The Simplex Reference Model: Limiting Fault-Propagation due to Unreliable Components in Cyber-Physical System Architectures

Tanya L. Crenshaw, Elsa Gunter, C. L. Robinson, Lui Sha, P. R. Kumar
University of Illinois at Urbana-Champaign
{tcrensha, egunter, clrobnsn, lrs, prkumar}@uiuc.edu

Abstract

Cyber-Physical Systems are networked, component-based, real-time systems that control and monitor the physical world. We need software architectures that limit fault-propagation across unreliable components. This paper introduces our Simplex reference model which is distinguished by: a Plant being controlled in an external context, a Machine performing the control, a Domain Model that estimates the Plant state, and the Safety Requirements that must be met. The Simplex reference model assists with constructing CPS architectures which limit fault-propagation. We present a representative case study to highlight the ideas behind the model and our particular decomposition.

I. Introduction

Cyber-Physical Systems (CPS) comprise networked devices which monitor and control the physical world. These systems have three qualities of interest. First, their domains include critical infrastructures such as automobiles, manufacturing plants, and health-management networks. For these domains, safety is paramount. Second, they relate to the physical world in real-time; these systems are event- and deadline-driven. Third, CPS architects take advantage of the Commercial-Off-The-Shelf (COTS) component model. As a result, they gain the valuable expertise of third-party companies at a lower cost, but face the complexity of assembling large numbers of components they may know little about or may not be able to explicitly verify.

Because of these three characteristics, these systems must limit fault-propagation. A single component fault cannot compromise system safety, deadlines, or cause further mayhem in other components. Yet, given the large size of these systems, time and cost do not allow formal verification of every component. Moreover, third-party components may be too complex to be formally verifiable, or they may simply be untrusted. Our goal is to a develop reference model which assists with constructing CPS architectures which limit fault-propagation.

This paper presents our Simplex reference model. Our reference model articulates how one can limit fault-propagation in the presence of unreliable components. The original Simplex architecture [1], [2] was designed to provide fault-tolerant, dynamic upgrades for real-time systems. It consists of two controllers used to balance an inverted pendulum. The first is safe, simple and is verifiably correct. The second offers desirable features but is at times incorrect. Simplex combines these two algorithms in such a way that the system maintains safety while gaining as many desired features as possible. There have been a number of prototypes of the Simplex architecture: an inverted pendulum [1], [3], a diving controller [4], and a simulated auto-pilot [5]. Yet, Simplex has not been modeled such that it may be readily applied to other systems.

This rest of the paper is as follows. First, we present our Simplex reference model, useful for developers interested in deploying a Simplex architecture for their own systems. In particular, we present a logical framework for a "recoverable" state, and describe how the *Machine* uses this framework to uphold the *Safety Requirements*. Second, we provide two examples and one case study. For our case study, we use the Convergence Laboratory testbed at the University of Illinois at Urbana-Champaign [6]. We redesigned two subsystems using our Simplex reference model. In one subsystem, we used Simplex to improve the reliability of unreliable data. In another, we used Simplex to provide safe control. Combined, these two subsystems cooperate in such a way that they provide greater safety together than each could individually.

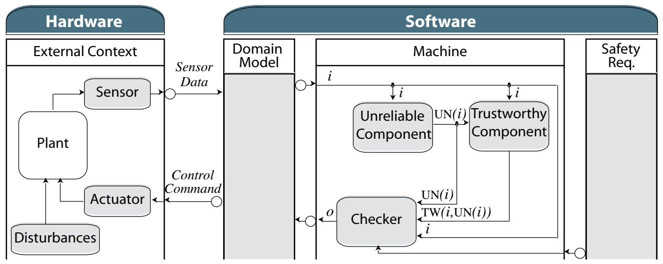

Fig. 1. The Simplex reference model.

II. The Simplex Reference Model

The Simplex reference model is designed to combine two redundant components or order to achieve a particular goal with the greatest possible performance while making safety a priority. On the one hand, an unreliable component is designed for high performance; it is a complex, possibly unverifiable component in our system that yields safe output most of the time. On the other hand, a trustworthy component is designed for safety. The trustworthy component has fewer features, but has been verified that it yields safe output all of the time. In this section, we describe the reference model we have developed for reasoning about Simplex.

Shown in Fig. 1, the Simplex reference model consists of the following decomposition:

1) External Context: Consists of all the physical devices external to the computing machine. This includes the *Plant* being controlled, as well as the *Environment* in which the *Plant* operates. The *Environment* includes *Disturbances* to the *Plant*, actuator dynamics, and sensor inaccuracies. The *External Context* contains no software, at least from the perspective of the development team. Perhaps there is a sensor that contains embedded code, but it is not under the discretion of development. A *Sensor* provides *Sensor Data* and an *Actuator* receives *Control Commands*.

2) Domain Model: Represents everything that is known about the *External Context* that can be used to estimate the behavior, or dynamics, of the *Plant*. It maintains an estimate of the current *Plant* state given *Sensor Data* and *Control Commands*. It can project the *Plant* state one time-step into the future. The *Domain Model* interprets the *Sensor Data* and sends it to the *Machine*. The *Machine* replies with an output which the *Domain Model* translates to a *Control Command* for the *Environment*.

3) Machine: Consists of all of the control logic used to generate control actions applied to the *Plant*. The *Machine* comprises multiple redundant subsystems designed to achieve the same minimal performance goals. A *Checker* chooses between the outputs of the two redundant components depending on the current *Plant* state.

4) Safety Requirements: A statement of what must always be true in the *Plant* to achieve safety. The *Safety Requirements* are embedded in the *Machine* so that it can choose an output from its redundant subsystems that will keep the *Plant* in a safe state, both now and in the future.

This decomposition we describe, notably the separation of the *Machine* functionality from the *Domain Model* is based on the principle of separation [7], [8]. To summarize, an optimal state estimate and optimal control action can be computed separately, and are still optimal when combined. Since our reference model depends on this separation result providing for such equivalent optimal control, it is not applicable to the CPS domain when the theorem is not valid [9].

We delve into greater detail for each entity in our Simplex reference model. Shown in Fig. 1, the *Machine* comprises an *Unreliable Component*, *Trustworthy Component*, and *Checker*, which we call UN, TW, and CHKR. Given interpreted *Sensor Data*, the CHKR selects output from the UN or the TW; preference is given to the UN to take advantage of its full features, but the output must always maintain the *Plant* in a safe state with respect to the *Safety Requirements* expressed in the CHKR.

Given the UN and the TW, the CHKR must be able to: i) Determine if the UN output will place the *Plant* in a state that satisfies the *Safety Requirements* at the current time. It must also determine if TW can continue to satisfy the requirements in the future; ii) Choose the TW output if the UN output places the *Plant* in a state that must eventually lead to a state that does not satisfy the *Safety Requirements*; iii) Choose the TW output in a timely manner such that the *Plant* is always in a safe state.

Suppose that our *Safety Requirement* is that a given object does not hit a particular obstacle. Just because the object is currently not hitting the obstacle at this moment does not mean that it is in a safe state. It may be moving at too high a velocity to stop before it hits the obstacle. The object can be "safe at the moment" and still not "always safe." Thus, we must differentiate between three kinds of *Plant* states.

- **Recoverable**. Satisfies the *Safety Requirements* to such a degree that the *Plant* can tolerate "aggressive" commands.
- **Safe**. Satisfies the *Safety Requirements*, yet cannot tolerate "aggressive" commands.
- **Unsafe**. Does not satisfy the *Safety Requirements*.

To say that the *Plant* is in a recoverable state makes a statement about two things: the *Plant* and the TW component. First, a recoverable state is one that satisfies the *Safety Requirements*. Second, to say that a *Plant* state is recoverable requires that if UN output is used in the current time-step, the TW component output must be able to satisfy the *Safety Requirements* in the next time-step. To state it simply, "In a recoverable state, if you choose the unreliable command now, the trustworthy command must ensure safety later."

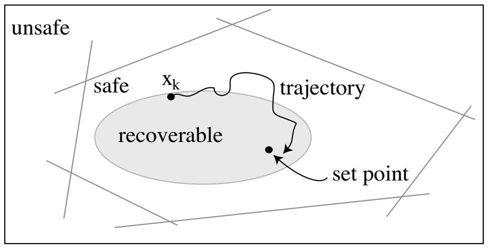

Fig. 2. The Simplex state space.

We define these states for the control domain. Referring to Fig. 2 we denote the state of the plant at time k as $\mathbf{x}_k$. As time passes, the state of the plant evolves, forming a trajectory, T. The trajectory is dependent on the *Plant* dynamics, control policy and initial condition. Hence, we represent a point in the system trajectory at time j, under control policy U, starting with initial condition $\mathbf{x}_k$ as $T(\mathbf{x}_k, U, j)$. A state is considered *safe* if it lies within the safety set, which is denoted by $\mathcal{S}$. Unsafe states are denoted as $\bar{\mathcal{S}}$. We define a state as being "recoverable" if the future trajectory lies within the safety set:

$$T(\mathbf{x}_k, U, j) \in \mathcal{S} \quad \forall \quad j \geq k \tag{1}$$

Thus, the set of all states $\mathbf{x}_k$ that satisfy (1) represents the recoverable set $\mathcal{R}$:

$$\mathcal{R} := \{\mathbf{x}_k : T(\mathbf{x}_k, U, j) \in \mathcal{S} \quad \forall \quad j \geq k\}$$

Note that $\mathcal{R}$ is dependent on the particular control policy U employed. If we use the trustworthy controller U_{TW} to define $\mathcal{R}$, then we can stipulate the following two requirements for the unreliable controller U_{UN} to be accepted at time k:

$$T(\mathbf{x}_k, U_{\text{UN}}, k+1) \in \mathcal{R}$$

This implies that:

$$\mathbf{x}_{k+1} = T(\mathbf{x}_k, U_{\text{UN}}, k+1) \in \mathcal{S}$$

and

$$T(\mathbf{x}_{k+1}, U_{\text{TW}}, j) \in \mathcal{S} \quad \forall \quad j \geq k+1$$

The notion of recoverable can also be expressed as "stability in the sense of Lyapunov" [7].

A. Logical Framework

We provide a companion logical framework to our reference model to disambiguate the English description provided thus far. We also present this generic framework because we foresee that Simplex can be applied to multiple domains, not just control.

Our reference model requires its logical framework to express reactivity. Given any possible input from the *External Context*, subject to the *Domain Model*, there must exist at least one TW output that will always keep the *Plant* in a safe state. Expressing this alternation of "any possible input" and "one control command" is a challenge.

On the one hand, we want to express the current *Plant* state in terms of the most recently received input and the most recently transmitted output. Yet at the same time, we would require recursion such that the next input partially depends on the current output.

Streams. We assume that we are given the streams of input and output a priori. Practically, it seems unrealistic to know all inputs at once, but we argue that, for our purposes, is mathematically equivalent to the reactive nature we wish to capture.

- Each stream is a sequence of inputs or outputs for each point in time. The input stream, ins, indicates the input, i, for each time k. We denote the input at time k as $i_k = ins(k)$.
- We denote the k^{th} prefix of an infinite stream, s, as $s^k = (s(0), \ldots, s(k))$. Consider a finite prefix of the input stream, $ins^k = (ins(0), \ldots, ins(k))$. The function $out(ins^k)$ produces the k^{th} output in response to the first k inputs, meaning that we do not allow the output stream to look into the future. To denote the output at time k, we use $o_k = out(ins^k)$.
- The current state of the plant, constructed from all previous inputs and outputs, is defined by the current input/output pair, (i_k, o_k).

Safety. The ultimate goal of Simplex is for the *Machine* output to satisfy the *Safety Requirements*. To express this notion of safety, we must differentiate between two safety predicates: "safe at the moment" and "always safe."

- We denote "safe at the moment" as $\texttt{sf}(i_k, o_k)$. For $\texttt{sf}(i_k, o_k)$ to be satisfied means that the *Plant* satisfies the *Safety Requirements* in the current state (i_k, o_k).
- To define "always safe" or `Safe`, we use a predicate $\texttt{pn}(ins^k, out(ins^k))$ that yields the set of "possible next" inputs at time $k+1$ given the input and output prefixes up to time k. Essentially, this `pn` predicate is equivalent to our reference model's *Domain Model*, which dynamically predicts the future state of the plant based on past and current sensor input and command output.
- We define `Safe` with respect to: i) the `sf` predicate; ii) the current *Domain Model* of the *Plant*, `pn`; and iii) the output stream function. Thus, `Safe` is evaluated

dynamically with respect to the current state and the current output.

$$\begin{aligned}
&\texttt{Safe}\langle \texttt{sf}, \texttt{pn}, \mathit{out}\rangle = \\
&\forall\, \mathit{ins}, \text{ let } \mathit{outs}(m) = \mathit{out}(\mathit{ins}^m) \text{ in} \\
&\texttt{sf}(\mathit{ins}(0), \mathit{outs}(0)) \wedge \\
&(\exists\ i \,\epsilon\, \texttt{pn}(\mathit{ins}^0, \mathit{outs}^0)) \wedge \\
&(\forall n,\, 0 < n \leq k, \mathit{ins}(n) \,\epsilon\, \texttt{pn}(\mathit{ins}^{(n-1)},\, \mathit{outs}^{(n-1)}) \\
&\Rightarrow \texttt{sf}(\mathit{ins}(k),\, \mathit{outs}(k)) \wedge (\exists\ i \,\epsilon\, \texttt{pn}(\mathit{ins}^k, \mathit{outs}^k))
\end{aligned}$$

Recoverable. In order to guarantee that the *Plant* is safe with respect to the TW, we require that the *Plant* is in a recoverable state when choosing any output.

- The output of the UN component is $\text{UN}(i)$. Our only assumption of UN is that if it produces output, the output is of the correct type.
- TW uses both the original input as well as the unreliable output to calculate its own output, $\text{TW}([i_0, \ldots, i_k], [\text{UN}(i_0), \ldots, \text{UN}(i_k)])$.
- We define the recoverable predicate, $\texttt{r}(i, o)$. This predicate, like pn and Safe, can be evaluated dynamically to determine if the *Plant* is in a recoverable state. A *Plant* is recoverable if: i) at the current time-step the current input/output pair satisfies the *Safety Requirements*, $\texttt{sf}(i, o)$; and ii) for all the possible next states, a sequence of outputs exist which keeps the *Plant* Safe.

$$\begin{aligned}
&\texttt{r}\langle \texttt{sf}, \texttt{pn}, \text{TW}\rangle([i_0, \ldots, i_k], [o_0, \ldots, o_k]) = \\
&\texttt{sf}(i_k, o_k) \wedge \text{let } \mathit{out}([x_0, \ldots, x_m]) = \\
&\text{TW}([i_o, \ldots, i_k, x_0, \ldots, x_m], \\
&\qquad [o_0, \ldots, o_k, \mathit{out}([x_0]), \ldots, \mathit{out}([x_0, \ldots, x_{m-1}])]) \\
&\text{in } \texttt{Safe}\langle \texttt{sf}, \texttt{pn}, \texttt{out}\rangle
\end{aligned}$$

- **Criteria:** If the CHKR does not choose the TW output, then the UN output must be recoverable.

$$\begin{aligned}
&o_k \neq \text{TW}(\mathit{ins}^k, \text{UN}(\mathit{ins}^k)) \Rightarrow \\
&\texttt{r}\langle \texttt{sf}, \text{PN}, \text{TW}\rangle([i_0, \ldots, i_k], [o_0, \ldots, o_k])
\end{aligned}$$

Discussion. To evaluate the recoverable predicate requires knowledge of the *Plant's* state. Given that the system is interacting with the physical world, the system must have three sets of information to make this evaluation. Foremost, it needs *Sensor Data* interpreted by the *Domain Model*. Next, it requires knowledge of the entire *External Context*: the mass of the device, the speed of the vehicle, the accuracy of the sensor. Finally, it requires an estimate of the *Plant's* current state.

Though we have shown a fairly strenuous framework, we recognize that in some instances calculating the set of recoverable states is a more intuitive decision procedure. For example, in the original Simplex prototype, the Lyapunov function [10], [11] was used to calculate the set of recoverable states for the inverted pendulum. In this paper, we do not claim to estimate a recoverable state, rather we define what the recoverable state must imply. It is up to domain experts to determine how to calculate the recoverable state.

B. Example 1: Simple Object

With the Simplex Reference Model described, we proceed with a series of example architectures which are instantiations of our model. Consider an object starting at $x = 0$ meters and moving in one dimension towards a fixed obstacle, located at 50 meters. The object has one *Safety Requirement*; do not hit the obstacle. The object has one functional requirement; get as close to the obstacle as quickly as possible.

For this example, we use two controllers:

- **Unreliable**. Command the object to move towards the obstacle at its maximum velocity.
- **Trustworthy**. Command the object to stop completely.

For the *Domain Model*, we summarize what we know about the *External Context*. The object can move at a maximum velocity of 1 meter-per-second. For the purposes of this simple example, control commands are issued as a change in velocity, $\Delta\, v$. The object executes these control commands with exact precision. The object has instant acceleration, but when traveling at its maximum velocity, its maximum breaking force yields a minimum deceleration rate of $\frac{1}{7}$ meter-per-second^2. The *Domain Model* receives *Sensor Data* every second, which it interprets and forwards to the *Machine*. This sensor data provides perfect observations about the object's location, expressed as a distance from its origin.

At a high level, how do we expect Simplex to work in this example? First, we combine the *Safety Requirements* and the *Domain Model* to determine $\mathcal{R} = \{\mathbf{x}_k : \mathbf{x}_k < 46.5\}$, since it takes 3.5 meters to stop when moving at full speed under the trustworthy controller. The object starts at 0 meters, at a safe distance from the obstacle, in other words, $\mathbf{x}_0 \in \mathcal{R}$. Hence, the Simplex CHKR chooses the UN output to move full speed towards the obstacle. As the object approaches the obstacle, and nears an unsafe distance, it enters a state that is not recoverable and Simplex must choose the TW output. So, upon receiving each input, the CHKR must determine the following in order to choose the UN or TW output:

- What is the current position, $\mathbf{x}_k$?
- Suppose the UN command is issued, what is the estimate of x at the time-step, $k+1$? In other words, $T(\mathbf{x}_k, \text{UN}, k+1)$
- Is the estimate of x at $k+1$ a recoverable state for TW? That is, could a TW command be issued at $k+1$

that would ensure the object does not hit the obstacle? In other words, is $T(\mathbf{x}_{k+1}, U_{\mathrm{TW}}, k+1) \in \mathcal{R}$?

That is, the CHKR must examine the current state and "possible next" states of the *Plant* to determine if the output from UN is recoverable. If the output from UN satisfies the recoverable predicate, then the CHKR chooses the UN output. Otherwise, the CHKR chooses the output from TW component.

Fig. 3 shows the transition from the UN to TW output. Indicated by (1), the object reaches position $x = 45$ meters at $k = 45$ seconds. Given the UN output, the CHKR uses $\mathtt{pn}$ to estimate the next state, x'. At time $k+1$, the single possible next state is $x = 46$ meters. The CHKR chooses the UN output since x' is recoverable. Indicated by (2), the object reaches position $x = 46$ meters at $k = 46$ seconds. This time, the CHKR chooses the TW output since the UN output violates the recoverable predicate. The TW output is selected for subsequent outputs. Finally, indicated by (3), the object is stopped safely at $x = 49.5$ meters.

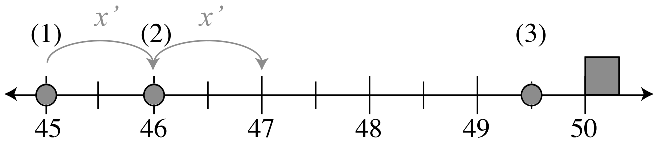

Fig. 3. Transitioning from unreliable to trustworthy output.

Note that the behavior is not necessarily optimal. Optimal behavior could be achieved by an "intelligent" checker; we highlight that this is not the functionality of the CHKR. It must simply approve or decline control commands issued to it.

C. Example 2: Unreliable Object

So far, our discussion of the Simplex reference model makes very strong assumptions. Our first example exists in an impractically perfect world; our object responds instantly to control commands and the *Machine* receives perfect sensor data. Yet, there are many applications where input is sporadic or incorrect and control commands are not followed with precision. To utilize unreliable data with a minimal impact on safety, we rely on the *Domain Model*.

As mentioned previously, the *Domain Model* represents everything that is known about the *External Context* that can be used to estimate the behavior, or dynamics, of the *Plant*. For the simple, deterministic example discussed our first example, the *Domain Model* is straight-forward. For other systems, more care needs to be taken in designing the *Domain Model*.

As before, consider a object moving in one dimension towards a fixed obstacle. However, in this example, the object must get as close to the obstacle as possible, in finite time, without hitting it. Moreover, the sensor cannot take perfect observations, and the object does not follow control commands exactly. We have selected two UN controllers and one TW controller for this example. These three controllers yield the architecture pictured in Fig. 4.

- **Unreliable, Level 1**. Can only instruct the object to move towards the obstacle at maximum velocity, either forwards or backwards. It is unaware of the requirement not to overshoot 50 meters; it would give the instruction to move forward at full speed even when located at 49 meters. This is a "high gain" controller which does not attempt to preserve safety, but is fast. We denote this controller as UN_1.
- **Unreliable, Level 2**. The same as controller UN_1 except that it instructs the object to move at one-fifth its maximum velocity. We denote this controller as UN_2.
- **Trustworthy**. This controller has been fully verified. It can generate control actions anywhere between a complete stop and one-tenth of the maximum velocity. We denote this controller as TW.

For the *Domain Model*, we summarize what we know about the *External Context*, including the *Plant*. As before, the object can move at a maximum velocity of 1m/sec. Control commands are issued as a change in velocity, Δv. However, there is process noise, and the object does not follow these control commands perfectly.

As before, the *Domain Model* receives sensor data every second. Initially, we know the exact location of the object, but in this example, our sensor is not perfect; due to observation noise we cannot observe the object's location accurately.

We use a linear stochastic equation to model the position and velocity of the object using the following terms:

- $\mathbf{x_k} = \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix}$ where x_k is the position and $\dot{x}_k$ is the velocity of the object at time k.
- A: Matrix modeling the transition of the state in one time-step, ΔT.
- B: Matrix modeling the effect of the control input, u_k.
- u_k: Control input which in this context is equivalent to Δv.
- H: Matrix modeling the state observation.
- w_k, v_k: Zero-mean Gaussian process and observation noise, respectively.
- ΔT: Time-step, equal to 1sec in this example.
- $\Sigma_k^{\mathbf{x}}$: The state estimation error covariance.

That is,

$$\begin{aligned} \mathbf{x}_k &= A\mathbf{x}_{k-1} + Bu_k + w_k \\ \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} &= \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ \dot{x}_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta T \\ 1 \end{bmatrix} u_k + w_k \end{aligned}$$

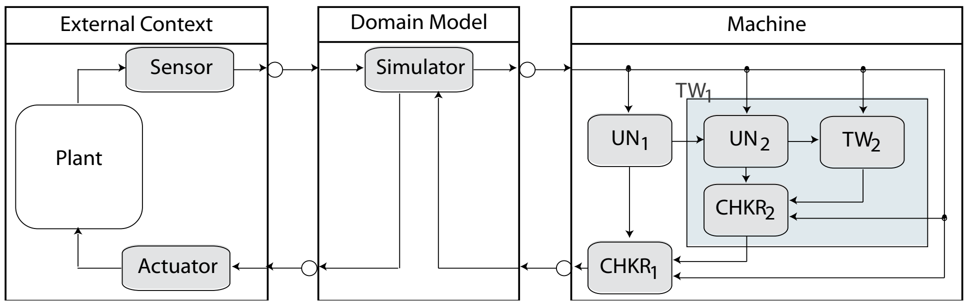

Fig. 4. Architecture for the "unreliable object" example.

Observations are denoted by z_k and consist of only the location of the object:

$$\begin{aligned} z_k &= H\mathbf{x}_k + v_k \\ z_k &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} + v_k \end{aligned}$$

Based on this linear Gaussian model, we use a Kalman filter [12] for the *Simulator*, to calculate an estimate of the *Plant*. The Kalman filter computes the i) the expected value, $\bar{\mathbf{x}}_k$, and ii) the probability distribution Σ_k^x of the object's position and velocity. That is, the Kalman filter produces an estimate and associated distribution for the location and velocity of the object based on the *Domain Model*, *Sensor Data* and the *Control Commands*.

At a high level, how do we expect Simplex to work in this example? The object starts at location 0 with no estimation error, and is safely distanced from the obstacle; with high probability, the object will remain recoverable regardless of the control command implemented. Referring again to the architecture in Fig. 4, CHKR$_2$ will approve the commands from UN$_2$. This control action is then forwarded as "trustworthy" input to CHKR$_1$. Since the *Plant* state is recoverable, CHKR$_1$ will then approve commands from UN$_1$.

As the object moves, the *Domain Model* uncertainty of the position estimate increases. When the probability of moving into a recoverable state using controller UN$_1$ becomes unacceptably low, such that the recoverable predicate is not satisfied, the CHKR$_1$ will switch to approve the control actions from the output of CHKR$_2$, which would correspond to controller UN$_2$. Similarly, at some later object, CHKR$_2$ would start to decline control actions from UN$_2$ and approve those from TW. Note that to ensure safety, the conditions for a recoverable state in CHKR$_1$ must at most as restrictive as those in CHKR$_2$. This will ensure that control commands approved by CHKR$_2$ are subsequently declined by CHKR$_1$.

We note that same architecture used in the first example is reproduced here as the "nested" structure depicted in the *Machine* in Fig. 4. Such nesting demands careful attention to the algorithms employed by the CHKRS. The *Safety Requirements* across the two CHKRS must be consistent, and the recoverable predicate enforced by CHKR$_1$ should be strictly stronger than those in CHKR$_2$.

III. Case Study: Convergence Lab Testbed

The Convergence Laboratory testbed at the University of Illinois at Urbana-Champaign [13] [6] uses a networked, component-based middleware for control called Etherware [14]. The application-layer components shown in Fig. 5 execute on top of Etherware to control of a fleet of remote-controlled cars. The application is based on a control loop consisting of a **Vision Sensor**, **Controller**, and **Actuator**. The **Trajectory Planner** generates waypoints for each car, to be followed by each car's **Controller**. The cars have two directives. First, they must follow the set of waypoints determined by the system **Trajectory Planner** to a reach a set of goal-bins in the network of roads. Second, they must do so without colliding with any other cars.

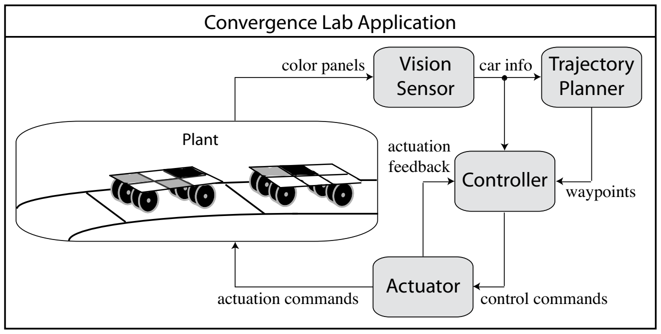

Fig. 5. The testbed application components control a fleet of remote-controlled cars.

We selected the Convergence Laboratory testbed because it demonstrates four of the challenges in the CPS domain. First, the testbed uses a component-based design.

Second, the testbed is designed for control. A classical, discrete time control system relates to the physical world and requires reliable communication, minimal jitter, and hard deadlines [15]. Third, it components execute concurrently across multiple nodes. the testbed controls the plant using multiple heterogeneous nodes concurrently executing tasks such as sensing and actuation. Finally, the testbed uses both wired and wireless communication across its multiple heterogeneous nodes. What follows is a description of how we have redesigned two subsystems in the testbed using the Simplex reference model. Interested readers can consult [16] for a description of the original architecture.

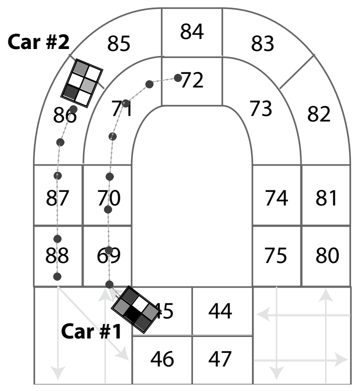

Fig. 6. Car #1 and #2 travel to their goal-bins 88 and 72 on the road network.

Fig. 6 shows a simple scenario in the testbed. The cars are uniquely identified by a set of six color panels placed on the roof of the car. Two cameras mounted to the ceiling are each responsible for monitoring one-half of the road network. Each camera tracks the car identities, positions and orientations based on the color panels. We use a third-party imaging library to extract a collection of digital color **blobs** from the raw camera feed, where a **blob** is described by its hue, saturation, luminance, and center. We have no control over the imaging library, so we consider it as part of the camera and in the *External Context*. We treat the **blobs** as our *Sensor Data*.

The cameras are unreliable components and very often produce unreliable vision data. In the worst lighting conditions, the cameras report 80% of the total color panels and only 63% of those are the correct color. That is, in the worst case, they only detect about half of the color panels correctly.

We apply our Simplex reference model to the testbed so that it may use the unreliable camera component while limiting fault-propagation. We begin by splitting the vision data into two logical streams, to make a clear divide between performance and safety.

- **CarInfo**. A list of all car information. Each entry in the list consists of a single car's identification number, x- and y-position, orientation, and confidence. The confidence value indicates the probability that the car's information is correct. This confidence is derived in two-passes. If the car is within a validation region of its last location, then confidence is high. Otherwise, the confidence is derived from the Kalman filter covariance.
- **BinInfo**. A list of all bin information. Each entry in the list consists of a bin number, and whether or not that bin is safe for driving. Each entry is watchdog timer-based, meaning that if the bin has not been verified as empty within a particular time-window, it will be flagged as unsafe.

By splitting the data in this way, we separate the unreliable information from the reliable. In the worst case–if the camera breaks, the lights go dark, and all the wireless cards fail simultaneously–if a bin's state has not been reported within a particular time window, then no car will travel in it. The time window is derived from the worst case: it is the length of time it takes one car to travel from one bin to the next at maximum speed. Thus, the watchdog timer on the **BinInfo** stream maintain a timely list of assured bins. Whether or not the system is certain of a car's identity and location, it is certain of the safe bins in the road network.

Similarly, we make a logical division in system control. Each car is assigned three controllers:

- **High performance**. When a car is in a configuration of high confidence–when its identity and location are known with high probability, and the surrounding bins are safe for driving–we intend to use the high performance controller. This model-predictive controller uses high speeds and direct routes to reach the goal-bins.
- **Low performance**. When a car is in a configuration of low confidence–when its identity and location are not known with high probability, yet the surrounding bins are safe for driving–we intend to use the low performance controller. This controller uses low speeds and safe routes to reach the goal-bins.
- **Collision Avoidance**. This is a verified, trustworthy controller which avoids collisions. It always commands the car to perform a full stop.

Using these logical divisions, Fig. 7 summarizes how we apply Simplex to the testbed. Starting from the *External Context*, the camera sends **blob** data to the *Domain Model*. The blob information is processed by the *Simulator* to get a probabilistic model of the current state of the *Plant*. The **CarInfo** and **BinInfo** streams are sent to every *Machine* controlling a car. The **Collision Avoidance Checker** ensures a local car does not cause collisions, while the **Collision Avoidance Supervisor** ensures against collisions

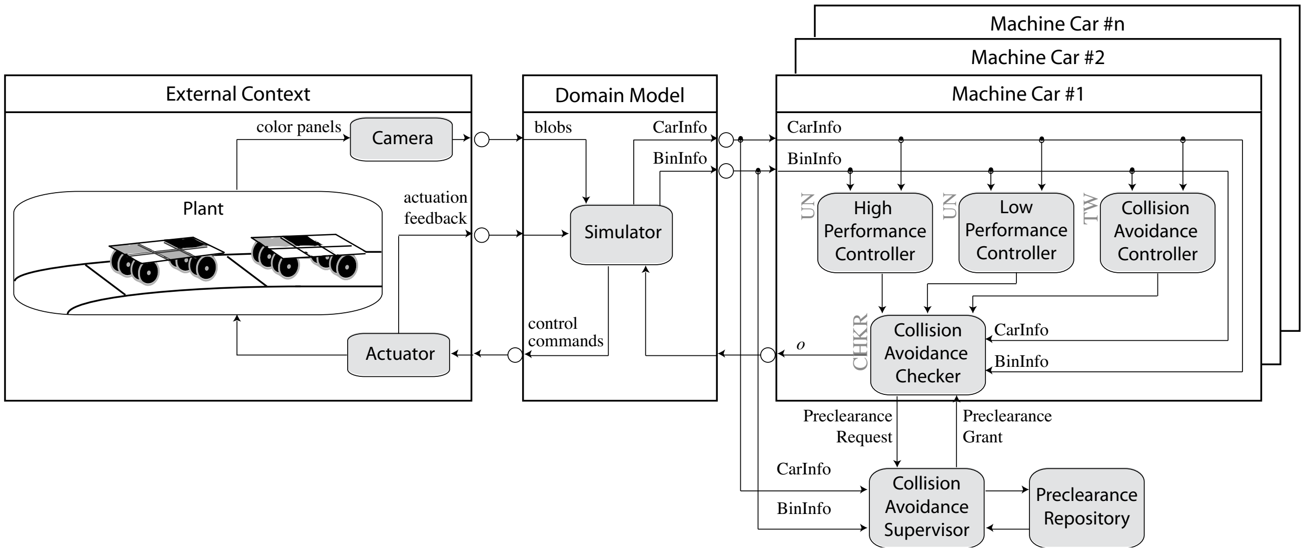

Fig. 7. Simplex architecture for testbed.

among all the cars. The **Actuator** sends feedback to the *Simulator* to indicate which control commands were issued.

A. Simplex for Reliable Vision

The cameras in the Convergence Lab testbed are unreliable at successfully detecting all of the color panels on all of the cars. However, they do correctly report **blob** data at least 63% of the time. That said, it is not necessary to simply throw out the simple least squares algorithm used to associate blobs to car identities. Instead we can use the Simplex architecture reference model to make the **Vision** subsystem more reliable.

Recall the logical divide we created in our vision data stream: **CarInfo** and **BinInfo**. The *Domain Model* must make the logical division in the camera data while the *Simulator* keeps an up-to-date estimate of the *Plant*. Shown in Fig. 8, the *Simulator* is another instance of a Simplex *Machine*.

This instance of the *Machine* is divided into two controllers. The UN component is a simple **Least Squares** association of the blobs to determine a list of car identities. The TW is a **Kalman Filter** of the blob input to make a similar association. The UN component makes a faster, though unreliable, association of blobs to **CarInfo** and **BinInfo**. The CHKR prefers the UN output.

To determine which output to choose, the CHKR maintains a list of **CarInfo** from the previous time-step. Based on the *Rules*, the CHKR can determine if every car reported in the current time-step is within a reasonable distance from the previous time-step. If so, the CHKR chooses the **Least Squares** output, reporting the car location with high confidence. If not, the CHKR chooses the **Kalman Filter** output, whose confidence is a function of the Kalman filter's covariance value.

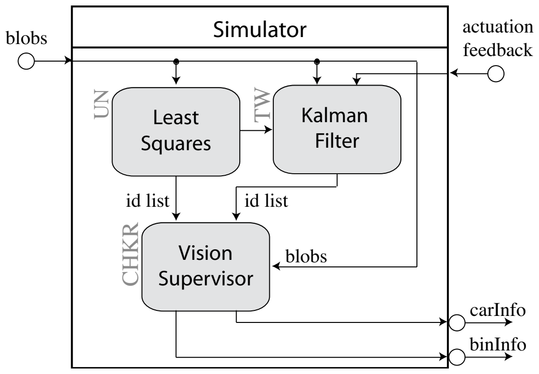

Fig. 8. Detail of the *Simulator* in Fig. 7.

As we mentioned, this Simplex subsystem uses the **CarInfo** confidence to cooperate with the *Machine*. While the Kalman Filter on the **blob** information alone increases reliability of the **Vision** subsystem, reporting confidence information to the *Machine* means a more coherent choice on which controller to use for the given **CarInfo** and **BinInfo**.

B. Simplex for Safe Control

The control of each car is divided into three controllers: **High performance**, **Low performance**, and **Collision Avoidance**. These three controllers must be combined to

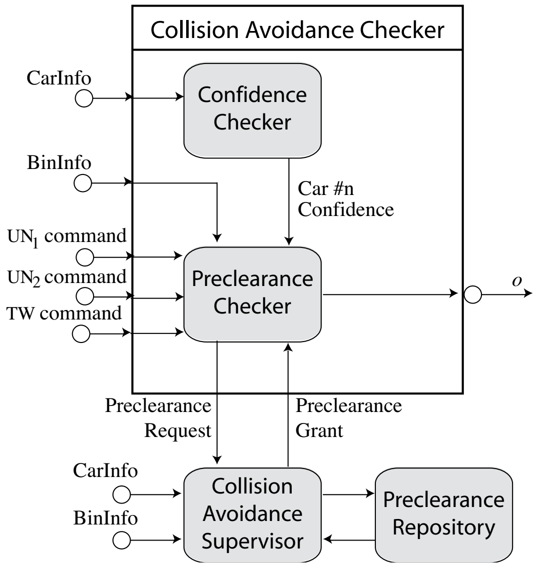

Fig. 9. The Collision Avoidance Checker.

achieve the *Safety Requirement* of collision avoidance. Fig. 9 summarizes how the **Collision Avoidance Supervisor** acts as the CHKR to achieve this. However, the **Collision Avoidance Checker** is only a local CHKR for a single car. A global CHKR, the **Collision Avoidance Supervisor**, is necessary to prevent collisions among all the cars.

To achieve global collision avoidance, the **Collision Avoidance Supervisor** uses a notion of "preclearance" to supervise all of the cars in the testbed [16]. No car can move in a given region until it has been granted to move in that region. The **Collision Avoidance Supervisor** keeps track of all granted preclearance regions using the **Preclearance Repository**. This repository forms the basis for the global *Safety Requirements* that must be maintained for all cars in the *Plant*. Meanwhile, each car has a local **Preclearance Checker** maintains a list of preclearance regions that it has been granted. Thus, the **Collision Avoidance Checker** makes a three-pass check:

1) The **Confidence Checker** extracts the local car's confidence from **CarInfo**.
2) The **Preclearance Checker** confirms that the car's confidence places it reliably in the already granted preclearance. If there is sufficiently high probability that the car is outside of the granted preclearance region, the TW is used. The **Preclearance Checker** then makes a request to the **Collision Avoidance Supervisor** for additional preclearance.
3) If the car's confidence is located safely in the already granted preclearance, the **Preclearance Checker** checks if the UN_1 control command will keep it within preclearance. If not, the same check is made on UN_2. If not, the TW control is used.

Notice how the cooperation between the *Domain Model* and the *Machine* incurs greater system safety. The confidence generated by the **Domain Model** is used by the CHKR to determine which control command to use.

IV. Related Work

The original Simplex architecture [1], [2] was designed to provide fault-tolerant, dynamic upgrades for real-time systems. A number of implemented prototypes [1], [3], [4], [5] use a decision procedure to choose control commands from either a complex controller or a simple controller.

Our Simplex reference model is about describing a working solution that can be used in multiple applications. Other work seeks to achieve this goal. Recent design patterns, such as [17] and [18], address issues faced by real-time domains. The "Safety Executive" [17] is one pattern which describes how a component should enter its fail-safe state. Yet, unlike Simplex, the Safety Executive is centralized; it is responsible for both identifying the system fault behavior as well as coordinating the system into its fail-safe state. Simplex separates these tasks, so that no one component is more complicated than necessary, allowing the system to be less fault-prone as a result. The 3CoFramework [19] separates a distributed, component-based system into three types of entities: component, connector, and coordinator. This separation creates a flexible implementation framework, yet the hefty responsibility of fault-tolerance is left entirely up to the connector.

Simplex ensures plant safety based on an estimate of the current and future state. Such estimations have been addressed in several domains. For example, see [20] for the general estimation problem, and [21] for approaches in the collision avoidance problem. Switching controllers falls within the realm of hybrid systems [11] and the Simplex architecture represents a state-dependent switching controller. Within that realm, Simplex ensures safety of a system by 'falling back' on a reliable controller, as compared to switching between controllers in an attempt to stabilize the system, avoid collisions or provide some performance guarantee.

Another source of faults in CPS systems is the physical world, which can often produce unreliable data through faulty sensors or misinterpreted sensor data. Kalman filtering [12] [20] is a well-known state estimation approach to handle faulty, inaccurate, or sporadic sensor data. We utilize Kalman filtering as a tool for state-estimation in the Simplex architecture. Extended Kalman Filtering is presented in [22] for a very specific CPS application, location tracking and mobility connectivity for cellular phone networks. However, this extensions does not present Kalman filtering in the context of software architectures which improve safety for CPS. All told, we know no work to date that combats the challenges of limiting fault-propagation in the presence of of unreliable components through the careful assembly of system architecture in a manner similar to Simplex.

V. Conclusions

With the emergence of CPS, we need reference models to reason about their software architectures. This paper described our Simplex reference model to assist developers with CPS architectures which limit fault-propagation.

Our contribution was presented in two parts. First we presented our Simplex reference model based on the lessons learned from existing prototypes. Our reference model decomposes a CPS architecture into a *Plant*, *External Context*, *Machine*, *Domain Model*, and *Safety Requirements*. In particular, we presented a logical framework to formally define a *recoverable* state, and how the *Machine* uses this notion to always uphold the *Safety Requirements*.

Second, we presented a case study of the Convergence Laboratory testbed at the University of Illinois at Urbana-Champaign [6]. In this case study, we redesigned two subsystems. In the first, we used Simplex to improve the reliability of unreliable data. In the second, we used Simplex to provide safe control of the cars. Combined, these two subsystems cooperate in such a way that they provide greater safety together than each could individually.

We are in the initial stages of implementing these subsystems in the Convergence Lab. While safety was a paramount issue in this paper, real-time quality of service issues cannot be ignored during the implementation stage. Namely, as useful as the Simplex architecture is at limiting fault propagation, it cannot affect deadline completion in a real-time system. We look forward to publishing future performance results based on this and similar implementation issues. We forsee that the implementation will yield additional lessons that we may reincorporate back into our reference model to strengthen its relevance.

VI. Acknowledgments

The authors of this paper are thankful for our writer's workshop reviewers: Jacob Biehl, Jodie Boyer, Erin Wolf Chambers, Joe Hendrix, Shamsi Iqbal, Michael Katelman, and Mu Sun. The authors are also appreciative of Dr. Ralph Johnson's feedback. This material is based upon work partially supported by: AFOSR under Contract No. F49620-02-1-0217; ONR under Contract No. N000140510739; NSF under Contract Nos. NSF ANI 02-21357 and CCR-0325716; USARO under Contract No. DADD19-01010-465

References

[1] Neal Altman, Chuck Weinstock, Lui Sha, and Danbing Seto, "Simplex in a hostile communications environment: The coordinated prototype", *Software Engineering Institute Technical Report CMU/SEI-99-TR-016*, 1999.

[2] L. Sha, "Using simplicity to control complexity", *IEEE Software*, vol. 18, no. 4, July/August 2001.

[3] Kihwal Lee and Lui Sha, "A dependable online testing and upgrade architecture for real-time embedded systems", in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[4] Software Engineering Institute, "http://www.sei.cmu.edu/simplex/demonstrations/distributed.html", 1999.

[5] Danbing Seto, Enrique Ferriera, and Theodore Marz, "Case study: Development of a baseline controller for automatic landing of an F-16 aircraft using linear matrix inequalities (LMIs)", *Software Engineering Institute Technical Report CMU/SEI-99-TR-020*, 2000.

[6] Scott Graham and P. R. Kumar, "The convergence of control, communication, and computation", *Proceedings of PWC 2003: Personal Wireless Communications. Lecture Notes in Computer Science*, vol. 2775, 2003.

[7] W. L. Brogan, *Modern Control Theory*, Prentice Hall, 3 edition, 1991.

[8] H. S. Witsenhausen, "Separation of estimation and control for discrete time systems", *Proceedings of the IEEE*, vol. 59, no. 11, pp. 1557–1566, November 1971.

[9] H. S. Witsenhausen, "A counter example in stochastic optimal control", *Siam J. Control*, vol. 6, pp. 131–147, 1968.

[10] A. M. Lyapunov, *Stability of motion*, Academic Press, New-York and London, 1966.

[11] D. Liberzon, *Switching in Systems and Control*, Systems and Control: Foundations and Applications. Birkhäuser, 2003.

[12] Rudolph Emil Kalman, "A new approach to linear filtering and prediction problems", *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[13] "IT Convergence Lab", in *http://decision.csl.uiuc.edu/ testbed*.

[14] Girish Baliga, Scott Graham, Lui Sha, and P. R. Kumar, "Etherware: Domainware for wireless control networks", in *IEEE Seventh International Symposium on Object-Oriented Real-Time Distributed Computing*, 2004, pp. 155–162.

[15] C.L. Philips and H.T. Nagle, *Digital Control System Analysis and Design*, Prentice Hall, 1995.

[16] C. L. Robinson, H.-J. Schutz, G. Baliga, and P. R. Kumar, "Architecture and algorithm for a laboratory vehicle collision avoidance system", in *IEEE Multi Conf. on Systems and Control*, 2007.

[17] B. Douglass, *Real-Time Design Patterns*, Addison-Wesley Publishing Company, Boston, Massachusetts, 2003.

[18] Pau Arumí, David García, and Xavier Amatriain, "Data-flow pattern catalog for sound and music computing", in *Proceedings of the 13th Conference on Pattern Language of Programs*, Portland, OR, 2006.

[19] Shifeng Zhang and Steve Goddard, "3CoFramework: A component-based framework for distributed applications", in *Proceedings of the International Conferences on Software Engineering Research and Practice*, Las Vegas, NV, 2003.

[20] I. B. Rhodes, "A tutorial introduction to estimation and filtering", *IEEE Transactions on Automatic Control*, vol. AC-16, no. 6, December 1971.

[21] J. K. Kuchar and L. C. Yang, "A review of conflict detection and resolution modeling methods", *IEEE Transactions on Intelligent Transporation Systems*.

[22] Pubudu N. Pathirana, Andrey V. Savkin, and Sanjay Jha, "Robust extended Kalman filter applied to location tracking and trajectory prediction for PCS networks", in *Proceedings of the 2004 IEEE International Conference on Control Applications*, 2004.