trusted* and *untrusted* components. Components are trusted because of long experience using them, extensive testing, or formal proof of correctness. In the uni-processor version of Simplex the unexpected failure of a trusted component can cause the system to fail. Physical I/O, Upgrade Manager, Safety, and Decision are the trusted components. Baseline and Complex are untrusted components that the application developers have produced to control the system.

The *Physical I/O* unit has control of all input and output to the plant. It is the only component that communicates directly to the device.

The *Upgrade Manager* unit handles all changes in the system configuration (i.e., creation and destruction of units, establishing connections to new units, and disconnecting old units) and is responsible for the replacement transaction described later.

The *Decision* unit is ultimately responsible for the controller (either Complex, Baseline, or Safety) that is in control of the device. When everything is operating smoothly, the *Complex* unit controls the device. If Complex fails hard (i.e., fail-stop) then the *Baseline* unit is given control of the device. If Complex fails in a way that is pushing the device outside the safety region then the *Safety* unit is given control. Once the Safety Controller stabilizes the device, control is given to Baseline and Complex is killed. Should Baseline cause the device to head outside the safety region the Safety unit is given control and Baseline is killed. The Safety unit is the last resort. It guarantees that the device will remain in a safe state, but makes no attempt to maintain performance characteristics. Although this example only shows replacement of Complex, each of these units is a *replacement unit* that can potentially be replaced online. However in practice, for the uni-processor Simplex being described, only the Baseline Controller and the Complex Controller can actually be replaced.

Figure 2 shows graphically the relationship between these units.

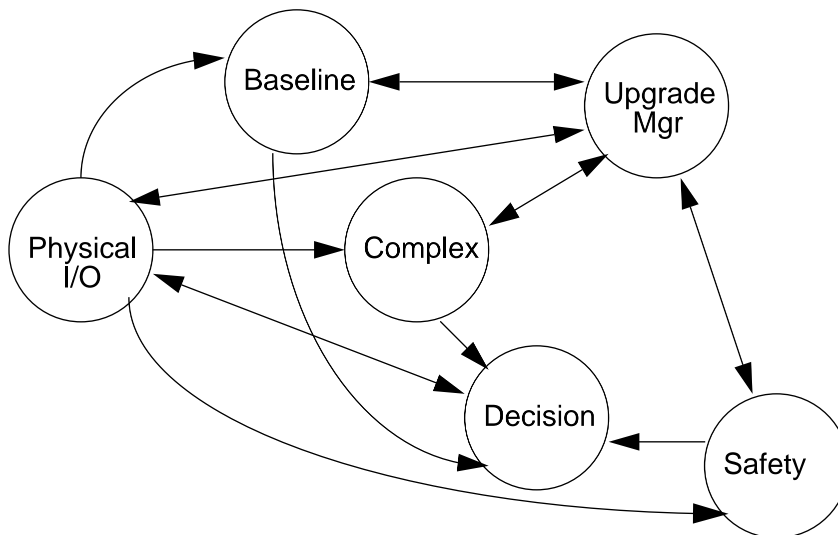


Figure 2: Simplex Architecture Unit Relationships

3 System Behavior

Given the preceding informal description of the uni-processor Simplex architecture, we are ready to attempt to formally describe it. In this chapter we will develop a CSP description of it, while in the next we will use the Wright language to describe it.

The following diagram depicts the interactions of Simplex with its environment:

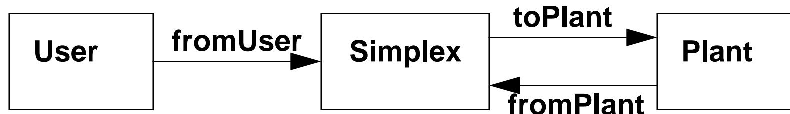


Figure 3: Simplex Context Diagram

In this context, the *Real World* can be modeled as the process resulting from the user, the plant and Simplex interacting:

```
REAL_WORLD = USER || SIMPLEX || PLANT
```

where

```
SIMPLEX = fromPlant?status → toPlant!cntrlout → SIMPLEX
       []
       fromUser?cmd → SIMPLEX
```

The following sections present the assumptions and abstraction techniques, the global definitions, two representations of the internal behavior of Simplex, and the properties verified along with the results of these verifications.

3.1 Assumptions

This section presents the assumptions made while preparing these models, and the assumptions upon which the Simplex architecture is based.

The following are the assumptions we made to simplify the model and restrict the level of abstraction to the point required to represent the architecture and prove the desired properties:

- Neither Wright nor CSP provide support for modeling and analyzing timing behaviors. To work around this problem we are assuming that the hard real-time constraints can be abstracted and do not need to be modeled to define the software architecture.[1] See the abstraction techniques for details on how the timing constraints were handled.

---

1. As we shall see later in Section 5, this assumption ultimately proves invalid.

- The Baseline component is always running before the user starts the Complex component. Also, the user has to kill Complex before being able to kill Baseline. These assumptions imply that whenever the Complex component is running, the Baseline component is also running. In the real prototype the user can start and kill components in any order and at any time. The assumption simplifies the modeling of the *fall-back* technique and reduces the size of the models. To represent the real situation, the model can be expanded to include all possible configurations of the system (i.e., Complex and Safety running, but not Baseline). In that case, if the Complex component fails or is killed, the control of the plant will be transferred to Safety or Baseline depending on the state of the system. With the assumption made, there is only one possible state of the system after Complex disappears and control is transferred to Baseline.
- The start-up of the system is not modeled. We begin the modeling of the system's behavior assuming that all trusted components are running.
- The underlying assumptions that support the Simplex architecture are
  - Every component in the system is assigned a priority. The Physical I/O and Decision components hold the highest priorities. Safety, Baseline, and Complex follow in that order. The lowest priority corresponds to the Upgrade Manager unit.
  - The underlying operating system assures that there is no priority inversion.
  - Resource utilization hazards are avoided by using Generalized Rate Monotonic scheduling and locking all real-time tasks in main memory. Resource corruption hazards are avoided by the underlying operating system and hardware support.

The first two are very important assumptions since they resolve a divergence presented later in this paper.

3.2 Abstraction Techniques

The abstraction techniques used in the models are summarized below to facilitate understanding.

Dynamic creation and destruction of processes: The first state of all processes is “inexistent.” Once they are created by the Upgrade Manager it takes them a finite time to acquire all required resources and be ready to become operational. Thus, other two states can be identified: “initializing” and “ready to run.” These states and the associated transitions are modeled as follows:

Inexistent: process ready to engage in the `start` event.

Initializing: process has engaged in the `start` event and is ready to send an `initDone` event to the Upgrade Manager.

Ready to run: process has sent the `initDone` event to the Upgrade Manager.

The Upgrade Manager can kill a process after a user request or a Decision request to do so. For a process being killed, it takes a finite time to return all allocated resources to the operating system. Two states can be identified: “being killed,” once the kill command is received from the Upgrade Manager; and “dead,” when all system resources have been returned. Then the process returns to the “inexistent” state. These states and transitions are modeled as follows:

Being killed: a process engages in a `kill` event received from the Upgrade Manager.

Dead: a process that engaged in a `kill` event has sent a `dead` event to the Upgrade Manager.

Another situation occurs when an untrusted component fail-stops. In the first model presented, this case is abstracted by having the process generate a `fail-stop` event and return to the “inexistent” state. In the second model, the case is detected by Decision as a `responseTimeout` from the controller.

Replacement transaction completeness criteria: The replacement transaction is completed once the new unit reaches convergence. The convergence criterion is embedded in the Upgrade Manager which is responsible for verifying it. However, a new unit may be defective and never reach convergence. For that reason, new units are given a time for reaching convergence, after which they are killed if they do not succeed. This is modeled by having the new units generate undeterministically a `convergenceDetected` event, or a `convergenceTimeout` event.

Untrusted components' normal behavior and failure: Controller components have a deadline for submitting a control value to the Decision unit. If the component in control of the device misses its deadline, it is killed. Also, a timely response can be erroneous (i.e., out of range, push the device outside the safety region) or correct. We do not model actual values. The types of responses are modeled as:

Illegal: If the value provided is out of range or pushes the device outside the safety region the component sends an `illegalout` event to Decision.

Time-out: If the component missed its deadline for providing a control value to Decision it generates a `responseTimeout` event.

Normal: The component issues a valid control value to Decision by sending the event `cntrlout`.

Plant condition: The device can be operating within the safety region or outside it. Feedback on the status of the plant is received periodically through the Physical I/O component. This feedback is modeled by having the plant generate undeterministically a `safe` or `unsafe` event to the system.

Priority management: One of the underlying assumptions of the Simplex architecture is that processes run with different priorities. We did not model the actual value being assigned to process. However, we modeled the modifications on a component's priority by the Upgrade Manager:

Raise: The Upgrade Manager raises a component's priority by sending a `raise<Component>Prio` message to it.

Lower: The Upgrade Manager lowers a component's priority by sending a `lower<Component>Prio` message to it.

3.3 FDR Model Checking Guidelines

Let M be a CSP model of a software system S, and let *p* be a desired property for the system S. In order to check with FDR that *p* is satisfied by M (that is, M $\models$ *p*) we have to do the following:

- Express the property *p* as a “trivial” CSP process P that describes the sequence of relevant events that characterizes it. P can be seen as the simplest process that satisfies the property *p*.
- Find a modified version of M, named Mp, such that αMp = αP (e.g., using the renaming and hiding operators of CSP).
- Verify P for Mp as follows:

If P is a safety property, **then** check:

$$P \sqsubseteq_T M_p$$

(in FDR syntax: CheckTrace "P" "Mp")

else if P is a liveness property, **then** check:

$$P \sqsubseteq_{FD} M_p$$

(in FDR syntax: Check1 "P" "Mp")

3.4 Definitions

This section provides the definitions for the system events, channel alphabets and channels used in the models presented. The table below presents the definitions of all the event names used in the model.

Table 1: Simplex Events

Event	Description
baselineRunning	The upgrade manager informs Decision that a new baseline controller is running and ready to send output.
cntrlout	One of the controllers (complex, baseline, or safety) generates a valid control output to be sent to the plant.
complexRunning	The upgrade manager informs Decision that a new complex controller is running and ready to send output.
convergenceDetected	The upgrade manager detects that the replacement unit for one of the untrusted components (baseline or complex) has reached convergence (after it was started, as part of a replacement transaction).
convergenceTimeout	The upgrade manager detects that the replacement unit for one of the untrusted components (baseline or complex) has failed to reach convergence within the stipulated time frame.
dead	One of the untrusted controllers (complex or baseline) acknowledges a kill request received from the upgrade manager.

Table 1: Simplex Events

enableOutput	The upgrade manager enables the outputs of one of the untrusted controllers (complex or baseline)
illegalout	One of the untrusted controllers (complex or baseline) generates an illegal control output.
initDone	One of the untrusted controllers (complex or baseline) tells the upgrade manager that it has completed its initialization process.
kill	The upgrade manager asks one of the untrusted controllers (complex or baseline) to die.
KillBaseline	The upgrade manager receives a request to kill the baseline controller.
KillComplex	The upgrade manager receives a request to kill the complex controller.
lowerBaselinePrio	The upgrade manager lowers the priority of the baseline controller.
lowerComplexPrio	The upgrade manager lowers the priority of the complex controller.
raiseBaselinePrio	The upgrade manager raises the priority of the baseline controller.
raiseComplexPrio	The upgrade manager raises the priority of the complex controller.
responseTimeout	One of the untrusted controllers (complex or baseline) does not generate a control output on time. It misses its deadline or falls into an infinite loop.
safe	The Simplex software detects that the plant is operating inside the safety region (operational states [Sha 95])
start	One of the untrusted controllers (complex or baseline) is started by the upgrade manager.
startBaseline	The user requests to start the baseline controller.
startComplex	The user requests to start a new complex controller.
unsafe	The Simplex software detects that the plant is operating outside the safety region (hazard states [Sha 95])
userKillBaseline	The upgrade manager notifies to decision that the user has requested to kill the baseline controller.
userKillComplex	The upgrade manager notifies to decision that the user has requested to kill the complex controller.

The table below presents the definition of the channel alphabets used in the model.

Table 2: Channel Alphabet

Alphabet	Description
FROMPLANT = {safe, unsafe}	Events received from the controlled plant.
TOPLANT = {cntrlout}	Events sent to the controlled plant.
CNTRLEVT = {cntrlout, illegalout, responseTimeout}	Events generated by the controllers.
UMtoCTRL = {start, enableOutput, kill}	Events sent from the Upgrade Manager to the controllers.
CTRLtoUM = {initDone, dead}	Events sent from the controllers to the Upgrade Manager.
DtoUM = {killBaseline, killComplex}	Events sent from Decision to the Upgrade Manager.
UMtoD = {baselineRunning, complexRunning, userKillBaseline, userKillComplex}	Events sent from the Upgrade Manager to Decision.
FROMUSER = {startBaseline, startComplex, killBaseline, killComplex}	Events received from the user.

The following are the declarations of the channels used in the model:

```
pragma channel fromPlant, fromIO: FROMPLANT
pragma channel toPlant, toIO: TOPLANT
pragma channel fromComplex, fromBaseline, fromSafety:
CNTRLEVT
pragma channel UMtoComplex, UMtoBaseline: UMtoCTRL
pragma channel ComplexToUM, BaselineToUM: CTRLtoUM
pragma channel DecisionToUM: DtoUM
pragma channel UMtoDecision: UMtoD
pragma channel fromUser: FROMUSER
```

The independent events observed/produced by the Upgrade Manager are

```
pragma channel convergenceDetected
pragma channel convergenceTimeout
pragma channel raiseBaselinePrio
pragma channel raiseComplexPrio
pragma channel lowerBaselinePrio
pragma channel lowerComplexPrio
```

3.5 Preliminary Model

The CSP model presented next describes the overall dynamic behavior for the uni-processor Simplex architecture. It presents the fall-back mechanism starting from a state in which all components (trusted and untrusted) are already running. This model does not include the Upgrade Manager unit that will be introduced in Section 3.6.

The model is presented in FDR syntax [FDR 92], since the FDR model checker was used to verify the desired properties for it. The following diagram illustrates the CSP processes and channels used in the model:

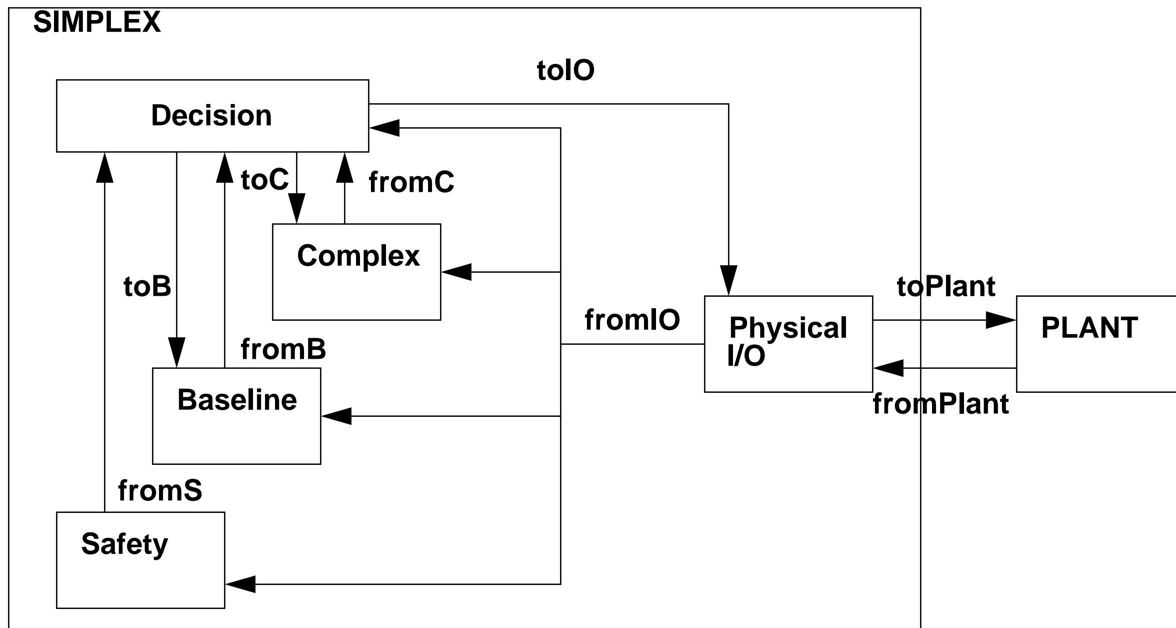


Figure 4: Preliminary Model

- ***Decision Component***

```
DECISION = COMPLEXLOOP
COMPLEXLOOP =
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
   toComplex!kill → TEMPSAFETYLOOP
   []
   fromIO.safe →(fromComplex.cntrlout → toIO!cntrlout →
   COMPLEXLOOP
          []
          ([] x: {illegalout, responseTimeout} •
           fromComplex.x → toComplex!kill → BASELINESAFE)
          []
          fromComplex.failstop → BASELINESAFE)

BASELINESAFE =
   fromBaseline.cntrlout → toIO!cntrlout → BASELINELOOP
   []
   fromBaseline.failstop → fromSafety.cntrlout →
   toIO!cntrlout → SAFETYLOOP
BASELINELOOP =
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
   toBaseline!kill → SAFETYLOOP
   []
   fromIO.safe → BASELINESAFE
TEMPSAFETYLOOP =
```

```
   fromIO.safe → BASELINESAFE
   []
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
   TEMPSAFETYLOOP
SAFETYLOOP = fromIO?x → fromSafety.cntrlout →
   toIO!cntrlout → SAFETYLOOP
```

- ***Physical I/O Component***

```
PHYSICALIO = INPUT ||| OUTPUT
```

-- Take input from Plant:

```
INPUT = fromPlant?status → fromIO!status → INPUT
```

-- Send output to Plant:

```
OUTPUT =  toIO?command → toPlant!command → OUTPUT
```

- ***Complex Component***

```
COMPLEX =
   ((⊓ x: {cntrlout, illegalout, responseTimeout} •
         fromComplex!x → COMPLEX)
    []
    toComplex.kill → SKIP)
   ⊓
   (fromComplex!failstop → SKIP
    []
    toComplex.kill → SKIP)
```

- ***Baseline Component***

```
BASELINE =
   (fromBaseline!cntrlout → BASELINE
    []
    toBaseline.kill → SKIP)
   ⊓
   (fromBaseline!failstop → SKIP
    []
    toBaseline.kill → SKIP)
```

- ***Safety Component***

```
SAFETY =  fromSafety!cntrlout → SAFETY
```

3.6 Final Model

This model is based on the one presented before but it includes the Upgrade Manager unit. The system is modeled starting at a state in which all trusted components are running and new untrusted units (Baseline or Complex) are started, controlled, and killed through the Upgrade Manager. We are primarily modeling the Simplex Architecture and demonstrating its correctness independent of the actual application.

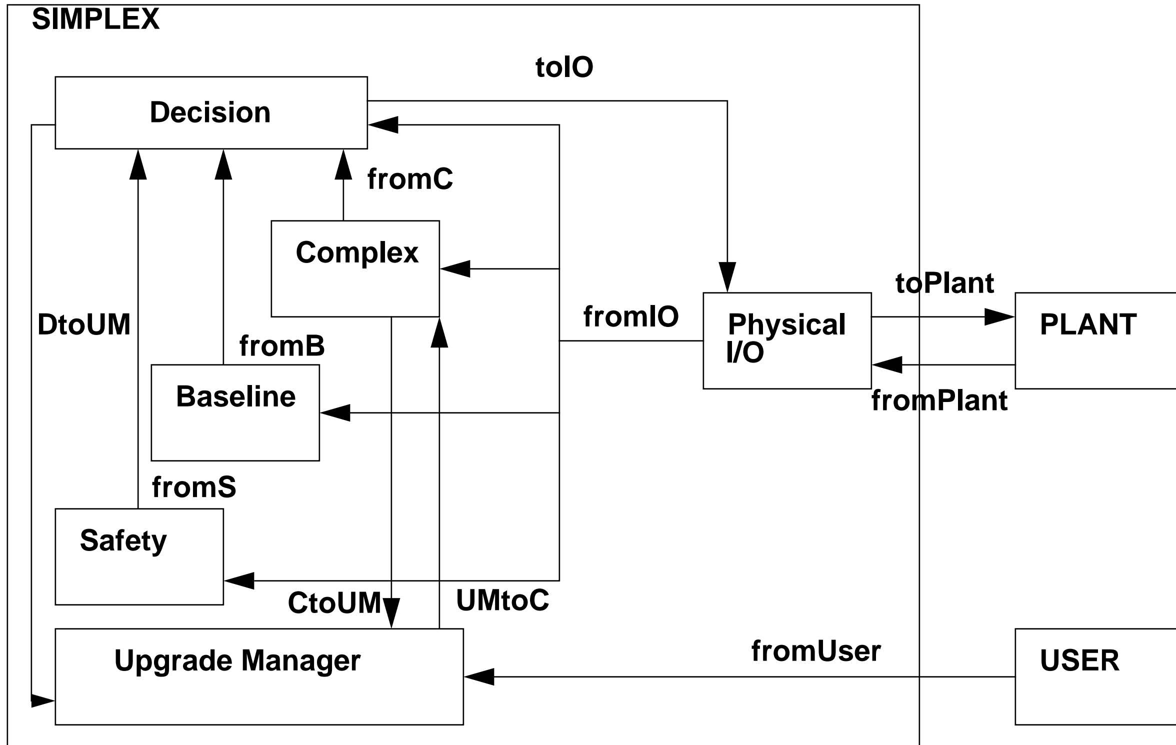


Figure 5: Final Model

To increase readability, not all the channels are shown in the diagram. `Decision` has also an incoming channel from the `Upgrade Manager`. `Baseline` and `Physical I/O` have incoming and outgoing channels with the `Upgrade Manager`. Since `Safety` is actually embedded in `Decision`, it does not have explicit connections with the `Upgrade Manager`.

3.6.1 Process Definitions

This section presents the process definitions for the final model of the system's behavior.

• ***Top-level Process***

```
SIMPLEX =
    UPGRADEMGR
    |{ UMtoComplex, ComplexToUM, UMtoBaseline,
       BaselineToUM, DecisionToUM, UMtoDecision }|
    ((DECISION
```

```
         |{ fromComplex, fromBaseline, fromSafety }|
         (COMPLEX ||| BASELINE ||| SAFETY))
        |{ fromIO, toIO }|
        PHYSICALIO)
```

NOTE: It is important to notice that the synchronization on channel *fromIO* for processes *Complex*, *Baseline,* and *Safety* has been omitted since it is not relevant to the purpose of the model.

- ***Upgrade Manager***

```
UPGRADEMGR = WILLINGTOSTARTBASELINE
WILLINGTOSTARTBASELINE =
   fromUser.startBaseline → UMtoBaseline!start →
   BaselineToUM.initDone → raiseBaselinePrio →
   (convergenceDetected → UMtoBaseline!enableOutput →
   UMtoDecision!baselineRunning → WILLINGTOSTARTCOMPLEX
   ⊓
   convergenceTimeout → KILLBASELINE)
WILLINGTOSTARTCOMPLEX =
   fromUser.startComplex → STARTCOMPLEX
   []
   DecisionToUM.killBaseline → KILLBASELINE
   []
   fromUser.killBaseline → UMtoDecision!userKillBaseline
   → KILLBASELINE
KILLBASELINE =
   lowerBaselinePrio → UMtoBaseline!kill →
   BaselineToUM.dead → WILLINGTOSTARTBASELINE
STARTCOMPLEX =
      UMtoComplex!start → ComplexToUM.initDone →
      raiseComplexPrio →(convergenceTimeout →
      KILLCOMPLEX
   ⊓
      convergenceDetected → UMtoComplex!enableOutput →
      UMtoDecision!complexRunning →
      (fromUser.killComplex →
       UMtoDecision!userKillComplex → KILLCOMPLEX
       []
       DecisionToUM.killComplex → KILLCOMPLEX))
KILLCOMPLEX =
      lowerComplexPrio → UMtoComplex!kill →
      ComplexToUM.dead → WILLINGTOSTARTCOMPLEX
```

- ***Decision Component***

```
DECISION = SAFETYLOOP
SAFETYLOOP =
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
   SAFETYLOOP
   []
   fromIO.safe → (UMtoDecision.baselineRunning →
      BASELINESAFE
```

```
      []
      fromSafety.cntrlout → toIO!cntrlout → SAFETYLOOP)
BASELINELOOP =
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
   DecisionToUM!killBaseline → SAFETYLOOP
   []
   fromIO.safe → (UMtoDecision.complexRunning →
      COMPLEXSAFE
      []
      BASELINESAFE)
   []
   UMtoDecision.userKillBaseline → SAFETYLOOP
BASELINESAFE =
   fromBaseline.cntrlout → toIO!cntrlout → BASELINELOOP
   []
   ([] x: {illegalout, responseTimeout} •
      fromBaseline.x → DecisionToUM!killBaseline →
      fromSafety.cntrlout → toIO!cntrlout → SAFETYLOOP)
COMPLEXLOOP =
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
      DecisionToUM!killComplex → TEMPSAFETYLOOP
   []
   fromIO.safe → COMPLEXSAFE
   []
   UMtoDecision.userKillComplex → BASELINELOOP
COMPLEXSAFE =
   fromComplex.cntrlout → toIO!cntrlout → COMPLEXLOOP
   []
   ([] x: {illegalout, responseTimeout} •
      fromComplex.x → DecisionToUM!killComplex →
      BASELINESAFE)
TEMPSAFETYLOOP =
   fromIO.safe → BASELINESAFE
   []
   fromIO.unsafe → fromSafety.cntrlout → toIO!cntrlout →
      TEMPSAFETYLOOP
```

- ***Safety Controller***

```
SAFETY =  fromSafety!cntrlout → SAFETY
```

- ***Baseline Controller***

```
BASELINE =
   UMtoBaseline.start → BaselineToUM!initDone →
   (UMtoBaseline.enableOutput → BASELINERUNNING
   []
   UMtoBaseline.kill → BaselineToUM!dead → BASELINE)
BASELINERUNNING =
   (⊓ x: {cntrlout, illegalout, responseTimeout} •
      fromBaseline!x → BASELINERUNNING)
   []
   UMtoBaseline.kill → BaselineToUM!dead → BASELINE
```

• ***Complex Controller***

```
COMPLEX =
   UMtoComplex.start → ComplexToUM!initDone →
      (UMtoComplex.enableOutput → COMPLEXRUNNING
   []
   UMtoComplex.kill → ComplexToUM!dead → COMPLEX)
COMPLEXRUNNING =
   (⊓ x: {cntrlout, illegalout, responseTimeout} •
      fromComplex!x → COMPLEXRUNNING)
   []
   UMtoComplex.kill → ComplexToUM!dead → COMPLEX
```

• ***Physical I/O component***

```
PHYSICALIO = INPUT ||| OUTPUT
```

-- Take input from the Plant:

```
INPUT = fromPlant?status → fromIO!status → INPUT
```

-- Send output to the Plant:

```
OUTPUT =  toIO?command → toPlant!command → OUTPUT
```

3.6.2 Properties Verified

This section presents some behavioral properties about the Simplex system that were verified using FDR. The properties are presented along with a simple process that satisfies them. If no comments are appended, the refinement was satisfied and the property holds for the final model.

• Auxiliary definitions

```
SIGMA = {| fromUser, fromPlant, toPlant, fromIO, toIO,
           fromComplex, fromBaseline, fromSafety,
           UMtoComplex, UMtoBaseline, ComplexToUM,
           BaselineToUM, DecisionToUM, UMtoDecision,
           convergenceDetected,
           convergenceTimeout,
           raiseBaselinePrio,
           raiseComplexPrio,
           lowerBaselinePrio,
           lowerComplexPrio |}
pragma channel e
```

• ***Property 1***: Deadlock-free.

```
⊢   P1 is failure-divergence-refined by P1SIMPLEX
```

where:

```
P1 = e → P1
```

```
P1SIMPLEX = identify (SIGMA, e, SIMPLEX)
```

• ***Property 2***: Control commands are sent to the plant infinitely often. That is, it is never the case that from a given instant commands are not sent to the plant anymore.

```
⊦   P2 is failure-divergence-refined by P2SIMPLEX
```

where:

```
P2 = ⊓ x: TOPLANT • toIO!x → P2
P2SIMPLEX = SIMPLEX \ diff (SIGMA, {| toIO |})
```

• ***Property 3***: It is never the case that there are two consecutive readings from the plant without an output command between them. In other words, after reading the status of the plant, and before performing the next reading, a command has to be sent to the plant. (This could be a way of detecting that a deadline was missed.)

```
⊦   P3 is trace-refined by P3SIMPLEX
```

where:

```
P3 = fromIO?x → toIO!cntrlout → P3
P3SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO |})
```

• ***Property 4***: After unsafe condition the plant is immediately controlled by safety, and as long as it is unsafe it remains in control by safety.

```
⊦   P4 is trace-refined by P4SIMPLEX
```

where:

```
P4 =  fromIO.unsafe → fromSafety.cntrlout → toIO.cntrlout
         → P4
      ⊓
      fromIO.safe → (⊓ c: { fromComplex, fromBaseline,
         fromSafety } •
         c.cntrlout → toIO.cntrlout → P4)
P4SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO,
         fromSafety.cntrlout, fromComplex.cntrlout,
         fromBaseline.cntrlout |})
```

• ***Property 5***: Whenever the plant is in safe state and is being controlled by the complex controller, if the complex controller does not produce an output on time (i.e. it misses its deadline or falls into an infinite loop) or if the output is illegal, the control of the device is passed to the baseline controller (or to the safety controller, in case the baseline controller fails).

```
⊦   P5 is trace-refined by P5SIMPLEX
```

where:

```
P5 = fromIO.safe →
   ((⊓ x: {illegalout, responseTimeout} •
```

```
    fromComplex.x → (⊓ c: {fromBaseline, fromSafety} •
    c.cntrlout → P5))
    ⊓
    (⊓ c: {fromComplex, fromBaseline, fromSafety} •
       c.cntrlout → P5))
    ⊓
    fromIO.unsafe → fromSafety.cntrlout → P5
P5SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, fromComplex,
         fromBaseline.cntrlout,
         fromSafety.cntrlout |})
```

• ***Property 6*****:** Whenever Complex is started, Baseline has to be running, and it never happens that there is more than one Baseline or more than one Complex running.

```
⊢   P6 is trace-refined by P6SIMPLEX
```

where:

```
P6 = UMtoBaseline.start → P6AUX
P6AUX = UMtoComplex.start → UMtoComplex.kill → P6AUX
      ⊓
      UMtoBaseline.kill → P6
P6SIMPLEX = SIMPLEX \ diff (SIGMA, { UMtoBaseline.start,
         UMtoBaseline.kill, UMtoComplex.start,
         UMtoComplex.kill })
```

3.6.3 FDR Verification Results

The model presented above has a problem: while properties 3 through 6 are satisfied, properties 1 and 2 are not because ofspecial cases in starting Complex or killing Baseline/Complex that were not considered. When FDR is told to check failure-divergence refinement for property 1, it finds the following failure as a counterexample:

```
After
<fromPlant.safe,fromIO.safe,fromPlant.unsafe,fromUser.startBaseli
ne,UMtoBaseline.start,BaselineToUM.initDone,raiseBaselinePrio,tau
,convergenceDetected,UMtoBaseline.enableOutput,UMtoDecision.basel
ineRunning,fromUser.killBaseline,tau,fromBaseline.illegalout,tau>
 refuses
{|{|fromPlant,fromIO,toPlant,toIO,fromComplex,fromBaseline,fromSa
fety,UMtoComplex,UMtoBaseline,ComplexToUM,BaselineToUM,DecisionTo
UM,UMtoDecision,fromUser,convergenceDetected,convergenceTimeout,r
aiseBaselinePrio,raiseComplexPrio,lowerBaselinePrio,lowerComplexP
rio,tick|}|}
```

Our interpretation of this result from FDR is that at just about the same instant, both the user and Decision could want to kill Baseline. This is the case when the user asks the Upgrade Manager to kill Baseline, and just about at the same time Baseline fails, causing Decision to also ask the Upgrade Manager to kill Baseline. This situation is revealed by FDR as a CSP deadlock in which the Upgrade Manager only wants to engage in the event *UMtoDecision. userKillBaseline* but Decision only wants to engage in the event *DecisionToUM.killBaseline*.

While we were fixing this problem, FDR showed some other failures that revealed other special cases we did not consider. Finally, we modified the model as presented in the next section.

3.6.4 Corrections to the Model

In order for the model to satisfy property 1 we had to modify the specification of the Upgrade Manager unit as shown below. This solution suggests that, at the implementation level, when the Upgrade Manager sends the message *userKillBaseline* to Decision, it should wait for an acknowledge for that message or the message *killBaseline* back from Decision.

```
UPGRADEMGR = WILLINGTOSTARTBASELINE
WILLINGTOSTARTBASELINE =
   fromUser.startBaseline → UMtoBaseline!start →
   BaselineToUM.initDone → raiseBaselinePrio →
   (convergenceDetected → UMtoBaseline!enableOutput →
   UMtoDecision!baselineRunning → WILLINGTOSTARTCOMPLEX
   ⊓
   convergenceTimeout → KILLBASELINE)
WILLINGTOSTARTCOMPLEX =
   fromUser.startComplex → STARTCOMPLEX
   []
   DecisionToUM.killBaseline → KILLBASELINE
   []
   fromUser.killBaseline → (UMtoDecision!userKillBaseline
   → KILLBASELINE
         []
         DecisionToUM.killBaseline → KILLBASELINE)
KILLBASELINE =
   lowerBaselinePrio → UMtoBaseline!kill →
   BaselineToUM.dead → WILLINGTOSTARTBASELINE
STARTCOMPLEX =
   UMtoComplex!start → ComplexToUM.initDone →
raiseComplexPrio →
   (convergenceTimeout → KILLCOMPLEX
   ⊓
   convergenceDetected → UMtoComplex!enableOutput →
   (UMtoDecision!complexRunning →
      (fromUser.killComplex →
      (UMtoDecision!userKillComplex → KILLCOMPLEX
      []
      DecisionToUM.killComplex → KILLCOMPLEX)
      []
      DecisionToUM.killComplex → KILLCOMPLEX)
      []
      DecisionToUM.killBaseline → lowerComplexPrio →
      UMtoComplex!kill → ComplexToUM.dead →
      KILLBASELINE))
KILLCOMPLEX =
   lowerComplexPrio → UMtoComplex!kill →
   ComplexToUM.dead →
   WILLINGTOSTARTCOMPLEX
```

However, the new model including this modification does not satisfy property 2. The reason is a divergence caused by lack of fairness in FDR. When trying to check failure-divergence refinement, FDR finds the following divergence as a counterexample:

```
After <> diverges:
<fromUser.startBaseline,UMtoBaseline.start,BaselineToUM.initDone,
raiseBaselinePrio,tau,convergenceTimeout,lowerBaselinePrio,UMtoBa
seline.kill,BaselineToUM.dead>
```

This divergence presents the case in which the user requests the Upgrade Manager to start a Baseline component that, once started, never reaches convergence. The Upgrade Manager kills the Baseline component but then the user requests a new start-up.

In the actual environment this divergence does not affect the system's ability to send a command to the plant infinitely often. The operating system grants fairness among concurrent processes (see assumptions in Section 3.1). Therefore, Decision and Physical I/O, the two processes with the highest priorities, always have a chance to meet their deadlines.

One of the tasks in requirements engineering is to analyze the current environment and possible future environments to detect extensibility constraints. In this case, the process for starting the Baseline component is initiated by a human being. However, a possible extension to the system is to have a process automatically bring up all the Simplex components. If the user provides a defective Baseline that will never reach convergence, the automatic process will try to start it forever. The divergence has to be solved by adding a maximum number of retries.

Also, in the current environment the Simplex architecture presents only six components. However, in the future users may want to add more intermediate levels of fall-back. At one point, the number of processes running might preclude Decision and Physical I/O from meeting their deadlines. There will be a need for schedulability analysis.

4 Software Architecture

This section presents the Simplex software architecture derived from the models of the behavior of the system.

4.1 Graphical Representation

The following two figures illustrate the overall structure of the Simplex architecture. The first presents the components and connectors excluding the Upgrade Manager. This part of the architecture implements the fall-back mechanism:

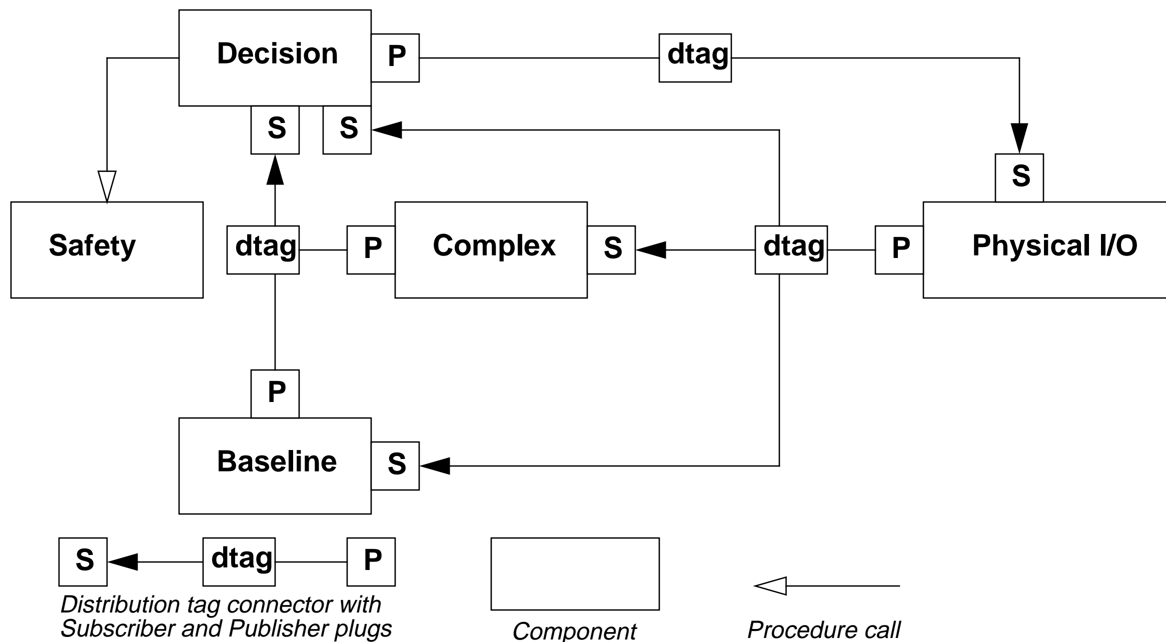


Figure 6: Software Architecture Excluding Connections to the Upgrade Manager

The next view presents all Simplex components and all connectors from/to the Upgrade Manager.

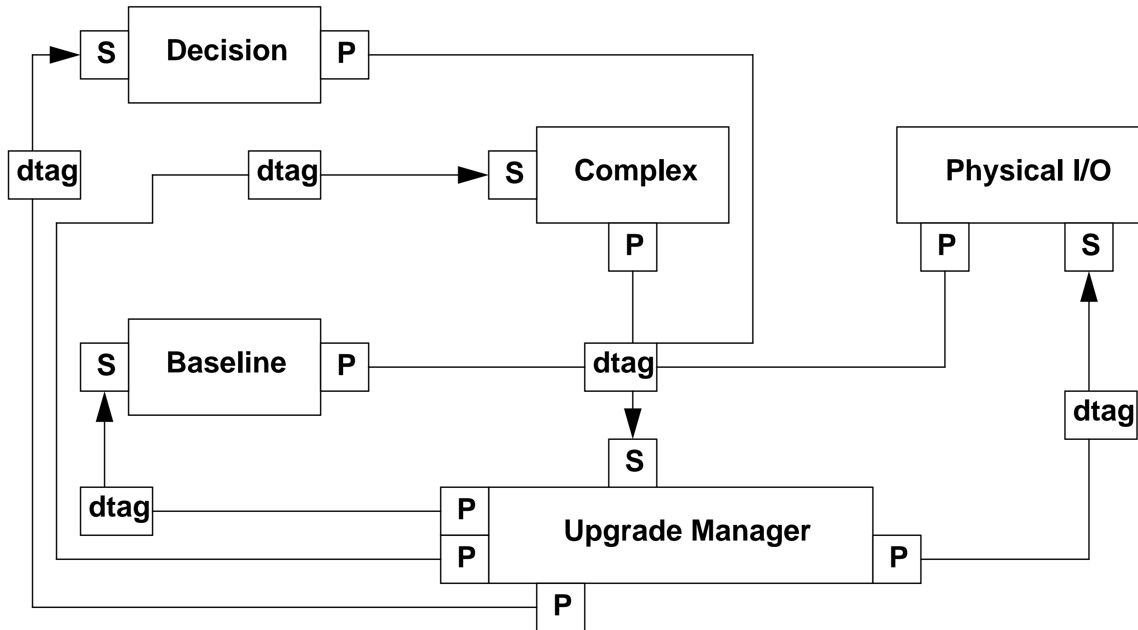


Figure 7: Software Architecture Showing Connections to the Upgrade Manager

These connectors are used to implement the control of processes (i.e., creation, destruction, replacement, priority change). This view of the architecture adds the component and connectors necessary to implement the final model presented. Note that the CSP model used makes it appear that all processes exist a priori, and are sent stop and start messages. This is for modeling purposes only. The actual implementation does not behave this way.

Both views are part of the Simplex software architecture. They were presented separately to avoid the cluttering of the drawing. The formalization of this architecture is presented in the following sections.

4.2 Wright Specification

A formal specification of the Uni-processor Simplex architecture was developed to describe precisely the interaction between the architectural components as well as the required architectural connectors. The Wright notation [FDR 92] was used because it allowed us to structure an architectural description in terms of connectors, components and the behavioral relations among them. The specification was derived from the latest version of the CSP model presented in the previous sections.

The following is the Wright skeleton that describes the overall structure of the Simplex architecture:

```
System SIMPLEX
```

```
   Connector DISTRIBUTION TAG (numPublishers: 1..;
        numSubscribers: 1..)
      Role Publisher1 .. numPublishers
      Role Subscriber1 .. numSubscribers
      Glue
   Connector ProcedureCall
      Role Caller
      Role Declarer
      Glue
   Component UpgradeManager
      Port ToReplacementUnit{Decision, Baseline, Complex, PhysicalIO}
      Port FromReplacementUnits
      Computation
   Component Decision
      Port ToUpgradeManager
      Port FromUpgradeManager
      Port ToPhysicalIO
      Port FromPhysicalIO
      Port FromUntrustedControllers
      Port CallSafety
      Computation
   Component SafetyController
      Port DeclareSafety
      Computation
   Component UntrustedController (id: {Baseline, Complex})
      Port ToUpgradeManager
      Port FromUpgradeManager
      Port ToDecision
      Port FromPhysicalIO
      Computation
   Component PhysicalIO
      Port ToUpgradeManager
      Port FromUpgradeManager
      Port FromDecision
      Port ToPhysicalInputSubscribers
      Computation
Instances
   -- Connectors:
   upgradeManagerInTag: DistributionTag (4, 1)-- data flow to
UpgradeManager
   upgradeManagerOutTag1 .. 4: DistributionTag (1, 1)-- data flow from
UpgradeManager
   decisionInTag : DistributionTag (2, 1)-- data flow to Decision
   physicalIOInTag : DistributionTag (1, 1)-- data flow to PhysicalIO
   physicalIOOutTag : DistributionTag (1, 3)-- data flow from
PhysicalIO
   safetyCall : ProcedureCall
   -- Components:
   upgradeManager: UpgradeManager
   decision: Decision
   safety: SafetyController
   baseline: UntrustedController (Baseline)
   complex: UntrustedController (Complex)
   physicalIO: PhysicalIO
```

```
Attachments
   -- Connections for upgradeManager:
   upgradeManager.ToReplacementUnit_Decision as
upgradeManagerOutTag1.Publsiher
   upgradeManager.ToReplacementUnit_Baseline as
upgradeManagerOutTag2.Publsiher
   upgradeManager.ToReplacementUnit_Complex  as
upgradeManagerOutTag3.Publsiher
   upgradeManager.ToReplacementUnit_PhysicalIO as
upgradeManagerOutTag4.Publsiher
   upgradeManager.FromReplacementUnits  as
upgradeManagerInTag.Subscriber
   -- Connections for decision:
   decision.ToUpgradeManager  as  upgradeManagerInTag.Publisher_1
   decision.FromUpgradeManager  as  upgradeManagerOutTag_1.Subscriber
   decision.ToPhysicalIO  as  physicalIOInTag.Publisher
   decision.FromPhysicalIO  as  physicalIOOutTag.Subscriber_1
   decision.FromUntrustedControllers  as  decisionInTag.Subscriber
   decision.CallSafety  as  safetyCall.Caller
   -- Connections for safety:
   safety.DeclareSafety  as  safetyCall.Declarer
   -- Connections for baseline:
   baseline.ToUpgradeManager  as  upgradeManagerInTag.Publisher_2
   baseline.FromUpgradeManager  as  upgradeManagerOutTag_2.Subscriber
   baseline.ToDecision  as  decisionInTag.Publisher_1
   baseline.FromPhysicalIO  as  physicalIOOutTag.Subscriber_2
   -- Connections for complex:
   complex.ToUpgradeManager  as  upgradeManagerInTag.Publisher_3
   complex.FromUpgradeManager  as  upgradeManagerOutTag_3.Subscriber
   complex.ToDecision  as  decisionInTag.Publisher_2
   complex.FromPhysicalIO  as  physicalIOOutTag.Subscriber_3
   -- Connections for physicalIO:
   physicalIO.ToUpgradeManager  as  upgradeManagerInTag.Publisher_4
   physicalIO.FromUpgradeManager  as
         upgradeManagerOutTag_4.Subscriber
   physicalIO.FromDecision  as  physicalIOInTag.Subscriber
   physicalIO.ToPhysicalInputSubscribers  as
physicalIOOutTag.Publisher
end SIMPLEX
```

The WRIGHT specifications for connectors and components are presented next.

4.2.1 Process Types

The process types used in the connector and port specifications are defined below:

```
Process PublisherLifeCycle =
   getSendAccess → μX.(send!msg → X
                       ⊓
                       releaseSendAccess → (PublisherLifeCycle  ⊓  §))
Process SubscriberLifeCycle =
   subscribe → μX.(receive → return?msg → X
                                   ⊓
```

```
                    receiveWithTimeout → (return?msg → X
                                                    []
                                              timeout → X)
              ⊓
       unsubscribe → (SubscriberLifeCycle ⊓ §))
```

4.2.2 DistributionTag Connector

The DistributionTag connector is a multi-cast message passing mechanism that allows a group of components (publishers) to advertise messages and allows other group of components (subscribers) to receive the published messages [Rajkumar 95]. There can be one or more publishers and one or more subscribers. The publishers do not need to know the identity of the subscribers and vice versa. When a publisher sends a message, the message is broadcasted to all the registered subscribers. For a given distributionTag connector, their publishers and subscribers have to register with it before they can start to publish/receive messages. The number of registered publishers and subscribers can change dynamically. At any time new publishers or subscribers can register, or registered publishers or subscribers can cancel their registration.

An internal priority queue of messages is kept for each subscriber, where published messages are enqueued according to the execution priority of the corresponding publisher. (Queueing has to be according to publisher's priority in order to avoid priority inversion problems for the publishers). It is up to each subscriber to decide when to read messages from its queue. However, if a subscriber tries to read when the queue is empty, it will block unless it specifies a timeout value. Mutual exclusion is required to control concurrent access to the queues in order to guarantee atomicity for queue operations.

In the specification of the distributionTag connector presented below, the numbers used to distinguish between several publishers also represent the execution priorities of them, with number 1 being the highest priority. Although not shown in the specification, the connector has to serve requests from publishers and subscribers in order of arrival (FIFO order), to ensure that no starvation occurs for publishers or subscribers. Also, in order to avoid priority inversion problems when a high-priority publisher/subscriber is waiting for its request to be served, the connector has to use some sort of priority inheritance mechanism. (Typically, this is solved by the underlying operating system.)

```
Connector DistributionTag (numPublishers: 1..;
numSubscribers: 1..)
   Role Publisher_{1 .. numPublishers} = PublisherLifeCycle
   Role Subscriber_{1 .. numSubscribers} = SubscriberLifeCycle
   Glue =
      (ServePublishers{} (numPublishers)
        |||
       ServeSubscribers{} (numSubscribers) )
      ||_{enqueue, dequeue, timeout}
      MsgQueues (numPublishers, numSubscribers)
   where
```

```
   ServePublishersPublishersSet(numPublishers) =
      (∀ p: (1..numPublishers) - PublishersSet  []
         Publisherp.getSendAccess →
         ServePublishersPublishersSet∪{p}(numPublishers))
      []
      (∀ p: PublishersSet  []
         Publisherp.send?msg → enqueuep!msg →
         ServePublishersPublishersSet (numPublishers))
      []
      (∀ p: PublishersSet  []
         Publisherp.releaseSendAccess →
         ServePublishersPublishersSet - {p} (numPublishers))

   ServeSubscribersSubscribersSet(numSubscribers) =
      (∀ s: (1..numSubscribers) - SubscribersSet  []
         Subscribers.subscribe →
         ServeSubscribersSubscribersSet∪ {s} (numSubscribers))
      []
      (∀ s: SubscribersSet  []
         Subscribers.receive → dequeues?msg →
Subscribers.return!msg →
         ServeSubscribersSubscribersSet (numSubscribers))
      []
      (∀ s: SubscribersSet  []
         Subscribers.receiveWithTimeout →
                (dequeues?msg → Subscribers.return!msg →
          ServeSubscribersSubscribersSet(numSubscribers))
          []
          timeouts → Subscribers.timeout →
          ServeSubscribersSubscribersSet(numSubscribers))
      []
      (∀ s: SubscribersSet  []
         Subscribers.unsubscribe →
         ServeSubscribersSubscribersSet - {s}(numSubscribers))
   MsgQueues (numPublishers, numSubscribers) =
      (∀ s: (1..numSubscribers) ||{enqueue}
         Queues, <> (numPublishers))

-- NOTE: When a message is published, the connector has to
-- ensure that  the message is
-- stored in the queues of all the subscribers, before
-- doing anything else. This is
-- represented above by the syncrhonization on {enqueue}.

   Queuesubscriber, buffer(numPublishers) =
      (∀ p: (1..numPublishers)  []
         enqueuep?msg →
         Queuesubscriber, priorityInsert (buffer, p, msg)(numPublishers))
      []
      ((dequeuesubscriber!(head buffer) → Queuesubscriber, tail buffer
(numPublishers))
         if  buffer ≠ <>  else
        (timeoutsubscriber→ Queuesubscriber, buffer(numPublishers)
         []
```

```
          Queue_subscriber. buffer (numPublishers)))
```

The **priorityInsert** function can be defined in Z [Potter 91] as:

```
PRIO == N
[MSG]
PRIOQUEUE == seq (PRIO x MSG)
```

```
priorityInsert: PRIOQUEUE x PRIO x MSG → PRIOQUEUE
-----------------------------------------------------
∀ queue: PRIOQUEUE; prio: PRIO; msg: MSG •
    priorityInsert (queue, prio, msg) =
   if queue = <> then
      <prio, msg>
   else
      if prio < first (head queue) then
      <prio, msg> ⁀ queue
      else
         <head queue> ⁀ priorityInsert (tail queue, prio, msg)
```

4.2.3 Procedure Call Connector

This connector represents the normal procedure call and return sequence to communicate two components, with one exporting a procedure and the other calling that procedure.

```
Connector ProcedureCall
   Role Caller = invoke!x → return?y  → (Caller ⊓ §)
   Role Declarer =  invoke?x → return!y → Declarer
   Glue  =  Caller.invoke?x → Declarer.invoke!x  →
                  Declarer.return?y → Caller.return!y   → Glue
```

4.2.4 Upgrade Manager Component

The Upgrade Manager component provides the online upgrade services. In the context of the Uni-processor Simplex architecture, it allows the user to start and terminate the untrusted controllers (Baseline and Complex) without requiring shut-down of the entire system. Also, it accepts requests from the Decision component to kill Baseline or Complex.

The specification presented below includes an implementation of the proposed solution to the race condition problem that was detected by the FDR and presented in the previous section.

```
Component UpgradeManager
   Port ToReplacementUnit_{Decision, Baseline, Complex, PhysicalIO} = PublisherLifeCycle
   Port FromReplacementUnits = SubscriberLifeCycle
   Computation =
      ToReplacementUnit_Decision.getSendAccess →
      ToReplacementUnit_PhysicalIO.getSendAccess →
      FromReplacementUnits.subscribe →
      WILLINGTOSTARTBASELINE
   where
```

```
WILLINGTOSTARTBASELINE =
   readUserInput → fromUser.startBaseline →
   ToReplacementUnit_Baseline.getSendAccess →
   ToReplacementUnit_Baseline.send!start →
   FromReplacementUnits.receive →
   FromReplacementUnits.return.Baseline.initDone →
   raiseBaselinePrio →
   (convergenceDetected →
      ToReplacementUnit_Baseline.send!enableOutput →
      ToReplacementUnit_Decision.send!baselineRunning →
      WILLINGTOSTARTCOMPLEX
    ⊓
    convergenceTimeout → KILLBASELINE)
WILLINGTOSTARTCOMPLEX =
   readUserInput →

      (fromUser.startComplex → STARTCOMPLEX
       []
       fromUser.killBaseline →
       ToReplacementUnit_Decision.send!userKillBaseline →
       FromReplacementUnits.receive →
       (FromReplacementUnits.return.Decision.acknowledge →
         KILLBASELINE
         []
         FromReplacementUnits.return.Decison.killBaseline →
         KILLBASELINE))
   ⊓
   FromReplacementUnits.receive →
      FromReplacementUnits.return.Decision.killBaseline →
      KILLBASELINE
KILLBASELINE =
   lowerBaselinePrio →
   ToReplacementUnit_Baseline.send!kill →
   FromReplacementUnits.receive →
   FromReplacementUnits.return.Baseline.dead →
   ToReplacementUnit_Baseline.releaseSendAccess →
   WILLINGTOSTARTBASELINE
STARTCOMPLEX =
   ToReplacementUnit_Complex.getSendAccess →
   ToReplacementUnit_Complex.send!start →
   FromReplacementUnits.receive →
   FromReplacementUnits.return.Complex.initDone →
   raiseComplexPrio →
   (convergenceTimeout → KILLCOMPLEX
    ⊓
    convergenceDetected → COMPLEXCONVERGED)
KILLCOMPLEX =
   lowerComplexPrio →
   ToReplacementUnit_Complex.send!kill →
   FromReplacementUnits.receive →
   FromReplacementUnits.return.Complex.dead →

   ToReplacementUnit_Complex.releaseSendAccess →
   WILLINGTOSTARTCOMPLEX
```

```
COMPLEXCONVERGED =
    ToReplacementUnit_Complex.send!enableOutput →
   (ToReplacementUnit_Decision.send!complexRunning →
      (readUserInput → fromUser.killComplex →
      ToReplacementUnit_Decision.send!userKillComplex →
      FromReplacementUnits.receive →
      (FromReplacementUnits.return.Decision.acknowledge →
       KILLCOMPLEX
        []
        FromReplacementUnits.return.Decision.killComplex →
       KILLCOMPLEX)
       ⊓
       FromReplacementUnits.receive →
       FromReplacementUnits.return.Decision.killComplex →
       KILLCOMPLEX)
     ⊓
     FromReplacementUnits.receive →
      FromReplacementUnits.return.Decision.killBaseline →
      lowerComplexPrio →
      ToReplacementUnit_Complex.send!kill →
      FromReplacementUnits.receive →
      FromReplacementUnits.return.Complex.dead →
      KILLBASELINE)
```

4.2.5 Decision Component

The Decision component receives the output sent by the safety controller and the untrusted controllers (Baseline and Complex) and decides which one to send to the controlled plant, and sends it to the physical I/O component. Initially, Decision takes the output from the safety controller, which is a built-in procedure inside Decision. When the Baseline component is started, Decision starts to take its output instead of the safety controller's output, and keeps taking Baseline's output until it detects a safety hazard or it is told by the Upgrade Manager that the user wants to kill Baseline. Possible safety hazards include the plant going to an unsafe state (leaving the safety region), and Baseline sending illegal output or missing its deadline. When Decision detects a safety hazard, it returns to take the output from the safety controller and asks the Upgrade Manager to kill Baseline.

If the Complex controller is started while Decision is taking Baseline's output, Decision starts to take Complex's output instead, and keeps doing so until it detects a safety hazard or it is told by the Upgrade Manager that the user wants to kill Complex. If the former case, Decision asks the Upgrade Manager to kill Complex, and temporarily takes the output from the a safety controller until the plant is in a safe state again. Then, it switches to take Baseline's output and things continue as described above. In the latter case, Decision just switches to take Baseline's output.

```
Component Decision
   Port ToUpgradeManager = PublisherLifeCycle
   Port FromUpgradeManager = SubscriberLifeCycle
   Port ToPhysicalIO = PublisherLifeCycle
```

```
  Port FromPhysicalIO = SubscriberLifeCycle
  Port FromUntrustedControllers = SubscriberLifeCycle
  Port CallSafety = invoke → return?y  → CallSafety
  Computation =
     ToUpgradeManager.getSendAccess →
     FromUpgradeManager.subscribe →
     ToPhysicalIO.getSendAccess →
     FromPhysicalIO.subscribe →
     FromUntrustedControllers.subscribe →
     SAFETYLOOP
  where
  SAFETYLOOP =
     FromPhysicalIO.receive →
n?cntrlout →
        ToPhysicalIO.send!cntrlout → SAFETYLOOP
      []
      FromPhysicalIO.return.safe →
        (FromUpgradeManager.receive →
         FromUpgradeManager.return.baselineRunning →
         BASELINESAFE
         ⊓
         CallSafety.invoke → CallSafety.return?cntrlout →
         ToPhysicalIO.send!cntrlout → SAFETYLOOP))
  BASELINELOOP =
     FromPhysicalIO.receive →
        (FromPhysicalIO.return.unsafe →
         CallSafety.invoke → CallSafety.return?cntrlout →
         ToPhysicalIO.send!cntrlout →
        ToUpgradeManager.send!killBaseline → SAFETYLOOP
         []
         FromPhysicalIO.return.safe →
         (FromUpgradeManager.receive →
         FromUpgradeManager.return.complexRunning →
         COMPLEXSAFE
          ⊓
          BASELINESAFE)
     ⊓
     FromUpgradeManager.receive →
        FromUpgradeManager.return.userKillBaseline →
        ToUpgradeManager.send!acknowledge → SAFETYLOOP
  BASELINESAFE =
     FromUntrustedControllers.receiveWithTimeout →
     (FromUntrustedControllers.return.Baseline.cntrlout →
        ToPhysicalIO.send!cntrlout → BASELINELOOP
      []
      FromUntrustedControllers.return.Baseline.illegalout →
        ToUpgradeManager.send!killBaseline →
        CallSafety.invoke → CallSafety.return?cntrlout →
        ToPhysicalIO.send!cntrlout → SAFETYLOOP
      []
      FromUntrustedControllers.timeout →
        ToUpgradeManager.send!killBaseline →
        CallSafety.invoke → CallSafety.return?cntrlout →
        ToPhysicalIO.send!cntrlout → SAFETYLOOP
```

```
COMPLEXLOOP =
   FromPhysicalIO.receive →
      (FromPhysicalIO.return.unsafe →
       CallSafety.invoke → CallSafety.return?cntrlout →
       ToPhysicalIO.send!cntrlout →
       ToUpgradeManager.send!killComplex →
       TEMPSAFETYLOOP
       []
       FromPhysicalIO.return.safe → COMPLEXSAFE)
   ⊓
   FromUpgradeManager.receive →
      FromUpgradeManager.return.userKillComplex →
      ToUpgradeManager.send!acknowledge →
      BASELINELOOP
COMPLEXSAFE =
   FromUntrustedControllers.receiveWithTimeout →
   (FromUntrustedControllers.return.Complex.cntrlout →
      ToPhysicalIO.send!cntrlout → COMPLEXLOOP
    []
    FromUntrustedControllers.return.Complex.illegalout →
      ToUpgradeManager.send!killComplex → BASELINESAFE
    []
    FromUntrustedControllers.timeout →
      ToUpgradeManager.send!killComplex → BASELINESAFE)
TEMPSAFETYLOOP =
   FromPhysicalIO.receive →
   (FromPhysicalIO.return.safe → BASELINESAFE
    []
    FromPhysicalIO.return.unsafe →
      CallSafety.invoke → CallSafety.return?cntrlout →
      ToPhysicalIO.send!cntrlout → TEMPSAFETYLOOP
```

4.2.6 Safety Controller Component

The SafetyController component is a trusted controller that has been extensively tested and/or formally verified. It is the default controller and the last resort in case of failure of the untrusted controllers (Baseline and Complex).

```
Component SafetyController
   Port DeclareSafety = invoke → return!y → DeclareSafety
   Computation =
      DeclareSafety.invoke →
      DeclareSafety.return!cntrlout → Computation
```

4.2.7 Untrusted Controller Component

The UntrustedController component represents any type of controller that has a better performance than the safety controller but that may not be as reliable as the safety controller. Untrusted controllers are assumed to fail at any time by producing an illegal output or by missing its deadline (not producing its output in time, or falling into infinite loop). In the Uni-processor Simplex architecture there are two untrusted controllers: Baseline and Complex (Complex is supposed to be more sophisticated than Baseline, thus providing better performance, but perhaps less reliability).

```
Component UntrustedController (id: {Baseline, Complex})
   Port ToUpgradeManager = PublisherLifeCycle
   Port FromUpgradeManager = SubscriberLifeCycle
   Port ToDecision = PublisherLifeCycle
   Port FromPhysicalIO = SubscriberLifeCycle
   Computation =
      ToUpgradeManager.getSendAccess →
      FromUpgradeManager.subscribe →
      FromUpgradeManager.receive → FromUpgradeManager.return.start →
      ToDecision.getSendAccess →    FromPhysicalIO.subscribe →
      ToUpgradeManager.send!id.initDone →
      FromUpgradeManager.receive →
      (FromUpgradeManager.return.enableOutput →
         CONTROLLERRUNNING
       []
       FromUpgradeManager.return.kill → CONTROLLERKILLED)
   where
   CONTROLLERRUNNING =
      ToDecision.send!id.cntrlout → CONTROLLERRUNNING
      ⊓
      ToDecision.send!id.illegalout → CONTROLLERRUNNING
      ⊓
      CONTROLLERRUNNING
      ⊓
      FromUpgradeManager.receive →
         FromUpgradeManager.return.kill → CONTROLLERKILLED

   CONTROLLERKILLED =
      ToDecision.releaseSendAccess → ToPhysicalIO.unsubscribe →
      ToUpgradeManager.send!id.dead →
      ToUpgradeManager.releaseSendAccess →
      FromUpgradeManager.unsubscribe → Computation
```

4.2.8 Physical I/O Component

The PhysicalIO component provides physical I/O services to the other components in the system. It is the only component that has access to the I/O devices. It reads input from the plant sensors and broadcasts it to the appropriate components, and writes output from Decision to the plant control devices.

```
Component PhysicalIO
```

```
Port ToUpgradeManager = PublisherLifeCycle
Port FromUpgradeManager = SubscriberLifeCycle
Port FromDecision = SubscriberLifeCycle
Port ToPhysicalInputSubscribers = PublisherLifeCycle
Computation =
   ToUpgradeManager.getSendAccess →
   FromUpgradeManager.subscribe →

   FromDecision.subscribe →
   ToPhysicalInputSubscribers.getSendAccess →

   (INPUT ||| OUTPUT)
where
INPUT =
   readPlantSensors → fromPlant?sensorInputs →
   ToPhysicalInputSubscribers.send!sensorInputs → INPUT
OUTPUT =
   FromDecision.receive → FromDecision.return?IOcommand →
   writeToPlantControlDevices!IOcommand → OUTPUT
```

5 Conclusions

The main strength of the Wright notation is that it decouples the specification of externally observable behavior (i.e., external interfaces) from the internal behavior (i.e., internal flow of control) of architectural components. Also, the Wright connectors capture the interconnection mechanisms required by the architecture as first class entities. This feature enhances the opportunities for reuse, as connectors can be made explicit and decoupled from the application components. As an example, consider the *DistributionTag* connector presented in Section 4.2.2 that can be reused in any other application because of its application independence.

However, Wright does not offer checking of system-wide properties. The available Wright tools allow the developers to check only properties local to each component. By deriving the Wright model from the CSP model we gained confidence in the robustness of the system.

Proving deadlock-free using FDR is very useful. Independently of what deadlock-free means in the system being modeled, the FDR checker may help in revealing race conditions and checking that the model is consistent and well-formed.

During the preparation of this work we verified empirically the effectiveness of peer reviews for inspecting specifications. That allowed us to identify and remove subtle defects present in previous versions of the models. The rigor of design reviews can be as great as the rigor of code inspections thanks to the precision of formal notation. In the context of software development this technique allows the correction of errors early on in the process when it is cheaper to remove them.

We showed empirically the power of formal specifications and model checkers to detect subtleties in design that otherwise cannot be uncovered until later stages of the product life cycle. A formal specification of the software architecture can be very useful as a guidance for developing better testing strategies and test cases (e.g., by identifying dependencies, complex interactions, etc.). It is important to interpret counter-examples provided by the FDR checker in the context of reality. However, be aware that there are three possible causes behind the checking failure:

- the representation of the property is not well-formed
- the model is ill-formed and the counter-example reveals the problem
- the system being modeled has a flaw that is revealed by the model checker.

Upon discovery of a counter-example these three possibilities should be followed as a checklist. An example of the problems found in the system being modeled is the divergence presented in Section 3.6.4.

On the other hand, the formal model is not defect free either. In spite of many meetings between the designers of the Simplex architecture and the developers of the Wright and CSP descriptions, there is at least one serious error in the descriptions as presented here. Property 5 on page 22 states, “Whenever the plant is in safe state and is being controlled by the com-

plex controller, if the complex controller does not produce an output on time (i.e. it misses its deadline or falls into an infinite loop) ...illegal, the control of the device is passed to the baseline controller (or to the safety controller, in case the baseline controller fails)." This property is said to be verified by the FDR. However, there is a counter example. In the CSP model, the upgrade manager has priority lower than the controllers. Suppose that the complex controller enters an infinite loop. Since the upgrade manager has the lowest priority, the upgrade manager will be preempted by the complex controller and therefore cannot take the control from it and give it to the baseline controller. On the other hand, assigning a higher priority to the upgrade manager would also be incorrect, since the process creation and destruction operation is very time consuming. Given high priorities, such actions will cause real-time tasks to miss their deadlines.

In reality the upgrade manager is multi-threaded. It has a higher priority thread that manages the change of priorities and issues commands. The low priority thread carries out the actual process management. Thus, when complex enters an infinite loop, the high priority thread of the upgrade manager can lower complex's priority and then the lower priority thread of the upgrade manager can then kill the complex process at its leisure.

The problem with the CSP model is that events are not modeled as preemptable. That is, if the process that processes certain events is preempted, the said events will not occur. If the events are modeled as the results of executing processes, FDR should be able to show that property 5 is false until the upgrade manager is modeled as multi-threaded processes with one thread's priority higher than the applications and another thread's priority lower than the applications.

From the viewpoint of system architecture modeling, this problem is a result of the level of abstraction chosen for the architectural description. Choosing the right level to highlight the key features of the architecture without resulting in an overly complex description is not an easy job. In the case of the Simplex architecture description it may have been possible to choose a lower level abstraction that would have allowed for the modeling of preemptability and priorities of concurrent processes. Although this was suggested a number of times during the modeling, it was rejected on the grounds that this would complicate the model, making it even harder to understand. It was conjectured that the current level of abstraction would be adequate. In retrospect, the real reason for this was probably that the specification techniques and tools did not provide facilities to represent time explicitly.

Since it is difficult to determine what level of abstraction is appropriate a priori, it may be necessary to perform a separate schedulability modeling analysis in addition to the Wright and CSP models when real-time systems are involved. The real-time issue should be addressed in the future, perhaps by somehow marrying the current Architectural Description Language technologies with Rate Monotonic Scheduling theory.

6 Acknowledgments

An earlier version of this paper appeared as a CMU School of Computer Science Report, CMU-CS-95-224 by José German Rivera and Alejandro Andrés Danylyszyn under the title *Formalizing the Uni-processor Simplex Architecture*. That research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and N66001-95-C-8623; and by National Science Foundation Grant CCR-9357792. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, or the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

The authors would like to thank professor David Garlan for his advice and review comments, which were very valuable and gave insight to define the strategy for specifying the Simplex architecture.

Appendix A CSP Models of Simplex

A.1 CSP Model Without Upgrade Manager

```
-- +=========================================================+
-- | CSP Model of the overall dynamic behavior for the       |
-- | Uni-processor Simplex Architecture.                     |
--                                                           |
-- | Author: Jose German Rivera                              |
-- | Alejandro Andres Danylyszyn                             |
--                                                           |
-- | Created: 10/4/95 - Initial version: Upgrade Manager     |
-- | not included.                                           |
-- +=========================================================+

-- +=========================================================+
-- | Channel Alphabets                                       |
-- +=========================================================+

-- Events received from the controlled plant:
FROMPLANT = {safe, unsafe}

-- Events sent to the controlled plant:
TOPLANT = {cntrlout}

-- Events generated by the controllers:
CNTRLEVT = {cntrlout, failstop, illegalout, responseTimeout}

-- Event sent to the controllers:
TOCNTRL = {kill}

-- +=========================================================+
-- | Channel Declarations                                    |
-- +=========================================================+

pragma channel fromPlant, fromIO: FROMPLANT
pragma channel toPlant, toIO: TOPLANT
pragma channel fromComplex, fromBaseline, fromSafety: CNTRLEVT
pragma channel toComplex, toBaseline, toSafety: TOCNTRL

-- +=========================================================+
-- | Process Definitions                                     |
-- +=========================================================+

--
-- Simplex architecture:
--

SIMPLEX = (DECISION
 [| {| fromIO, toIO |} |]
 PHYSICALIO)
 [| {| fromComplex, toComplex, fromBaseline, toBaseline,
 fromSafety, toSafety |} |]
 (COMPLEX ||| BASELINE ||| SAFETY)
```

```
--
-- Decision Component:
--
DECISION = COMPLEXLOOP

COMPLEXLOOP = fromIO.unsafe -> fromSafety.cntrlout ->
toIO!cntrlout ->
 toComplex!kill -> TEMPSAFETYLOOP
 []
     fromIO.safe -> (fromComplex.cntrlout -> toIO!cntrlout ->
 COMPLEXLOOP
 []
 ([] x: {illegalout, responseTimeout} @
 fromComplex.x -> toComplex!kill ->
 BASELINESAFE)
 []
 fromComplex.failstop -> BASELINESAFE)

BASELINESAFE = fromBaseline.cntrlout -> toIO!cntrlout ->
BASELINELOOP
 []
 fromBaseline.failstop -> fromSafety.cntrlout ->
 toIO!cntrlout -> SAFETYLOOP

BASELINELOOP = fromIO.unsafe -> fromSafety.cntrlout ->
toIO!cntrlout ->
 toBaseline!kill -> SAFETYLOOP
 []
                                  fromIO.safe -> BASELINESAFE

TEMPSAFETYLOOP = fromIO.safe -> BASELINESAFE
 []
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 TEMPSAFETYLOOP

SAFETYLOOP = fromIO?x -> fromSafety.cntrlout -> toIO!cntrlout
-> SAFETYLOOP

--
-- Physical I/O component:
--
PHYSICALIO = INPUT ||| OUTPUT

-- Take input from Plant:
INPUT = fromPlant?status -> fromIO!status -> INPUT

-- Send output to Plant:
OUTPUT = toIO?command -> toPlant!command -> OUTPUT

--
-- Complex Component:
--
COMPLEX = ((|~| x: {cntrlout, illegalout, responseTimeout} @
 fromComplex!x -> COMPLEX)
 []
 toComplex.kill -> SKIP)
 |~|
 (fromComplex!failstop -> SKIP
```

```
 []
 toComplex.kill -> SKIP)

--
-- Baseline Component:
--
BASELINE = (fromBaseline!cntrlout -> BASELINE
 []
 toBaseline.kill -> SKIP)
 |~|
 (fromBaseline!failstop -> SKIP
 []
 toBaseline.kill -> SKIP)

--
-- Safety Component:
--
SAFETY = fromSafety!cntrlout -> SAFETY


-- +=========================================================+
-- | Properties to verify                                    |
-- +=========================================================+

-- Auxiliary definitions:
SIGMA = {| fromPlant, toPlant, fromIO, toIO,
 fromComplex, fromBaseline, fromSafety,
                           toComplex, toBaseline, toSafety |}

pragma channel e


-- Property 1. Deadlock-free.
--           |- P1 is failure-divergence-refined by P1SIMPLEX
-- where:

P1 = e -> P1

P1SIMPLEX = identify (SIGMA, e, SIMPLEX)


-- Property 2. Control commands are sent to the plant
infinitely often. That is,
-- it is never the case that from a given instant commands are
not sent
-- to the plant anymore.
--           |- P2 is failure-divergence-refined by P2SIMPLEX
-- where:

P2 = |~| x: TOPLANT @ toIO!x -> P2

P2SIMPLEX = SIMPLEX \ diff (SIGMA, {| toIO |})


-- Property 3. It is never the case that there are two
-- consecutive readings from
-- the plant without an output command between them. In other
-- words, after
```

```
-- reading the status of the plant, and before performing the
-- next reading,
-- a command has to be sent to plant. (This could be a way of
-- detecting that
-- a deadline was missed)
--                        |- P3 is trace-refined by P3SIMPLEX
-- where:

P3 = fromIO?x -> toIO!cntrlout -> P3

P3SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO |})


-- Property 4. After unsafe condition the plant is immediately
-- controlled by
-- safety, and as long as it is unsafe it remains in control
-- by safety:
--                        |- P4 is trace-refined by P4SIMPLEX
-- where:

P4 = fromIO.unsafe -> fromSafety.cntrlout -> toIO.cntrlout ->
P4
 |~|
 fromIO.safe -> (|~| c: { fromComplex, fromBaseline, fromSafety
} @
 c.cntrlout -> toIO.cntrlout -> P4)

P4SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO,
fromSafety.cntrlout,
 fromComplex.cntrlout,
 fromBaseline.cntrlout |})


-- Property 5. Whenever the plant is in safe state and is being
-- controlled
-- by the complex controller, if the complex controller does
-- not
-- produce an output on time (i.e. it misses its deadline or
-- falls into an
-- infinite loop) or if the output is illegal, the control of
-- the device is
-- passed to the baseline controller (or to the safety
-- controller, in case
-- the baseline controller fails).
--                        |- P5 is trace-refined by P5SIMPLEX
-- where:

P5 = fromIO.safe ->
 ((|~| x: {illegalout, responseTimeout, failstop} @
 fromComplex.x -> (|~| c: {fromBaseline, fromSafety} @
 c.cntrlout -> P5))
 |~|
 (|~| c: {fromComplex, fromBaseline, fromSafety} @ c.cntrlout
-> P5)
 )
 |~|
 fromIO.unsafe -> fromSafety.cntrlout -> P5
```

```
P5SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, fromComplex,
 fromBaseline.cntrlout,
 fromSafety.cntrlout |})
```

A.2 First Attempt at a CSP Model with Upgrade Manager

```
-- +=========================================================+
-- | CSP Model of the overall dynamic behavior for the       |
-- | Uni-processor Simplex Architecture.                     |
--                                                           |
-- | Author: Jose German Rivera                              |
-- | Alejandro Andres Danylyszyn                             |
--                                                           |
-- | Created: 10/4/95 - Initial version: Upgrade Manager     |
-- | not included.                                           |
-- | Updated: 11/12/95 - Included Upgrade Manager. It has a  |
-- | problem: properties 1-2 are not                         |
-- | satisfied due to the fact that some                     |
-- | special cases of starting Complex                       |
-- | or killing Baseline/Complex were                        |
-- | not considered. Property 6 added.                       |
-- +=========================================================+

-- +=========================================================+
-- | Channel Alphabets                                       |
-- +=========================================================+

-- Events received from the controlled plant:
FROMPLANT = {safe, unsafe}

-- Events sent to the controlled plant:
TOPLANT = {cntrlout}

-- Events generated by the controllers:
CNTRLEVT = {cntrlout, illegalout, responseTimeout}

-- Events sent from the Upgrade Manager to the controllers:
UMtoCTRL = {start, enableOutput, kill}

-- Events sent from the controllers to the Upgrade Manager:
CTRLtoUM = {initDone, dead}

-- Events sent from Decision to the Upgrade Manager:
DtoUM = {killBaseline, killComplex}

-- Events sent from the Upgrade Manager to Decision:
UMtoD = {baselineRunning, complexRunning, userKillBaseline,
 userKillComplex}

-- Events Received from the user:
FROMUSER = {startBaseline, startComplex, killBaseline,
 killComplex}
```

```
-- +==========================================================+
-- | Channel Declarations                                     |
-- +==========================================================+

pragma channel fromPlant, fromIO: FROMPLANT
pragma channel toPlant, toIO: TOPLANT
pragma channel fromComplex, fromBaseline, fromSafety: CNTRLEVT
pragma channel UMtoComplex, UMtoBaseline: UMtoCTRL
pragma channel ComplexToUM, BaselineToUM: CTRLtoUM
pragma channel DecisionToUM: DtoUM
pragma channel UMtoDecision: UMtoD
pragma channel fromUser: FROMUSER

-- +==========================================================+
-- | Independent events observed/produced                     |
-- | by the Upgrade Manager                                   |
-- +==========================================================+

pragma channel convergenceDetected
pragma channel convergenceTimeout
pragma channel raiseBaselinePrio
pragma channel raiseComplexPrio
pragma channel lowerBaselinePrio
pragma channel lowerComplexPrio

-- +==========================================================+
-- | Process Definitions                                      |
-- +==========================================================+

--
-- Simplex architecture:
--
SIMPLEX =
 UPGRADEMGR
 [| {| UMtoComplex, ComplexToUM, UMtoBaseline, BaselineToUM,
 DecisionToUM, UMtoDecision |} |]
 ((DECISION
 [| {| fromComplex, fromBaseline, fromSafety |} |]
 (COMPLEX ||| BASELINE ||| SAFETY))
 [| {| fromIO, toIO |} |]
 PHYSICALIO)

--
-- Upgrade Manager Component:
--
UPGRADEMGR = WILLINGTOSTARTBASELINE

WILLINGTOSTARTBASELINE =
 fromUser.startBaseline -> UMtoBaseline!start ->
 BaselineToUM.initDone -> raiseBaselinePrio ->
 (convergenceDetected -> UMtoBaseline!enableOutput ->
 UMtoDecision!baselineRunning -> WILLINGTOSTARTCOMPLEX
 |~|
 convergenceTimeout -> KILLBASELINE)

WILLINGTOSTARTCOMPLEX =
 fromUser.startComplex -> STARTCOMPLEX
 []
```

```
 DecisionToUM.killBaseline -> KILLBASELINE
 []
 fromUser.killBaseline -> UMtoDecision!userKillBaseline ->
KILLBASELINE

KILLBASELINE =
 lowerBaselinePrio -> UMtoBaseline!kill ->
 BaselineToUM.dead -> WILLINGTOSTARTBASELINE

STARTCOMPLEX =
 UMtoComplex!start -> ComplexToUM.initDone -> raiseComplexPrio
->
 (convergenceTimeout -> KILLCOMPLEX
 |~|
 convergenceDetected -> UMtoComplex!enableOutput ->
 UMtoDecision!complexRunning ->
 (fromUser.killComplex -> UMtoDecision!userKillComplex ->
KILLCOMPLEX
 []
 DecisionToUM.killComplex -> KILLCOMPLEX))

KILLCOMPLEX =
 lowerComplexPrio -> UMtoComplex!kill -> ComplexToUM.dead ->
 WILLINGTOSTARTCOMPLEX

--
-- Decision Component:
--
DECISION = SAFETYLOOP

SAFETYLOOP =
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 SAFETYLOOP
 []
 fromIO.safe -> (UMtoDecision.baselineRunning -> BASELINESAFE
 []
 fromSafety.cntrlout -> toIO!cntrlout ->
 SAFETYLOOP)

BASELINELOOP =
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 DecisionToUM!killBaseline -> SAFETYLOOP
 []
 fromIO.safe -> (UMtoDecision.complexRunning -> COMPLEXSAFE
 []
 BASELINESAFE)
 []
 UMtoDecision.userKillBaseline -> SAFETYLOOP

BASELINESAFE =
 fromBaseline.cntrlout -> toIO!cntrlout -> BASELINELOOP
 []
 ([] x: {illegalout, responseTimeout} @
 fromBaseline.x -> DecisionToUM!killBaseline ->
 fromSafety.cntrlout -> toIO!cntrlout -> SAFETYLOOP)

COMPLEXLOOP =
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
```

```
 DecisionToUM!killComplex -> TEMPSAFETYLOOP
 []
 fromIO.safe -> COMPLEXSAFE
 []
 UMtoDecision.userKillComplex -> BASELINELOOP

COMPLEXSAFE =
 fromComplex.cntrlout -> toIO!cntrlout -> COMPLEXLOOP
 []
 ([] x: {illegalout, responseTimeout} @
 fromComplex.x -> DecisionToUM!killComplex ->
 BASELINESAFE)

TEMPSAFETYLOOP =
 fromIO.safe -> BASELINESAFE
 []
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 TEMPSAFETYLOOP

--
-- Safety Component:
--
SAFETY = fromSafety!cntrlout -> SAFETY

--
-- Baseline Component:
--
BASELINE =
 UMtoBaseline.start -> BaselineToUM!initDone ->
 (UMtoBaseline.enableOutput -> BASELINERUNNING
 []
 UMtoBaseline.kill -> BaselineToUM!dead -> BASELINE)

BASELINERUNNING =
 (|~| x: {cntrlout, illegalout, responseTimeout} @
 fromBaseline!x -> BASELINERUNNING)
 []
 UMtoBaseline.kill -> BaselineToUM!dead -> BASELINE

--
-- Complex Component:
--
COMPLEX =
 UMtoComplex.start -> ComplexToUM!initDone ->
 (UMtoComplex.enableOutput -> COMPLEXRUNNING
 []
 UMtoComplex.kill -> ComplexToUM!dead -> COMPLEX)

COMPLEXRUNNING =
 (|~| x: {cntrlout, illegalout, responseTimeout} @
 fromComplex!x -> COMPLEXRUNNING)
 []
 UMtoComplex.kill -> ComplexToUM!dead -> COMPLEX

--
-- Physical I/O component:
--
PHYSICALIO = INPUT ||| OUTPUT
```

```
-- Take input from the Plant:
INPUT = fromPlant?status -> fromIO!status -> INPUT

-- Send output to the Plant:
OUTPUT = toIO?command -> toPlant!command -> OUTPUT


-- +==========================================================+
-- | Properties to verify                                      |
-- +==========================================================+

-- Auxiliary definitions:
SIGMA = {| fromUser, fromPlant, toPlant, fromIO, toIO,
 fromComplex, fromBaseline, fromSafety,
 UMtoComplex, UMtoBaseline, ComplexToUM,
 BaselineToUM, DecisionToUM, UMtoDecision,
 convergenceDetected,
 convergenceTimeout,
 raiseBaselinePrio,
 raiseComplexPrio,
 lowerBaselinePrio,
 lowerComplexPrio |}

pragma channel e


-- Property 1. Deadlock-free.
--            |- P1 is failure-divergence-refined by P1SIMPLEX
-- where:

P1 = e -> P1

P1SIMPLEX = identify (SIGMA, e, SIMPLEX)


-- Property 2. Control commands are sent to the plant
-- infinitely often. That
-- is, it is never the case that from a given instant commands
-- are not sent
-- to the plant anymore.
--            |- P2 is failure-divergence-refined by P2SIMPLEX
-- where:

P2 = |~| x: TOPLANT @ toIO!x -> P2

P2SIMPLEX = SIMPLEX \ diff (SIGMA, {| toIO |})


-- Property 3. It is never the case that there are two
-- consecutive readings
-- from the plant without an output command between them. In
-- other words,
-- after reading the status of the plant, and before performing
-- the next
-- reading, a command has to be sent to the plant. (This could
-- be a way of
-- detecting that a deadline was missed)
```

```
--                          |- P3 is trace-refined by P3SIMPLEX
-- where:

P3 = fromIO?x -> toIO!cntrlout -> P3

P3SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO |})


-- Property 4. After unsafe condition the plant is immediately
-- controlled by
-- safety, and as long as it is unsafe it remains in control
-- by safety:
--                          |- P4 is trace-refined by P4SIMPLEX
-- where:

P4 = fromIO.unsafe -> fromSafety.cntrlout -> toIO.cntrlout ->
P4
 |~|
 fromIO.safe -> (|~| c: { fromComplex, fromBaseline, fromSafety
} @
 c.cntrlout -> toIO.cntrlout -> P4)

P4SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO,
fromSafety.cntrlout,
 fromComplex.cntrlout,
 fromBaseline.cntrlout |})


-- Property 5. Whenever the plant is in safe state and is being
-- controlled
-- by the complex controller, if the complex controller does
-- not
-- produce an output on time (i.e. it misses its deadline or
-- falls into an
-- infinite loop) or if the output is illegal, the control of
-- the device is
-- passed to the baseline controller (or to the safety
-- controller, in case
-- the baseline controller fails).
--                          |- P5 is trace-refined by P5SIMPLEX
-- where:

P5 = fromIO.safe ->
 ((|~| x: {illegalout, responseTimeout} @
 fromComplex.x -> (|~| c: {fromBaseline, fromSafety} @
 c.cntrlout -> P5))
 |~|
 (|~| c: {fromComplex, fromBaseline, fromSafety} @ c.cntrlout
-> P5)
 )
 |~|
 fromIO.unsafe -> fromSafety.cntrlout -> P5

P5SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, fromComplex,
 fromBaseline.cntrlout,
 fromSafety.cntrlout |})
```

```
-- Property 6. Whenever Complex is started, Baseline has to be
-- running, and
-- it never happens that there is more than one Baseline or
-- more than one
-- Complex running.
--                        |- P6 is trace-refined by P6SIMPLEX
-- where:

P6 = UMtoBaseline.start -> P6AUX
P6AUX = UMtoComplex.start -> UMtoComplex.kill -> P6AUX
 |~|
 UMtoBaseline.kill -> P6

P6SIMPLEX = SIMPLEX \ diff (SIGMA, { UMtoBaseline.start,
UMtoBaseline.kill,
 UMtoComplex.start, UMtoComplex.kill })
```

A.3 Final Attempt at a CSP Model with Upgrade Manager

```
-- +=========================================================+
-- | CSP Model of the overall dynamic behavior for the       |
-- | Uni-processor Simplex Architecture.                     |
--                                                           |
-- | Author: Jose German Rivera                              |
-- | Alejandro Andres Danylyszyn                             |
--                                                           |
-- | Created: 10/4/95 - Initial version: Upgrade Manager     |
-- | not included.                                           |
-- | Updated: 11/12/95 - Included Upgrade Manager. It has a  |
-- | problem: properties 1-2 are not                         |
-- | satisfied due to the fact that some                     |
-- | special cases of starting Complex                       |
-- | or killing Baseline/Complex were                        |
-- | not considered. Property 6 added.                       |
-- | Updated: 11/24/95 - Property 1 now is satisfied, but    |
-- | property 2 is not completely                            |
-- | satisfied (check2 succeeds but not                      |
-- | check1); reason being a divergence                      |
-- | caused by lack of fairness in FDR.                      |
-- +=========================================================+

-- +=========================================================+
-- | Channel Alphabets                                       |
-- +=========================================================+

-- Events received from the controlled plant:
FROMPLANT = {safe, unsafe}

-- Events sent to the controlled plant:
TOPLANT = {cntrlout}

-- Events generated by the controllers:
```

```
CNTRLEVT = {cntrlout, illegalout, responseTimeout}

-- Events sent from the Upgrade Manager to the controllers:
UMtoCTRL = {start, enableOutput, kill}

-- Events sent from the controllers to the Upgrade Manager:
CTRLtoUM = {initDone, dead}

-- Events sent from Decision to the Upgrade Manager:
DtoUM = {killBaseline, killComplex}

-- Events sent from the Upgrade Manager to Decision:
UMtoD = {baselineRunning, complexRunning, userKillBaseline,
 userKillComplex}

-- Events Received from the user:
FROMUSER = {startBaseline, startComplex, killBaseline,
 killComplex}

-- +=========================================================+
-- | Channel Declarations                                    |
-- +=========================================================+

pragma channel fromPlant, fromIO: FROMPLANT
pragma channel toPlant, toIO: TOPLANT
pragma channel fromComplex, fromBaseline, fromSafety: CNTRLEVT
pragma channel UMtoComplex, UMtoBaseline: UMtoCTRL
pragma channel ComplexToUM, BaselineToUM: CTRLtoUM
pragma channel DecisionToUM: DtoUM
pragma channel UMtoDecision: UMtoD
pragma channel fromUser: FROMUSER

-- +=========================================================+
-- | Independent events observed/produced                    |
-- | by the Upgrade Manager                                  |
-- +==========================================================
====+

pragma channel convergenceDetected
pragma channel convergenceTimeout
pragma channel raiseBaselinePrio
pragma channel raiseComplexPrio
pragma channel lowerBaselinePrio
pragma channel lowerComplexPrio

-- +=========================================================+
-- | Process Definitions                                     |
-- +=========================================================+

--
-- Simplex architecture:
--
SIMPLEX =
 UPGRADEMGR
 [| {| UMtoComplex, ComplexToUM, UMtoBaseline, BaselineToUM,
 DecisionToUM, UMtoDecision |} |]
 ((DECISION
 [| {| fromComplex, fromBaseline, fromSafety |} |]
```

```
 (COMPLEX ||| BASELINE ||| SAFETY))
 [| {| fromIO, toIO |} |]
 PHYSICALIO)

--
-- Upgrade Manager Component:
--
UPGRADEMGR = WILLINGTOSTARTBASELINE

WILLINGTOSTARTBASELINE =
 fromUser.startBaseline -> UMtoBaseline!start ->
 BaselineToUM.initDone -> raiseBaselinePrio ->
 (convergenceDetected -> UMtoBaseline!enableOutput ->
 UMtoDecision!baselineRunning -> WILLINGTOSTARTCOMPLEX
 |~|
 convergenceTimeout -> KILLBASELINE)

WILLINGTOSTARTCOMPLEX =
 fromUser.startComplex -> STARTCOMPLEX
 []
 DecisionToUM.killBaseline -> KILLBASELINE
 []
 fromUser.killBaseline -> (UMtoDecision!userKillBaseline ->
KILLBASELINE
 []
 DecisionToUM.killBaseline -> KILLBASELINE)

KILLBASELINE =
 lowerBaselinePrio -> UMtoBaseline!kill ->
 BaselineToUM.dead -> WILLINGTOSTARTBASELINE

STARTCOMPLEX =
 UMtoComplex!start -> ComplexToUM.initDone -> raiseComplexPrio
->
 (convergenceTimeout -> KILLCOMPLEX
 |~|
 convergenceDetected -> UMtoComplex!enableOutput ->
 (UMtoDecision!complexRunning ->
 (fromUser.killComplex ->
 (UMtoDecision!userKillComplex -> KILLCOMPLEX
 []
 DecisionToUM.killComplex -> KILLCOMPLEX)
 []
 DecisionToUM.killComplex -> KILLCOMPLEX)
 []
 DecisionToUM.killBaseline -> lowerComplexPrio ->
 UMtoComplex!kill -> ComplexToUM.dead -> KILLBASELINE))

KILLCOMPLEX =
 lowerComplexPrio -> UMtoComplex!kill -> ComplexToUM.dead ->
 WILLINGTOSTARTCOMPLEX

--
-- Decision Component:
--
DECISION = SAFETYLOOP

SAFETYLOOP =
```

```
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 SAFETYLOOP
 []
 fromIO.safe -> (UMtoDecision.baselineRunning -> BASELINESAFE
 []
 fromSafety.cntrlout -> toIO!cntrlout ->
 SAFETYLOOP)

BASELINELOOP =
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 DecisionToUM!killBaseline -> SAFETYLOOP
 []
 fromIO.safe -> (UMtoDecision.complexRunning -> COMPLEXSAFE
 []
 BASELINESAFE)
 []
 UMtoDecision.userKillBaseline -> SAFETYLOOP

BASELINESAFE =
 fromBaseline.cntrlout -> toIO!cntrlout -> BASELINELOOP
 []
 ([] x: {illegalout, responseTimeout} @
 fromBaseline.x -> DecisionToUM!killBaseline ->
 fromSafety.cntrlout -> toIO!cntrlout -> SAFETYLOOP)

COMPLEXLOOP =
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 DecisionToUM!killComplex -> TEMPSAFETYLOOP
 []
 fromIO.safe -> COMPLEXSAFE
 []
 UMtoDecision.userKillComplex -> BASELINELOOP

COMPLEXSAFE =
 fromComplex.cntrlout -> toIO!cntrlout -> COMPLEXLOOP
 []
 ([] x: {illegalout, responseTimeout} @
 fromComplex.x -> DecisionToUM!killComplex ->
 BASELINESAFE)

TEMPSAFETYLOOP =
 fromIO.safe -> BASELINESAFE
 []
 fromIO.unsafe -> fromSafety.cntrlout -> toIO!cntrlout ->
 TEMPSAFETYLOOP

--
-- Safety Component:
--
SAFETY = fromSafety!cntrlout -> SAFETY

--
-- Baseline Component:
--
BASELINE =
 UMtoBaseline.start -> BaselineToUM!initDone ->
 (UMtoBaseline.enableOutput -> BASELINERUNNING
 []
```

```
 UMtoBaseline.kill -> BaselineToUM!dead -> BASELINE)

BASELINERUNNING =
 (|~| x: {cntrlout, illegalout, responseTimeout} @
 fromBaseline!x -> BASELINERUNNING)
 []
 UMtoBaseline.kill -> BaselineToUM!dead -> BASELINE

--
-- Complex Component:
--
COMPLEX =
 UMtoComplex.start -> ComplexToUM!initDone ->
 (UMtoComplex.enableOutput -> COMPLEXRUNNING
 []
 UMtoComplex.kill -> ComplexToUM!dead -> COMPLEX)

COMPLEXRUNNING =
 (|~| x: {cntrlout, illegalout, responseTimeout} @
 fromComplex!x -> COMPLEXRUNNING)
 []
 UMtoComplex.kill -> ComplexToUM!dead -> COMPLEX

--
-- Physical I/O component:
--
PHYSICALIO = INPUT ||| OUTPUT

-- Take input from the Plant:
INPUT = fromPlant?status -> fromIO!status -> INPUT

-- Send output to the Plant:
OUTPUT = toIO?command -> toPlant!command -> OUTPUT


-- +========================================================+
-- | Properties to verify                                   |
-- +========================================================+

-- Auxiliary definitions:
SIGMA = {| fromUser, fromPlant, toPlant, fromIO, toIO,
 fromComplex, fromBaseline, fromSafety,
 UMtoComplex, UMtoBaseline, ComplexToUM,
 BaselineToUM, DecisionToUM, UMtoDecision,
 convergenceDetected,
 convergenceTimeout,
 raiseBaselinePrio,
 raiseComplexPrio,
 lowerBaselinePrio,
 lowerComplexPrio |}

pragma channel e


-- Property 1. Deadlock-free.
--           |- P1 is failure-divergence-refined by P1SIMPLEX
-- where:
```

```
P1 = e -> P1

P1SIMPLEX = identify (SIGMA, e, SIMPLEX)

-- To check this property use: Check1 "P1" "P1SIMPLEX";


-- Property 2. Control commands are sent to the plant
-- infinitely often. That
-- is, it is never the case that from a given instant commands
-- are not sent
-- to the plant anymore.
--           |- P2 is failure-divergence-refined by P2SIMPLEX
-- where:

P2 = |~| x: TOPLANT @ toIO!x -> P2

P2SIMPLEX = SIMPLEX \ diff (SIGMA, {| toIO |})

-- To check this property use: Check1 "P2" "P2SIMPLEX";


-- Property 3. It is never the case that there are two
-- consecutive readings
-- from the plant without an output command between them. In
-- other words,
-- after reading the status of the plant, and before performing
-- the next
-- reading, a command has to be sent to the plant. (This could
-- be a way of
-- detecting that a deadline was missed)
--                        |- P3 is trace-refined by P3SIMPLEX
-- where:

P3 = fromIO?x -> toIO!cntrlout -> P3

P3SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO |})

-- To check this property use: CheckTrace "P3" "P3SIMPLEX";


-- Property 4. After unsafe condition the plant is immediately
-- controlled by
-- safety, and as long as it is unsafe it remains in control
-- by safety:
--                        |- P4 is trace-refined by P4SIMPLEX
-- where:

P4 = fromIO.unsafe -> fromSafety.cntrlout -> toIO.cntrlout ->
P4
 |~|
 fromIO.safe -> (|~| c: { fromComplex, fromBaseline, fromSafety
} @
 c.cntrlout -> toIO.cntrlout -> P4)

P4SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, toIO,
fromSafety.cntrlout,
```

```
 fromComplex.cntrlout,
 fromBaseline.cntrlout |})

-- To check this property use: CheckTrace "P4" "P4SIMPLEX";


-- Property 5. Whenever the plant is in safe state and is being
-- controlled
-- by the complex controller, if the complex controller does
-- not
-- produce an output on time (i.e. it misses its deadline or
-- falls into an
-- infinite loop) or if the output is illegal, the control of
-- the device is
-- passed to the baseline controller (or to the safety
-- controller, in case
-- the baseline controller fails).
--                        |- P5 is trace-refined by P5SIMPLEX
-- where:

P5 = fromIO.safe ->
 ((|~| x: {illegalout, responseTimeout} @
 fromComplex.x -> (|~| c: {fromBaseline, fromSafety} @
 c.cntrlout -> P5))
 |~|
 (|~| c: {fromComplex, fromBaseline, fromSafety} @ c.cntrlout
-> P5)
 )
 |~|
 fromIO.unsafe -> fromSafety.cntrlout -> P5

P5SIMPLEX = SIMPLEX \ diff (SIGMA, {| fromIO, fromComplex,
 fromBaseline.cntrlout,
 fromSafety.cntrlout |})

-- To check this property use: CheckTrace "P5" "P5SIMPLEX";


-- Property 6. Whenever Complex is started, Baseline has to be
-- running, and
-- it never happens that there is more than one Baseline or
-- more than one
-- Complex running.
--                        |- P6 is trace-refined by P6SIMPLEX
-- where:

P6 = UMtoBaseline.start -> P6AUX
P6AUX = UMtoComplex.start -> UMtoComplex.kill -> P6AUX
 |~|
 UMtoBaseline.kill -> P6

P6SIMPLEX = SIMPLEX \ diff (SIGMA, { UMtoBaseline.start,
UMtoBaseline.kill,
 UMtoComplex.start, UMtoComplex.kill })

-- To check this property use: CheckTrace "P6" "P6SIMPLEX";
```

References

[Allen 94] Allen, Robert & Garlan, David. “Formalizing Architectural Connection,” 71-80. *Proceedings of the Sixteenth International Conference on Software Engineering*. Sorrento, Italy, May 16-21, 1994. Long Beach, CA: IEEE Computer Society Press, 1994.

[FDR 92] *Failures Divergence Refinement: User Manual and Tutorial*, Version 1.3. Oxford, England: Formal Systems (Europe) Ltd., 1992.

[Rajkumar 95] Rajkumar, Ragunathan; Gagliardi, Michael; & Sha, Lui. “The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation,” 66-75. *Proceedings of the First IEEE Real-Time Technology and Applications Symposium*. Los Alamitos, CA, May 15-17, 1995. Long Beach, CA: IEEE Computer Society Press.

[Hoare 85] Hoare, C.A.R. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall International, 1985.

[Potter 91] Potter, Ben; Sinclair, Jane; & Till, David. *An Introduction to Formal Specification and Z*. Englewood Cliffs, NJ: Prentice Hall International, 1991.

[Sha 95] Sha, Lui; Gagliardi, Michael; & Rajkumar, Ragunathan. “Analytic Redundancy: A Foundation for Evolvable Dependable Systems,” *Proceedings of the Second ISSAT International Conference on Reliability and Quality of Design*. Orlando, FL, March 8-10, 1995. International Society of Science and Applied Technologies, 1995.