The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety

Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
United States of America
{*sbak2, dchivuk2, oadekunl, musun, mcaccamo, lrs*}*@illinois.edu*

Abstract

Embedded systems in safety-critical environments demand safety guarantees while providing many useful services that are too complex to formally verify or fully test. Existing application-level fault-tolerance methods, even if formally verified, leave the system vulnerable to errors in the real-time operating system (RTOS), middleware, and microprocessor. We introduce the System-Level Simplex Architecture, which uses hardware/software co-design to provide fail-operational guarantees for both logical application-level faults, as well as faults in previously dependent layers including the RTOS and microprocessor. We also provide an end-to-end design process for the System-Level Simplex Architecture where the AADL architecture description is automatically constructed and checked and the VHDL hardware code is generated.

To show the efficacy of System-Level Simplex design, we apply the approach to both a classic inverted pendulum and a cardiac pacemaker. We perform fault-injection tests on the inverted pendulum design which demonstrate robustness in spite of software controller and operating system faults. For the pacemaker, we contrast the provided safety guarantees with those of a previous-generation pacemaker.

1. Introduction

Embedded systems are growing in complexity and must continue to meet requirements including reliability and performance. Reliability is difficult to scale using traditional designs because large systems have high complexity, and high complexity presents more possibilities for errors. High performance, on the other hand, drives system complexity upward.

One way to deal with complex systems in a safety-critical environment is through the Simplex Architecture [1]–[4]. This architecture provides an application-level safety guarantee by "using simplicity to control complexity". It uses a simple safety controller subsystem to ensure the stability of the plant. This conservative safety control core is then complemented by a high-performance complex control subsystem. A decision module then uses the high-performance complex controller whenever possible, but will switch to the safety controller when system liveliness is jeopardized. This design has been applied to improve the safety of a diving controller [5], a fleet of remote-controlled cars [4], and a set of advanced aircraft maneuvers [6].

One drawback of the Application-Level Simplex Architecture, however, is that bugs present in the microprocessor, Real-time Operating System (RTOS) or middleware, or resultant from their upgrades, are not guaranteed to be handled safely. Although great progress has been made towards producing a verifiable RTOS [7], most current RTOSes have been neither formally verified nor exhaustively tested. We would still like to use the services provided by an RTOS, without requiring its correctness to guarantee system safety.

We thus propose the System-Level Simplex Architecture, which provides robustness in the presence of *both* bugs in the application and bugs in previously dependent layers such as the RTOS. In this new architecture, we perform hardware/software partitioning on the Simplex framework. Two Simplex safety-critical components, the safety controller and decision module, are moved to a dedicated processing unit, not for the typical HW/SW co-design reasons of power and performance, but instead to provide isolation from software-related complexity. Additionally, this architecture meets the temporal constraints of the monitored safety properties by design.

Using Simplex preserves safety in the presence of logical faults, but only if Simplex is properly designed and implemented. To address proper design, we provide an AADL-based [8] System-Level Simplex architecture generator and checker. If the AADL architecture model contains an improper application of System-Level Simplex, our checker determines exactly where the violation occurs and notifies the user. We describe several necessary conditions for proper System-Level Simplex Architecture design, and for each provide an example of a safety violation that may occur if it is ignored. To address proper implementation, we extract the safety controller and decision module behaviors (which are the two simpler

1080-1812/09 $25.00 © 2009 IEEE
DOI 10.1109/RTAS.2009.20

modules in the Simplex system) from the AADL model provided in the AADL behavior annex [9]. These behavior descriptions can be expressed and formally verified in model checking tools such as UPPAAL [10]. The creation of these finite-state machines and their safety model checking are application-specific, and a responsibility of the designer. However, if such a finite-state machine description is provided, we can use our VHDL code generator to immediately create the associated hardware code for the safety core.

In brief, the key contributions of our work are:

- The design of the System-Level Simplex Architecture which can handle a superset of the failure modes of previous Simplex versions
- An end-to-end design process that both verifies a valid System-Level Simplex AADL architecture model and can generate hardware VHDL code from a finite-state machine description
- Empirical verification of both the practicality of end-to-end System-Level Simplex design, and the robustness guarantees through fault-injection testing in two case studies

We first briefly review Simplex and describe the System-Level Simplex Architecture in Section 2. This design provides resilience from failures caused by the operating system or middleware as well as logical errors in complex control software. In Section 3, we describe the end-to-end System-Level Simplex design process. This process consists of an AADL architecture generator and verifier, as well as a VHDL code generator. We then apply the architecture in two case studies in Section 4. In the first, we examine pacemaker design and contrast the failures handled by the System-Level Simplex Architecture pacemaker with a previous-generation pacemaker. Then we apply the System-Level Simplex Architecture to an inverted pendulum system and verify its robustness with fault-injection testing. We then discuss related work in Section 5 and conclude in Section 6.

2. System-Level Simplex Design

The System-Level Simplex Architecture is based on the original Simplex concept [1]. Simplex is logically divided into three subsystems: safety, complex, and decision (Figure 1). The safety subsystem has a simple, reliable controller which provides verifiably safe performance. This is used in case the complex controller malfunctions. The complex controller drives the system as long as it does not jeopardize system liveliness. This controller can be changed and upgraded while the system is running and may even contain bugs. The decision subsystem chooses which of the two previously-mentioned controllers to use. The decision

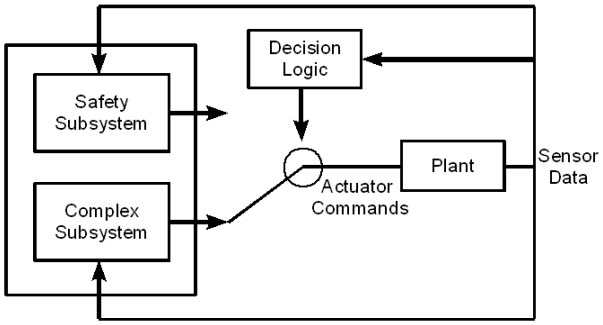

Figure 1. In the logical view of the Simplex architecture, the decision module chooses the controller version that drives the system.

module and safety controller make up the trusted computing base, and must function correctly for the system to remain safe, while most of the system's complexity is contained within the complex controller.

Previous Simplex designs had all three subsystems located at the application-level. This worked well for protecting the system from value faults from the complex controller, however it required that, to guarantee system safety, the middleware, the operating system, and the microprocessor be fully reliable. We relax this requirement in the System-Level Simplex architecture by performing hardware/software partitioning on the system. The two Simplex safety-critical components, the safety controller and the decision module, are moved into a dedicated processing unit outside the microprocessor.

This is akin to hardware/software co-design, except that we perform this move not primarily for reasons of performance and power consumption, but instead to protect from software-related faults. The designer therefore has a choice of what the dedicated processing unit should be. One option is to use a microcontroller to run the two safety core subsystems. However, in safety-critical systems, even processors are not completely trusted [11], and we would prefer to eliminate this underlying complexity. We instead choose to run the Simplex safety core on dedicated hardware. Ideally we would produce an Application-Specific Integrated Circuit (ASIC), but instead, for cost and reprogrammability reasons, we opted to perform our evaluation using a Field Programmable Gate Array (FPGA). The same VHDL code used to program an FPGA can be used to produce an ASIC.

By moving the Simplex safety core to isolated hardware, we can also provide temporal correctness for the monitored safety properties. If the high-performance complex subsystem does not produce a control command in the appropriate time, whether caused by an RTOS bug, poor cache performance or excessive bus contention, the conservative safety controller's output is used. Since the safety controller runs in parallel on isolated hardware (which prevents run-

time variations caused by resource sharing), the temporal constraints are met by design.

2.1. Fault Model

The System-Level Simplex Architecture tolerates two broad categories of faults: logical faults and resource sharing faults.

Logical faults occur when the complex controller passes an unsafe value to the decision module, or a value of an incorrect type. One cause for this sort of fault is a malfunctioning complex controller. An incorrectly-typed value, on the other hand, may cause logical operations that use it to fail. For example, if the control commands are IEEE floating-point values that correspond to voltages, the values NAN or infinity are incorrectly-typed. Another logical fault occurs when a non-functional complex controller does not output any value.

Resource sharing faults are caused by failures in common resources among components. The original Application-Level Simplex Architecture shares several resources, each of which can cause the system to fail. These include all the physical and logical resources managed by the OS like memory, CPU, and shared libraries. These faults can manifest directly (a misimplemented library causing the decision module process to crash), or indirectly (a mismanaged processor causing timing faults). Additional shared resources may include the communication bus and the power source.

There also exist out-of-scope faults which System-Level Simplex does not address. Specifically, the sensors and actuators used by our system must be reliable and accurate. The FPGA hardware, which runs our Simplex safety core, is assumed to be correctly manufactured. Additionally, the synthesis process, which takes our VHDL code and generates FPGA bitstreams, is unaddressed. However, these faults are rare since companies strive to provide reliable hardware and synthesis tools, and may be even further reduced by techniques like triple modular redundancy (TMR) [12]. We also do not handle environmental modeling faults which can be present in any system that uses formal methods. Since model checking is performed on the models and not the physical environment, a significant mismatch between the two results in an unsafe system, even if it is fully model checked. To account for these errors, the formal model should be reviewed, and fault-injection testing should still be performed on the final system.

3. System-Level Simplex Design Process

We now present the end-to-end design process for creating a System-Level Simplex design. The process begins with the designer providing a behavioral specification of the Simplex safety core (the safety controller and the decision module) and a logical architectural description. The behavioral specification is then formally verified to meet the application-specific safety requirements. One way to do this is by model checking a finite-state machine controller against safety properties also expressed as finite-state machines. After the behavior satisfies all safety requirements, we automatically transform the provided AADL architecture to a System-Level Simplex design. The designer can then modify the architecture to meet design-specific goals. To make sure the modifications do not result in an unsafe architecture, we provide a tool to assist the validation of the architecture, which checks a set of necessary safety requirements. Violations are reported back to the designer, who can then address each one. The process is summarized in Figure 2.

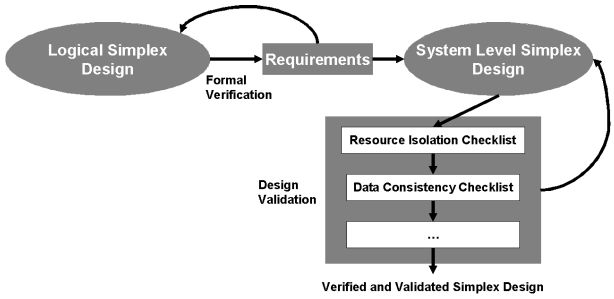

Figure 2. The end-to-end System-Level Simplex design process results in a verified behavior and a validated architecture.

For creating and verifying the System-Level Simplex architecture design, we use the AADL architecture description language [8]. This language is specifically designed to model the interaction of hardware and software for real-time and safety-critical embedded systems with the potential to support formal methods and the use of engineering models. Several organizations, including Boeing, the US Army, and SEI have evaluated using an architecture description language as the primary modeling notion for system-level analysis. One project by the US Army reported an estimated 50% man-hour savings in the reengineering effort required to upgrade an existing missile guidance system to run on a new hardware platform [13]. Similarly, our design process can be used to reengineer an existing Application-Level Simplex design into a System-Level Simplex design.

In the following sections, we describe each of the steps for creating a System-Level Simplex design in detail, and our contributions in automating the process. We used the OSATE [14] environment to create and validate our AADL models because of its support for both high-level logical design and low-level system properties. All of our tools are available for download at `https://agora.cs.uiuc.edu/display/realTimeSystems/System+Level+Simplex`.

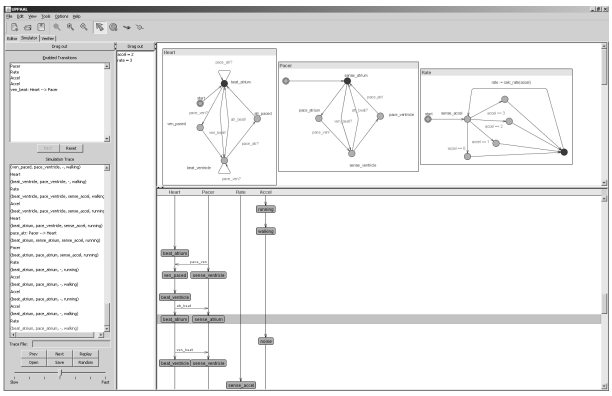

Figure 3. UPPAAL can be used to perform model checking on safety properties.

3.1. Initial Logical Design

The first step of the design process is to create the initial logical design using the classic Simplex paradigm. Our logical Simplex design for a cardiac pacemaker is shown in Figure 4. There are three threads *cc*, *sc*, *dm* for the complex controller, simple controller, and decision module, respectively. The sensor data are EKG signal events that detect when the heart's ventricle or atrium has paced. The actuation signals tell whether to send a shock to the ventricle or atrium through the pacemaker leads. There is also an extra sensor signal for the patient's acceleration that the complex controller uses to perform rate-adaptive pacing.

The specification of the logical design describes each thread's behavior using the AADL Behavior Annex [9]. The Behavior Annex describes behavior with message passing finite-state machines. This behavioral model can be translated to an equivalent specification in a model checking tool such as UPPAAL [10]. The UPPAAL model of the pacemaker behavior is shown in Figure 3. UPPAAL is then used to formally verify safety properties for the logical Simplex design.

3.2. System-Level Simplex Architecture Generation

We have created an OSATE plug-in that inputs the formally verified logical design from before, and automatically generates the general structure of the System-Level Simplex Architecture. The logical design in Figure 4 is transformed into the System-Level design shown in Figure 5.

The transformation wraps the individual threads into separate processes that have isolated memory and processors. Furthermore, all communication from the complex controller is put inside an application-level process. The pseudo code for the hardware design generation is shown in Code Block 1.

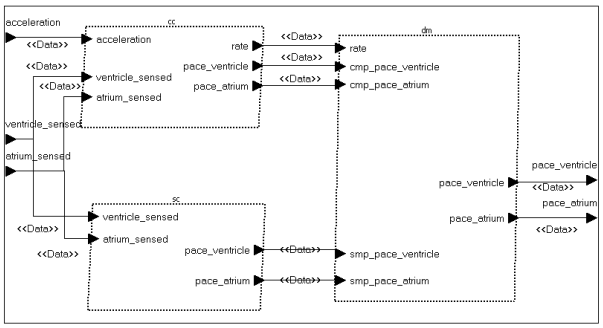

Figure 4. The design process takes in a logical Simplex AADL Model.

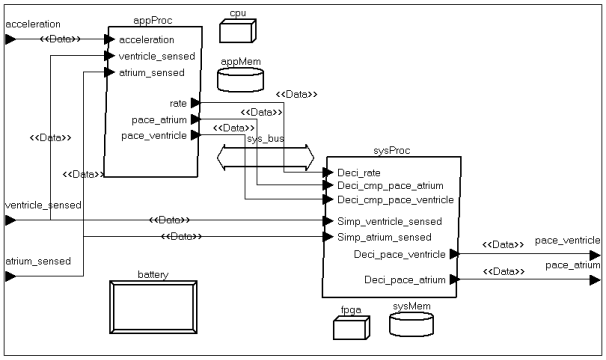

Figure 5. The design process generates a System-Level Simplex AADL Model.

3.3. System-Level Simplex Architecture Validation

We have also produced a System-Level Simplex architecture checker which will traverse an AADL model and enforce a checklist of architectural requirements. This is important because the generator does not provide a one-size-fits-all architecture, but rather an application-specific template that is further modified for the specific design. We want to guarantee that these further modifications do not violate key Simplex architectural safety requirements.

We have identified several necessary conditions required by a System-Level Simplex design that are checked by our OSATE architecture checker. The properties can be classified into resource isolation properties, data consistency properties, and data flow properties. We briefly describe each of these, along with associated failures that may occur if they are not present.

Resource isolation properties that we require include a real-time bus with an electrically-safe interface (such as the CAN bus [15]), and a power management scheme. If the bus is not real-time, correct complex controller commands may not reach the decision module in a timely fashion, which will result in degraded system performance as the safety controller will be used instead. If the bus interface is not electrically safe, the FPGA hardware may be damaged by

a short on the bus, which may damage the Simplex safety core. If the power is not managed, the complex controller may drain all the system's power, shutting down the Simplex safety core.

One data consistency property that we check is that the value received from the complex controller goes through a type-checking process. If this were not the case, a complex controller may send values of the wrong type (for example, sending the floating-point value NAN instead of a finite voltage value), which the decision module may not interpret correctly.

Data flow properties impose requirements on the connections among the System-Level Simplex components. The complex controller, for example, should not be sending data to the safety controller, only to the decision module. If these connection properties are not enforced, we can not guarantee that the architecture is actually an instance of Simplex.

Our architecture checking tool enforces each of these necessary requirements for architectural safety. The tool can be further extended to enforce additional requirements by defining new AADL properties and writing the expected invariants on these properties.

3.4. System-Level Simplex Implementation Generation

We have developed a VHDL code generator to automatically create the System-Level Simplex hardware code for both the safety controller and decision module. If these modules are described as finite-state machines in the AADL behavior annex, the corresponding VHDL code can be immediately generated. This removes an extra level of human interaction which may have lead to errors. The VHDL hardware code can then be synthesized and used to program an FPGA.

4. Case Studies

In order to demonstrate the practicality and robustness of System-Level Simplex systems, we examine two case studies in detail. First, we use our end-to-end design process to produce a cardiac pacemaker system and compare the resultant safety guarantees with those provided by a previous-generation pacemaker. Later, we apply the System-Level Simplex Architecture to a classic inverted pendulum and empirically verify fault-tolerance guarantees. In addition to the two case studies discussed in this paper, we are currently evaluating the System-Level Simplex design to provide safety for control of an autonomous tractor, in collaboration with John Deere.

Code 1 The architecture generator separates the logical Simplex system into a System-Level AADL model.

```
Transform(model) {
   newModel = copy(model);
   ss = new SimplexSwitch(model);
   ss.traverseModel(); // find simplex components

   appProc = newModel.wrapInProcess(ss.complex_ctrl);
   sysProc = newModel.wrapInProcess(ss.simple_ctrl,
                          ss.decision_module);
   ...
   appProc.bindToMemory(appMem);
   appProc.bindToProcessor(cpu);
   sysProc.bindToMemory(sysMem)
   sysProc.bindToProcessor(fpga);
   newModel.bindPower(power_source);
   ...
   newModel.createDataConnections(ss.conn);
   newModel.createBusBindings(sysBus);
   ...
   return newModel;
}
```

4.1. System-Level Simplex Design for a Cardiac Pacemaker

A cardiac pacemaker is a piece of hardware inserted into a patient's body in order to regulate his or her heart rate. Detailed designs of cardiac pacemakers have been disclosed [16], [17]. In this safety-critical application, we examine the practicality and usefulness of the System-Level Simplex Architecture, as well as the end-to-end design process.

We investigate three considerations for using the System-Level Simplex Architecture:

- Can the system be divided up into a safe controller and a complex controller, such that the most likely causes of failure are contained in the complex controller?
- Is the System-Level Simplex end-to-end design process effective in the cardiac pacemaker context?
- How do the resultant safety guarantees compare to those of existing pacemakers?

4.1.1. Dividing the Cardiac Pacemaker System. The first concern, the division of the system into complex and simple controllers, asks if the logical Simplex framework can be applied to a cardiac pacemaker. Since this is domain-specific, we examine some properties of artificial pacemakers.

The first generation of artificial pacemakers actuated the heart at a set interval. This functionality was sufficient to keep the patient alive, however, problems did arise. For example, when a healthy person walks up stairs or performs strenuous action, his heart rate increases. The first generation of pacemakers did not take this into account and patients would become dizzy and uncomfortable. Additional functionality was added to pacemakers to detect if the heart rate should be increased by monitoring the temperature of the blood, or the acceleration on the patient's body [16].

Requirements were then added on top of this to preserve smooth heart-rate transitions, rather than suddenly jumping from 65 to 120 beats per minute because of a sudden large acceleration. Additionally, modern pacemakers attempt to detect and log anomalous events with the heart to aid a doctor's diagnosis. The logged data must be retrieved, and this is done through wireless communication with an external device.

Modern pacemakers have many other requirements, however we already covered enough to apply the System-Level Simplex Architecture. The rate-adaptive pacing modes, where the heart rate changes over time, require complex functionality. The pacing rate to which we should change is a function of the current rate, as well as the past and present accelerometer readings. The safety properties we want to enforce are that the heart rate should be between a lower rate limit and an upper rate limit, and should not change by more than a doctor-specified rate-smoothing parameter. These are the properties monitored by the decision module. The safe controller is a finite-state machine that meets the safety requirements. We choose a safe controller that slows down the heart to the resting rate (lower rate limit) in a way that satisfies the rate smoothing requirement. This safety controller does not have the complex rate-adaptive functionality, but instead provides a fail-operational mode that will maintain safety for the patient.

4.1.2. Using the System-Level Simplex Design Process. The second concern addresses the effectiveness of the end-to-end System-Level Simplex design process. The inputs for the process are a finite-state machine behavioral description for the decision module and safety controller, and the logical AADL architecture description for Simplex.

As discussed in Section 3, we begin by describing both the decision module behavior and safety controller behavior as finite-state machines. The heart is then modeled in the same way, and we exhaustively test for violations of safety properties in UPPAAL (Figure 3). For example, one property we check is that the safety core will not actuate the heart unless it has been idle for a minimum time interval (to enforce a maximum heart rate), for all possible actuation commands coming from the unspecified complex controller. When this checking is complete, we describe the final finite-state machines in the AADL behavior annex.

The next step is the construction of the initial AADL model. This model, shown in Figure 4, outlines the logical Simplex connections. Our OSATE plug-in then takes this initial model and transforms it into a model for a System-Level Simplex Architecture (Figure 5). At this point the designer can modify the model as needed and run our architecture checker to make sure all System-Level Simplex Architecture requirements are met.

One item checked is power safety between the Simplex safety core and the complex controller. If power is not managed, the complex controller can drain the shared battery causing the system to fail. Although the architecture checker makes sure this constraint is satisfied, the designer must determine how to satisfy it. Our pacemaker is designed as a System-on-Chip (SoC) running on a Xilinx FPGA. Modern Xilinx FPGAs have several clock regions which can be toggled on or off [18]. By using a soft processor on the FPGA to run the complex subsystem, we can control the power consumed by disabling the processor's clock when the battery is low. In CMOS circuits, preventing transistor state changes (by stopping the clock) results in near-zero power consumption. In this way we are able to provide power isolation, and can set the appropriate power-isolation AADL property within our model. Without the architecture checker, this critical step could be overlooked.

After the architectural constraint checker validates the model, we proceed to generate the implementation for our pacemaker. The AADL behavior annex finite-state machines for both the decision module and the complex controller are put into our VHDL code generator. The generator produces synthesizable hardware VHDL code which is used to program the FPGA.

4.1.3. Comparing against Existing Pacemaker Reliability Mechanisms. The last consideration compares the existing reliability mechanisms found in one previous-generation pacemaker description [16] to the design created using the System-Level Simplex end-to-end design process. We focus on two mechanisms for enhanced reliability which were present in the pacemaker description we examined.

The first is a watchdog timer which is periodically reset during normal system execution. If the execution hangs at some point, the timer will not be reset and will timeout. The timeout triggers a high-priority interrupt which signals that an anomalous event has occurred and the system is reinitialized. Alternately, the system can be shut down as a fail-safe mechanism.

The watchdog timer mechanism is compatible with the System-Level Simplex Architecture. It provides a means to restart the system when it enters a rare error state. However, the watchdog timer does not protect the system from unsafe pacing, only system hangs. Additionally, deterministic bugs in the program will continue to restart the system, whereas a System-Level Simplex system is able to function safely in spite of deterministic bugs in the complex controller.

The other safety mechanism we examine is a redundant pacemaker system which, at first, appears to be similar to the System-Level Simplex Architecture. This system provides a simpler pacing mode without rate-adaption. This system takes control from the microprocessor when "a fault is detected in the operation of the microprocessor circuit." This component, like the System-Level Simplex Architecture, provides protection from microprocessor errors. This is a real cause of concern with this specific pacemaker design

because it uses a custom pacemaker-specific microprocessor. However, it does not provide protection from logical faults in the software. Additionally, control is switched to this system when any fault in the microprocessor is detected. In the System-Level Simplex Architecture, a hardware fault that only affects the logging mechanism (perhaps because of a rarely used instruction), one that does not compromise safety, would not trigger a change in control.

4.2. System-Level Simplex Design for Inverted Pendulum

An inverted pendulum is a classical control testbed where a rod must be maintained upright by moving a cart along a track. An inverted pendulum presents an obvious failure state when the rod falls over. We applied the System-Level Simplex Architecture to an inverted pendulum and evaluated its robustness by inserting faults and observing system robustness.

An inverted pendulum, however, does not completely lend itself to our end-to-end design process. We still generate the AADL architecture description and run it through the architecture checker to make sure the architectural constraints are met. However, control of an inverted pendulum is best done through differential equations rather than finite-state machines. This means that we can not use a finite-state machine model checker such as UPPAAL to guarantee safety. Instead, we guarantee safety through the same technique as previous Simplex applications [1]. We measure the inverted pendulum system and find a Lyapunov stability function [19]. From this, we can generate the safety controller C code using Matlab Simulink [20] and determine when the decision module should switch controllers (before the state leaves the Lyapunov stability neighborhood). The C code is then manually translated to VHDL for hardware synthesis.

Our hardware safety core resides on an externally-powered Xilinx ML505 FPGA. This FPGA contains a PCIe port which is used to communicate to a PC which runs the software portion of the architecture. The software portion uses a custom driver written for Linux. We run Linux/RK [21] as the operating system for the complex controller. Since the System-Level Simplex Architecture handles timing faults, we purposefully do not use the provided real-time scheduler. Through memory-mapped I/O, the complex controller reads the most recent angle and track position and suggests a motor voltage to the hardware-based decision module.

After constructing the system, we verified that the software-based complex controller was able to actuate the inverted pendulum as long as it did not jeopardize safety. When the pendulum's state approached the edge of the Lyapunov stability neighborhood, the safety controller took over and prevented the pendulum from collapsing. In this way, the system was able to tolerate a multitude of faults as outlined in Table 1. We outline two of these faults in detail.

Inverted Pendulum Fault-Injection	
Failure Type	Safe
No Output	√
Maximum Voltage	√
Wrong Way — Maximum Voltage	√
Time Degraded Control	√
OS Crash	√
Timing Faults	√
Computer Reboot	√

Table 1. The System-Level Simplex inverted pendulum tolerates a variety of faults.

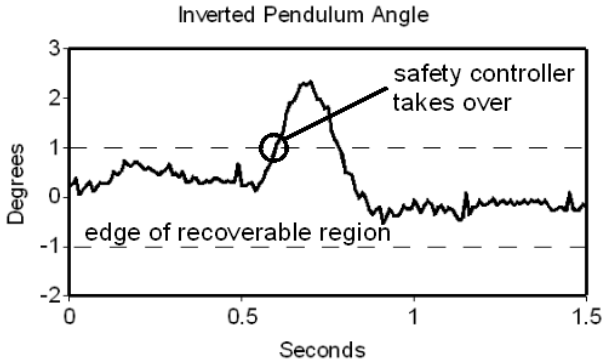

Figure 6. When the inverted pendulum state passes the edge of the recoverable region (dashed line), the safety controller takes over and prevents system collapse.

4.2.1. Wrong Way — Maximum Voltage. The pendulum should remain balanced even if the complex controller outputs a motor voltage that would normally destabilize the system. This test took an extreme case of this where we used a working inverted pendulum controller for a few seconds, and then output the maximum voltage in the direction opposite of that needed to stabilize the pendulum. The decision module detected this and switched control to the safety subsystem. The safety controller returned the pendulum to a stable state and control was again given to the complex subsystem after a few seconds. Measurements from one iteration of this process are given in Figure 6.

4.2.2. Computer Reboot. The System-Level Simplex Architecture provides protection from arbitrary operating system behavior, including rebooting the system. From the decision module's perspective, the computer rebooting is equivalent to a complex controller that sends no output. We ran this test on our inverted pendulum setup, and the pendulum remained stable throughout the reboot process. Additionally, after the computer restarted, the software-based complex controller was able to regain control of the inverted pendulum using memory-mapped I/O with the

FPGA. This is significant because a common remedy for software problems is rebooting the computer. A malfunctioning complex controller can be repaired in this fashion while the system remains stable and safe.

5. Related Work

Previous research has been performed on reliable system design. One method proposed to accomplish this has been N-version programming [22]. In this method, multiple versions of software are independently created from the same specification. Then, all are run and the result given by the majority of versions is taken as the output of the system. One drawback with this method is the lack of statistical independence of bugs [23], [24]. Additionally, for a constant amount of development effort, N-version programming is actually less reliable than focusing on a single version over a wide range of parameter values [1].

Another reliability mechanism is the recovery block concept [25]. In this approach, several alternative methods are developed. We first run the fully featured one and check if it is correct. If it is, we use it. Otherwise, we try the simpler ones. The essential difference between recovery blocks and the Simplex architecture is that the former is a backward recovery method while the latter is a forward recovery method.

A common engineering practice to increase system reliability in spite of unreliable hardware is triple modular redundancy (TMR) [12]. In this scheme, three versions of identical hardware running an identical program are run with the same input. The output is then voted upon, such that if any one of the outputs is incorrect (due to a hardware failure or random environmental interference [26]), the overall system continues to function correctly. This technique, unlike the System-Level Simplex Architecture, is powerless against errors in the logic of the program, since all three modules will produce the identically incorrect output. However, it is effective against hardware failures and can be used in conjunction with the System-Level Simplex Architecture. The resultant architecture prevents logical errors, transient faults, and hardware failure problems. To use this combined scheme, we would have three modules each with their own hardware and software portions running the System-Level Simplex Architecture with a reliable voter at the end to accumulate the results. Variations of this also are possible, for example by replicating only the safety-critical hardware subsystem and using a single microprocessor-based complex controller.

Our method is most closely related to the original Application-Level Simplex Architecture [1]–[3]. In this design, two controllers are used in software to provide reliability in spite of logical errors in the complex version. The System-Level Simplex is a novel architecture over the Application-Level Simplex in several ways. The System-Level Simplex Architecture eliminates a large body of common unverified dependencies between the safety and complex controllers, including the operating system, middleware, and microprocessor. Additionally, moving logic outside of software allows us to handle additional failure modes previously unavailable to software running within the Application-Level Simplex design, such as power and timing faults of the microprocessor-based system. The System-Level Simplex also removes computation overhead from the processor, which no longer has to run the simple controller and decision module (which could affect real-time schedulability). One drawback of the System-Level version of Simplex is that additional resources are required to run the decision module and complex controller, such as the FPGA.

Simplex (both versions) should not be regarded as a one-size-fits-all robustness approach. To guarantee safety, we must calculate a Lyapunov function in a dynamical system, or be able to exhaustively model check a finite-state machine behavioral description. Even after this step is complete, the pessimism in the decision module and the simplicity of the simple controller can unnecessarily reduce system performance when the simple controller is governing. This is the inherent trade-off a Simplex system makes in order to guarantee robustness and controllability.

6. Conclusions

We have presented the System-Level Simplex Architecture which uses hardware/software co-design to produce fault-tolerant systems. By leveraging on a simple safety controller and a decision module implemented in hardware, several types of previously unhandlable errors can be safely managed. These include failures in the complex software controller code, operating system, and microprocessor, as well as real-time temporal faults.

An end-to-end design process has been created for the architecture, which leverages on an initial AADL model to provide both an architectural and behavioral description. The output of the process is a checkable System-Level Simplex architectural model, and the corresponding VHDL hardware code.

We demonstrated the feasibility and robustness of the System-Level Simplex Architecture through two case studies involving a pacemaker and a classic inverted pendulum. The architecture is also currently being evaluated to improve the safety of autonomous tractor control, in collaboration with John Deere.

Acknowledgment

We would like to thank Rodolfo Pellizzoni for his plethora of knowledge and support related to VHDL synthesis and

Xilinx development tools. Additionally, Bach Duy Bui deserves recognition for furnishing the PCIe driver which presents the memory-mapped I/O interface for our inverted pendulum implementation.

This material is based upon work supported by John Deere and by the NSF under Awards No. CNS0237884, CNS0613665, CCF0325716. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] L. Sha, "Using simplicity to control complexity," *IEEE Softw.*, vol. 18, no. 4, pp. 20–28, 2001.

[2] ——, "Dependable system upgrade," in *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1998, p. 440.

[3] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *1996 IEEE Aerospace Applications Conference. Proceedings*. Aspen, CO: IEEE New York, NY, USA, 3–10 1996, pp. 335–46. [Online]. Available: citeseer.ist.psu.edu/ sha95evolving.html

[4] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 400–412.

[5] S. E. Institute, "The simplex distributed pilot study," www.sei.cmu.edu/ simplex/ demonstrations/ distributed.html, 1999.

[6] D. Seto, E. Ferreira, and T. F. Marz, "Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis)," Technical Report Cmu/ sei-99-Tr-020. [Online]. Available: citeseer.ist.psu.edu/ 606539.html

[7] T. I. der Rieden, "Verisoft," www.verisoft.de, 2008.

[8] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis and design language (aadl): An introduction," CMU/ SEI, Tech. Rep., 2006.

[9] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas, "The aadl behaviour annex – experiments and roadmap," in *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 377–382.

[10] A. University and U. University, "Uppaal a tool suite for verification of real-time systems," www.uppaal.com, 2008.

[11] Y. Yeh, "Triple-triple redundant 777 primary flight computer," *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1, pp. 293–307 vol.1, Feb 1996.

[12] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Research and Development*, vol. 6, pp. 200–209, 1962.

[13] P. Feiler, B. Lewis, and S. Vestal, "The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems," *In Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, October 2006.

[14] C. S. E. Institute, "An extensible open source aadl tool environment (osate)," la.sei.cmu.edu/ aadl/ downloads/ osate13/ AADLToolUserGuide1.3.0

[15] I. O. for Standardization, "Controller area network (can)," ISO 11898-1:2003, 2003.

[16] J. G. Webster, Ed., *Design of cardiac pacemakers*. Piscataway, NJ: IEEE Press, 1995, ecow.engr.wisc.edu/ cgi-bin/ get/ bme/ 762/ webster/ designofca/.

[17] B. Scientific, "Pacemaker system specifcation," sqrl.mcmaster.ca/ _SQRLDocuments/ PACEMAKER.pdf, 2007.

[18] M. Adhiwiyogo, "Virtex-4 clocking resources," www.xilinx.com/ publications/ xcellonline/ xcell_52/ xc_v4xesium52.htm, 2005.

[19] K. J. Astrom and B. Wittenmark, *Computer-controlled systems: theory and design (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[20] B. Messner and D. Tilbury, "Control tutorials for matlab," www.engin.umich.edu/ group/ ctm/ examples/ pend/ invpen.html, August 1997.

[21] R. Rajkumar, "Linux/ rk," www.cs.cmu.edu/ rajkumar/ linux-rk.html, 2008.

[22] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during program execution," in *COMPSAC 77*, 1977, pp. 149–155.

[23] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *Software Engineering*, vol. 12, no. 1, pp. 96–109, 1986. [Online]. Available: citeseer.ist.psu.edu/ knight86experimental.html

[24] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an n-version software experiment," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 238–247, 1990.

[25] B. Randell, "The evolution of the recovery block concept," in *Software Fault Tolerance*, Lyu, Ed., 1995, ch. 1, pp. 1–21. [Online]. Available: www.ece.cmu.edu/ ece849/ papers/ randell95_evolution_recovery_blocks.pdf

[26] K. Morris, "Fpgas in space," *FPGA and Programmable Logic Journal*, August 2004.