Formalizing the Uni-processor Simplex Architecture

José Germán Rivera
Alejandro Andrés Danylyszyn

CMU-CS-95-224
December 15, 1995

School of Computer Science
Carnegie Mellon University

Abstract

Simplex is a software architecture for dependable and evolvable process-control systems developed by the Software Engineering Institute. Our project consisted in creating a formal specification of this architecture, and analyzing its safety and liveness properties.

We developed a CSP model to describe the overall dynamic behavior of the Simplex architecture, that 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.

The research reported here 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.

Keywords: software architecture; system behavior; formal specification; model checking; software engineering; Simplex; CSP; Wright; FDR.

Table of Contents

1. UNI-PROCESSOR SIMPLEX 1

2. SYSTEM BEHAVIOR 3

2.1 ASSUMPTIONS 3
2.2 ABSTRACTION TECHNIQUES 4
2.3 FDR MODEL CHECKING GUIDELINES 5
2.4 DEFINITIONS 6
2.5 PRELIMINARY MODEL 8
2.6 FINAL MODEL 10
2.6.1 Process definitions 10
2.6.2 Properties verified 13
2.6.3 FDR verification results 15
2.6.4 Corrections to the model 15

3. SOFTWARE ARCHITECTURE 17

3.1 GRAPHICAL REPRESENTATION 17
3.2 WRIGHT SPECIFICATION 18
3.2.1 Process Types 20
3.2.2 DistributionTag connector 21
3.2.3 Procedure Call connector 23
3.2.4 Upgrade Manager component 23
3.2.5 Decision component 26
3.2.6 Safety Controller component 30
3.2.7 Untrusted Controller component 30
3.2.8 Physical I/O component 31

4. CONCLUSIONS 33

5. ACKNOWLEDGMENTS 34

6. REFERENCES 34

APPENDIX 35

List of Tables

TABLE 1- SIMPLEX EVENTS 7
TABLE 2- CHANNEL ALPHABETS 7

List of Figures

FIGURE 1: SIMPLEX CONTEXT DIAGRAM 3
FIGURE 2: PRELIMINARY MODEL 8
FIGURE 3: FINAL MODEL 10
FIGURE 4: SOFTWARE ARCHITECTURE EXCLUDING CONNECTIONS TO THE UPGRADE MANAGER 17
FIGURE 5: SOFTWARE ARCHITECTURE SHOWING CONNECTIONS TO THE UPGRADE MANAGER 17

Uni-processor Simplex Architecture

This paper presents the uni-processor simplex architecture, the modeling of the system's behavior, the verification of desired properties, and the system's software architecture.

The architecture and its requirements are described in section 1. In section 2 we present two models of the system behavior and the properties verified. The abstraction techniques and assumptions are also detailed for the reader to understand the models and the scope of this work. In section 3 we present an architectural model of Simplex. The summary and conclusions of this work are presented in section 4.

1. Uni-processor Simplex

The Simplex architecture is a family of high level application development platforms that has been designed to support the on-line evolution of software intensive systems specialized to a specific domain. From the perspective of application developers, it is a collection of on-line 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 on-line software evolution. The Simplex architecture is still a technology that is 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.

The system behavior is based on the use of a simple and reliable unit as the fall back controller for an advanced, but yet to be proven, controller unit. This is known as *analytic redundancy*[4]. 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). However, the trajectories of the advanced controller must be within the controllable states of the fall back controller. In other words, the state space of the advanced controller must be a subset of the fall back controller's controllable states, although they may have very different behaviors in response to a command.

The uni-processor version of the Simplex architecture has six units: *Physical I/O*, *Decision*, *Safety*, *Baseline*, *Complex* 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, output switching from one component to another) and is responsible for the replacement transaction described later.

The *Decision* unit is ultimately responsible for which of the three controllers (Complex, Baseline or Safety) 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 Safety 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.

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. During this replacement transaction, state information (i.e., 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 on-line. Hence, the replacement transaction of a single replacement unit is carried out in stages:

- The new replacement unit is created.
- New input and any state information are 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.
- The upgrade transaction manager waits for the output of the new unit to synchronize or converge to a stable point.
- Finally, the output of the old unit is turned off and the new unit is turned on. The old unit can now be destroyed.

2. System behavior

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

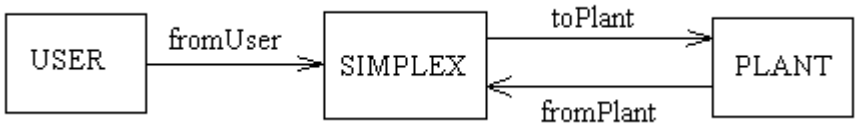

Figure 1: Simplex context diagram

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.

2.1 Assumptions

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

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:

- The hard real-time constraints can be abstracted and do not need to be modeled to define the software architecture. See the abstraction techniques for details on how the timing constraints were handled.

- 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 whichever 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.

2.2 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 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.

2.3 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:

$P \sqsubseteq_T Mp$
(in FDR syntax: `CheckTrace "P" "Mp"`)

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

$P \sqsubseteq_{FD} Mp$
(in FDR syntax: `Check1 "P" "Mp"`)

2.4 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.

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.
`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.

Event	Description
safe	The Simplex software detects that the plant is operating inside the safety region (operational states [4])
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 [4])
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.

Table 1- Simplex events

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

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.

Table 2- Channel alphabets

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
```

2.5 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 2.5.

The model is presented in FDR syntax [3], 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:

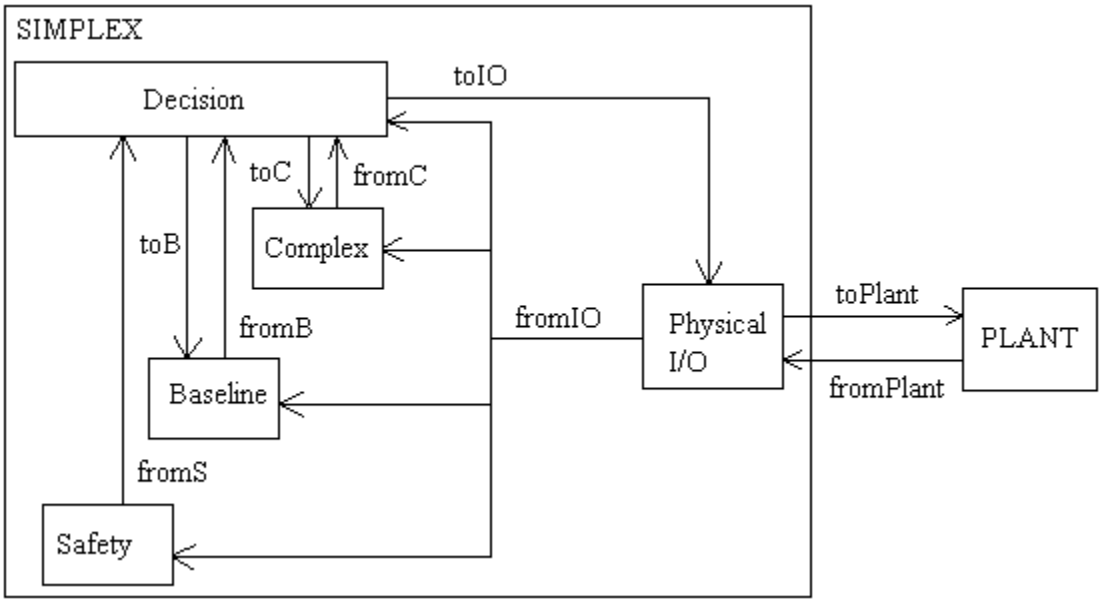

Figure 2: Preliminary model

• ***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
```

2.6 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.

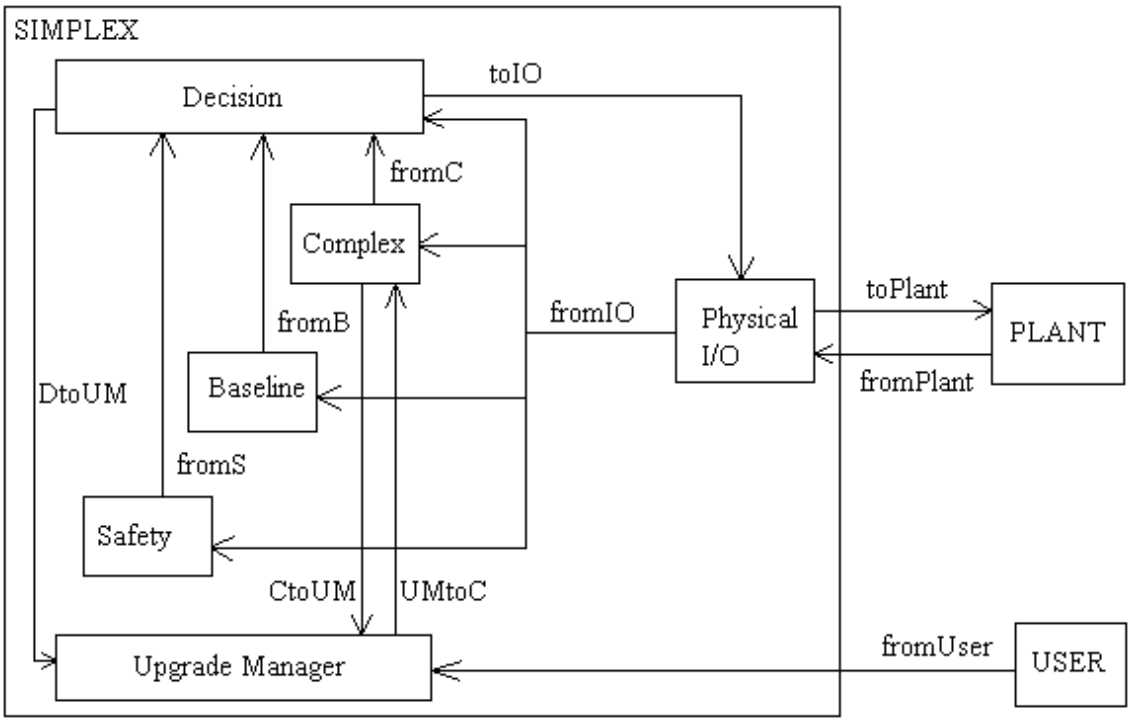

Figure 3: Final model

Not all the channels are shown in the diagram to increase readability. `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 actually `Safety` is embedded in `Decision`, it does not have explicit connections with the `Upgrade Manager`.

2.6.1 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
```

2.6.2 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 })
```

2.6.3 FDR verification results

The model presented above has a problem: while properties 3 through 6 are satisfied, properties 1 and 2 are not due to special 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.startBaseline,UMtoBaseli
ne.start,BaselineToUM.initDone,raiseBaselinePrio,tau,convergenceDetected,UMtoB
aseline.enableOutput,UMtoDecision.baselineRunning,fromUser.killBaseline,tau,fr
omBaseline.illegalout,tau> refuses
{|{|fromPlant,fromIO,toPlant,toIO,fromComplex,fromBaseline,fromSafety,UMtoComp
lex,UMtoBaseline,ComplexToUM,BaselineToUM,DecisionToUM,UMtoDecision,fromUser,c
onvergenceDetected,convergenceTimeout,raiseBaselinePrio,raiseComplexPrio,lower
BaselinePrio,lowerComplexPrio,tick|}|}
```

Our interpretation of this result from FDR is that it could happen that 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.

2.6.4 Corrections to the model

In order for the model to satisfy property 1 we have 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,raiseBaseline
Prio,tau,convergenceTimeout,lowerBaselinePrio,UMtoBaseline.kill,BaselineToUM.d
ead>
```

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 on section 2.1). 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.

3. Software architecture

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

3.1 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:

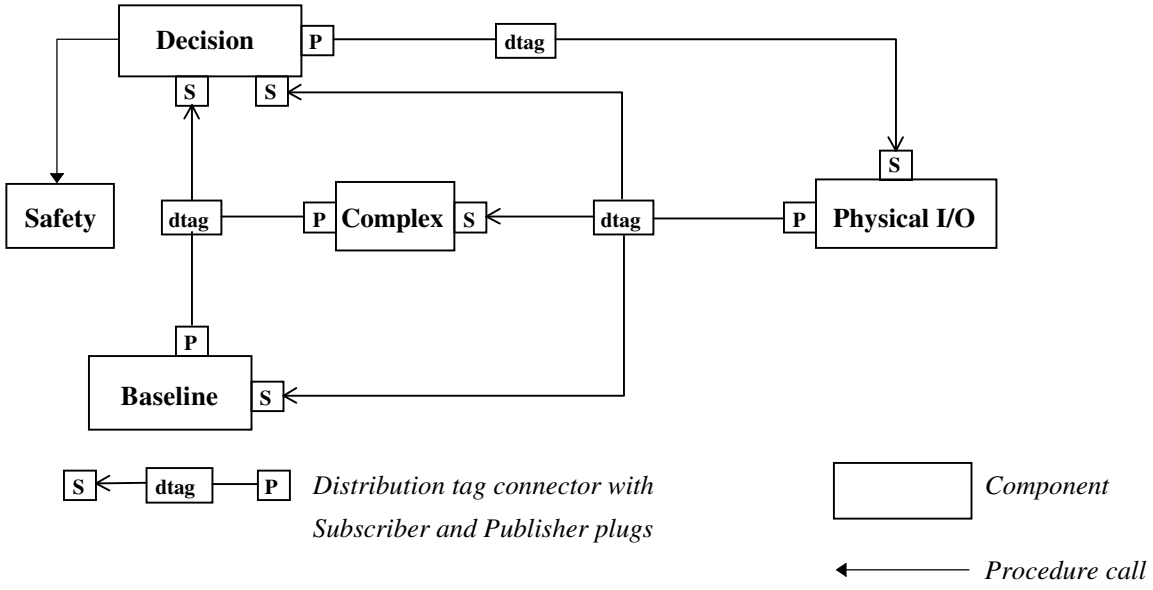

Figure 4: Software architecture excluding connections to the Upgrade Manager

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

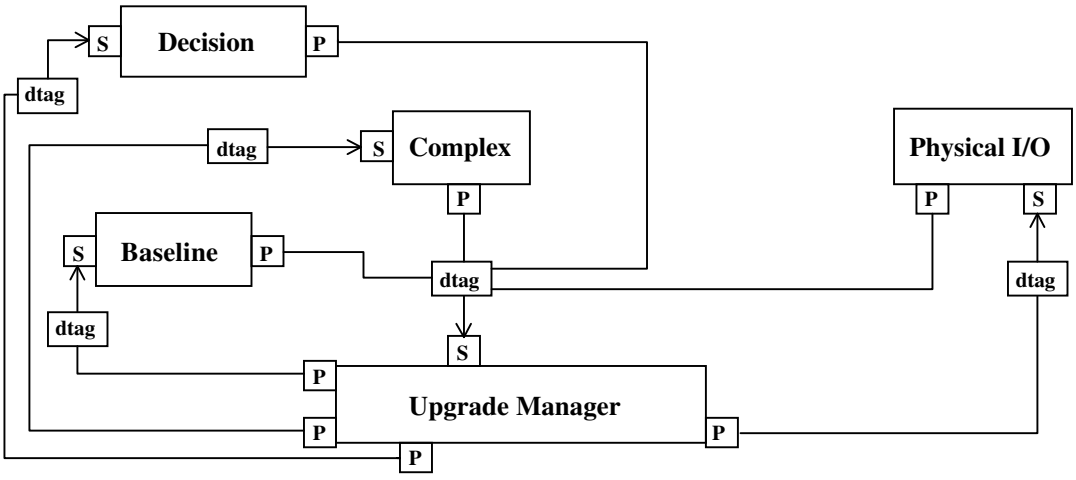

Figure 5: Software architecture showing connections to the Upgrade Manager

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.

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.

3.2 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 [3] was used, since it allows 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 Publisher$_{1\ ..\ numPublishers}$

Role Subscriber$_{1\ ..\ numSubscribers}$

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
$upgradeManagerOutTag_{1..4}$: 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** $upgradeManagerOutTag_1$.Publsiher
$upgradeManager.ToReplacementUnit_{Baseline}$ **as** $upgradeManagerOutTag_2$.Publsiher
$upgradeManager.ToReplacementUnit_{Complex}$ **as** $upgradeManagerOutTag_3$.Publsiher
$upgradeManager.ToReplacementUnit_{PhysicalIO}$ **as** $upgradeManagerOutTag_4$.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.

3.2.1 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 ⌈⌉ §))

3.2.2 DistributionTag connector

The DistributionTag connector is a multi-cast message passing mechanism that allows a group of components (publishers) to advertise messages and other group of components (subscribers) to receive the published messages [5]. 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 $\text{Publisher}_{1\ ..\ \text{numPublishers}}$ = PublisherLifeCycle

Role $\text{Subscriber}_{1\ ..\ \text{numSubscribers}}$ = SubscriberLifeCycle

Glue =

$(\text{ServePublishers}_{\{\}}\ (\text{numPublishers})$
$\ \|\|\|$
$\text{ServeSubscribers}_{\{\}}\ (\text{numSubscribers}))$
$\|\|_{\{\text{enqueue, dequeue, timeout}\}}$
MsgQueues (numPublishers, numSubscribers)

where

$\text{ServePublishers}_{\text{PublishersSet}}$ (numPublishers) =
$(\forall\ p: (1..\text{numPublishers}) - \text{PublishersSet}\ [\,]$
$\text{Publisher}_p.\text{getSendAccess} \rightarrow$
$\text{ServePublishers}_{\text{PublishersSet} \cup \{p\}}\ (\text{numPublishers}))$
$[\,]$
$(\forall\ p: \text{PublishersSet}\ [\,]$

$\text{Publisher}_p.\text{send?msg} \rightarrow \text{enqueue}_p\text{!msg} \rightarrow$

ServePublishers$_{PublishersSet}$ (numPublishers))
[]
(∀ p: PublishersSet []
Publisher$_p$.releaseSendAccess →
ServePublishers$_{PublishersSet - \{p\}}$ (numPublishers))

ServeSubscribers$_{SubscribersSet}$ (numSubscribers) =
(∀ s: (1..numSubscribers) - SubscribersSet []
Subscriber$_s$.subscribe →
ServeSubscribers$_{SubscribersSet \cup \{s\}}$ (numSubscribers))
[]
(∀ s: SubscribersSet []
Subscriber$_s$.receive → dequeue$_s$?msg → Subscriber$_s$.return!msg →
ServeSubscribers$_{SubscribersSet}$ (numSubscribers))
[]
(∀ s: SubscribersSet []
Subscriber$_s$.receiveWithTimeout →
(dequeue$_s$?msg → Subscriber$_s$.return!msg →
ServeSubscribers$_{SubscribersSet}$ (numSubscribers))
[]
timeout$_s$ → Subscriber$_s$.timeout →
ServeSubscribers$_{SubscribersSet}$ (numSubscribers))
[]
(∀ s: SubscribersSet []
Subscriber$_s$.unsubscribe →
ServeSubscribers$_{SubscribersSet - \{s\}}$ (numSubscribers))

MsgQueues (numPublishers, numSubscribers) =
(∀ s: (1..numSubscribers) ||$_{\{enqueue\}}$
Queue$_{s, <>}$ (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}.

Queue$_{subscriber, buffer}$ (numPublishers) =
(∀ p: (1..numPublishers) []
enqueue$_p$?msg →
Queue$_{subscriber,}$ **priorityInsert** (buffer, p, msg) (numPublishers))
[]
((dequeue$_{subscriber}$!(**head** buffer) → Queue$_{subscriber,}$ **tail** buffer (numPublishers))
if buffer ≠ <> **else**
(timeout$_{subscriber}$ → Queue$_{subscriber, buffer}$ (numPublishers)
⌈⌉

$Queue_{subscriber,\ buffer}$ (numPublishers)))

The **priorityInsert** function can be defined in Z [6] 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)

3.2.3 Procedure Call connector

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

Connector ProcedureCall

Role Caller = $\overline{\text{invoke!x}}$ → return?y → (Caller ⌈⌉ §)

Role Declarer = invoke?x → $\overline{\text{return!y}}$ → Declarer

Glue = Caller.invoke?x → $\overline{\text{Declarer.invoke!x}}$ →

Declarer.return?y → $\overline{\text{Caller.return!y}}$ → **Glue**

3.2.4 Upgrade Manager component

The Upgrade Manager component provides the on-line 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 to shut-down 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)
```

3.2.5 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 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 takes temporarily 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. If 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 →
(FromPhysicalIO.return.unsafe →
CallSafety.invoke → CallSafety.return?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

3.2.6 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**

3.2.7 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, by missing its deadline (producing its output not 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**

3.2.8 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

4. 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 3.2.2, 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.

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 2.5.4.

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 much 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 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).

An important aspect that was not addressed explicitly in our specification of the Simplex architecture was real time. The main reason for this was that the specification techniques and tools we used do not provide facilities to represent time explicitly. However, this issue should be addressed in the future, if our work is to be continued. Our specification should be enhanced to capture somehow the real-time characteristics of the architecture.

5. Acknowledgments

The authors would like to thank professor David Garlan for his constant supervision and guidance during the development of this work. His advise and review comments were very valuable, and gave us insight to define the strategy for specifying the Simplex architecture.

Equally important was the help of Chuck Weinstock, Michael Gagliardi and Lui Sha, from the Software Engineering Institute (SEI), to understand the Simplex architecture, and its uni-processor prototype. Their technical comments were very helpful for refining the models presented, and increasing our knowledge of the target application domain.

6. References

The following books and papers were used for the preparation of this essay:

[1] C.A.R. Hoare. *"Communicating Sequential Processes"*. Prentice Hall International, 1985.

[2] Robert Allen, David Garlan. *"Formalizing Architectural Connection"*. Proceedings of the 16th International Conference on Software Engineering, 1994.

[3] *"Failures Divergence Refinement, User Manual and Tutorial"*. Version 1.3. Formal Systems (Europe) Ltd., August 1993

[4] Lui Sha, Michael Gagliardi, and Ragunathan Rajkumar. *"Analytic Redundancy: A Foundation for Evolvable Dependable Systems"*. Software Engineering Institute, Carnegie Mellon University, November, 1994

[5] Ragunathan Rajkumar, Michael Gagliardi, and Lui Sha. *"The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation"*. Software Engineering Institute, Carnegie Mellon University, 1994

[6] Ben Potter, Jane Sinclair, and David Till. "An Introduction to Formal Specification and Z". Prentice Hall International, 1991.

Appendix