A Formal Method for Developing Provably Correct Fault-Tolerant Systems Using Partial Refinement and Composition

Ralph Jeffords, Constance Heitmeyer, Myla Archer, and Elizabeth Leonard

Naval Research Laboratory
Washington, DC 20375
{jeffords,heitmeyer,archer,leonard}@itd.nrl.navy.mil

Abstract. It is widely agreed that building correct fault-tolerant systems is very difficult. To address this problem, this paper introduces a new model-based approach for developing *masking fault-tolerant systems*. As in component-based software development, two (or more) component specifications are developed, one implementing the required normal behavior and the other(s) the required fault-handling behavior. The specification of the required normal behavior is verified to satisfy system properties, whereas each specification of the required fault-handling behavior is shown to satisfy both system properties, typically weakened, and fault-tolerance properties, both of which can then be inferred of the composed fault-tolerant system. The paper presents the formal foundations of our approach, including a new notion of *partial refinement* and two compositional proof rules. To demonstrate and validate the approach, the paper applies it to a real-world avionics example.

1 Introduction

It is widely agreed that building a correct fault-tolerant system is very difficult. One promising approach, proposed by us and others, for obtaining a high-assurance fault-tolerant system is to specify the system requirements in two phases [4, 18, 7, 19]. In the first phase, the *normal* (also called *ideal*) system behavior, the system behavior when no faults can occur, is specified. In the second phase, the no-faults assumption is removed, and the system's required fault-tolerant behavior is specified. Such an approach has many advantages. First, a specification of the normal behavior known to be correct can be reused if the design of fault-tolerance changes. Second, if the fault-tolerant system can be expressed as an extension of a system with normal behavior by adding a set of fault-handling components, the specification is easier to understand and easier to construct than a fault-tolerant system specified as a single component. Third, by applying formal specification during two separate phases, errors may be uncovered—e.g., by applying formal verification—that might otherwise be overlooked. For example, our application of two-phase specification and verification to a real-world avionics device [7] uncovered modeling errors previously unnoticed (see Section 5). Finally, specifications of the fault-handling components may be reused in other systems.

The model-based approach proposed in this paper has attributes of two other popular approaches for developing software systems. As in aspect-oriented programming [17, 16], the approach weaves certain aspects, specifically, the "fault-tolerant" aspects, into the original program. Moreover, as in component-based software development, two (or more) components are developed separately, and later composed to produce the final

implementation. This paper makes three contributions; it presents: 1) a component-based approach for developing a special class of fault-tolerant systems, called "masking" fault-tolerant systems, which uses formal specification and formal verification to obtain high confidence of system correctness; 2) a formal foundation, including a set of sound compositional proof rules, a formal notion of *fault-tolerant extension*, and a formal notion of *partial refinement* with an associated notion of *partial property inheritance*; and 3) a complete example of applying the approach to a real-world system.

The paper's organization is as follows. After defining *masking* fault-tolerance, Section 2 briefly reviews the SCR (Software Cost Reduction) method used in our example. Section 3 introduces our formal method for developing fault-tolerant systems, an extension of the approach to software development presented in [7]. To establish a formal foundation for the method, Section 4, inspired by the theory of fault tolerance in [18] and the theory of retrenchment applied to fault-tolerant systems in [5], presents our new notions of partial refinement and fault-tolerant extension, and two compositional proof rules. To demonstrate and validate our approach and to show how formal methods can be used to support the approach, Section 5 applies the method to a device controller in an avionics system [20]. Finally, Sections 6 and 7 discuss related work and present some conclusions. Although SCR is used in Section 5 to demonstrate our approach, the method and theory presented in this paper are basically applicable in any software development which specifies components as state machine models.

2 Background

2.1 Masking Fault-Tolerance

This paper focuses on *masking fault-tolerance*, a form of fault-tolerance in which the system always recovers to normal behavior after a fault occurs, so that the occurrence of faults is rendered mostly invisible, i.e., "masked." We consider two variants of masking fault tolerance. In the first variant, *transparent* masking, all safety properties [2] of the system are preserved even in the presence of faults, and the effect of faults on the system behavior is completely invisible. In the second variant, *eventual* masking, some critical subset of the set of safety properties is preserved during fault handling, though other safety properties guaranteed during normal behavior may be violated. When masking is transparent, the system's fault-tolerant behavior is a refinement of its normal behavior. For eventual masking, system behavior during fault-handling is a *degraded* version of normal behavior, and the relationship of the full fault-tolerant system behavior to normal system behavior is captured by the notions of fault-tolerant extension and partial refinement presented in Section 4.[1] The Altitude Switch (ASW) example in Section 5 illustrates both variants of masking fault-tolerance.

2.2 The SCR Requirements Method

The SCR (Software Cost Reduction) [13, 12] method uses a special tabular notation and a set of tools for formally specifying, validating, and verifying software and system requirements. See [12, 11] for a review of the SCR tabular notation, the state machine model which defines the SCR semantics, and the SCR tools.

[1] Many use "masking fault-tolerance" to refer only to what we call "transparent masking."

An important construct in SCR, the *mode class*, can be very useful in specifying the required behavior of fault-tolerant systems. Conceptually, each mode in a mode class corresponds to a “mode of operation” of the system. Thus, for example, in flight software, pilot-visible modes determine how the software reacts to a given pilot input. As shown in Section 5, modes similarly have a special role in SCR specifications of fault-tolerant systems.

3 A Formal Method for Building Fault-Tolerant Systems

This section introduces a new method for building a fault-tolerant system. Based on concepts in Parnas' Four Variable Model [21], the method is applied in two phases. In the first phase, the normal system behavior is specified and shown to satisfy a set of critical properties, most commonly, safety properties [2]. In the second phase, I/O devices, e.g., sensors and actuators, are selected, hardware and other faults which may occur are identified, and the system's *fault-tolerant* behavior is designed and specified. The fault-tolerant specification formulated in this phase is shown to satisfy 1) the critical system properties, typically weakened, which were verified in the first phase and 2) new properties specifying fault detection and fault recovery. While each phase is described below as a sequence of steps, the precise ordering of the steps may vary, and some steps may occur in parallel.

3.1 Specify the Normal System Behavior

In the first phase, the system behavior is specified under the assumption that no faults can occur, and essential system properties are formulated and verified. The “normal” behavior omits any mention of I/O devices, or of hardware faults and other system malfunctions.

Specify NAT and REQ. To represent the system's normal behavior, a state machine model of the system requirements is formulated in terms of two sets of environmental variables—monitored and controlled variables—and two relations—REQ and NAT—from Parnas' Four Variable Model [21]. Both NAT and REQ are defined on the monitored and controlled variables. NAT specifies the natural constraints on monitored and controlled variables, such as constraints imposed by physical laws and the system environment. REQ specifies the required relation the system must maintain between the monitored and controlled variables under the assumptions defined by NAT. In the first phase, an assumption is that the system can obtain perfect values of the monitored quantities and compute perfect values of the controlled variables. During this phase, the system tolerances are also defined; these may include the required precision of values of controlled variables, timing constraints imposed by REQ on the controlled variables, and timing constraints imposed by NAT.

Formulate the System Properties. In this step, the critical system properties are formulated as properties of the state machine model. If possible, these properties should be safety properties, since the second phase produces a refinement (i.e., when the system is operating normally), and safety properties are preserved under refinement [1].

Verify the System Properties. In the final step, the properties are verified to hold in the state machine model, using, for example, a model checker or theorem prover.

3.2 Specify the Fault-Tolerant Behavior

In the second phase, the assumption that the system can perfectly measure values of monitored quantities and perfectly compute values of controlled quantities is removed, and I/O devices are selected to estimate values of monitored quantities and to set values of controlled quantities. Also removed is the assumption that no faults occur. Possible faults are identified, and the system is designed to tolerate some of these faults. Finally, the fault-tolerant behavior is specified as a fault-tolerant extension (see Section 4) which adds extra behavior to handle faults and which may include new externally visible behavior, e.g., operator notification of a sensor failure.

Select I/O Devices and Identify Likely Faults. In the second phase, the first step is to select a set of I/O devices and to document the device characteristics, including identification of possible faults. Among the possible faults are faults that invalidate either sensor inputs or actuator outputs and faults that corrupt the program's computations. Examples of faults include the failure of a single sensor, the simultaneous failure of all system sensors, and the failure of a monitored variable to change value within some time interval. For practical reasons, the system is designed to respond to only some possible faults. An example of an extremely unlikely fault is simultaneous failure of all system sensors—recovery from such a massive failure is likely to be impossible. Once a set of faults is selected, a design is developed that either makes the system tolerant of a fault or reports a fault so that action may be taken to correct or mitigate the fault.

Design and Specify the Fault-Tolerant Behavior. A wide range of fault-tolerance techniques have been proposed. One example is hardware redundancy, where two or more versions of a single sensor are available, but only one is operational at a time. If the operational sensor fails, the system switches control to a back-up sensor. In another version of hardware redundancy, three (or any odd number of) sensors each sample a monitored quantity's value, and a majority vote determines the value of the quantity. Some fault-tolerance techniques make faults transparent. For example, if three sensors measure aircraft altitude, a majority vote may produce estimates of the altitude satisfying the system's tolerance requirements and do so in a transparent manner. Other techniques do not make faults transparent—for example, techniques which report a fault to an operator, who then takes some corrective action.

Verify Properties of the Fault-Tolerant Specification. In this step, the critical properties verified to hold for the normal system behavior must be shown to hold for the fault-tolerant behavior. In some cases, properties of the normal system will not hold throughout the fault-tolerant system but may remain true for only some behavior (e.g., for only the normal behavior). A new notion of partial refinement, defined in Section 4, describes the conditions which must be established for the fault-tolerant system to partially inherit properties of the normal system. In addition, new properties are formulated to describe the required behavior when a fault is detected and when the system recovers from a fault. It must then be shown that the fault-tolerant specification satisfies these new properties, which can be established as invariants with the aid of compositional proof rules, such as those presented in Section 4.2.

4 Formal Foundations

This section presents formal definitions, theoretical results, and formal techniques that support our approach to developing provably correct fault-tolerant systems. The most important concepts and results include our notions of *partial refinement* and *fault-tolerant extension*, and two proof methods for establishing properties of a fault-tolerant extension based on properties of the normal (fault-free) system behavior it extends. The first proof method is based on Theorem 1 concerning property inheritance under partial refinement; the second is based on compositional proof rules for invariants, two of which are shown in Figure 2. The section begins with general notions concerning state machines, then introduces fault-tolerance concepts, and finally, discusses additional concepts and results that apply as additional assumptions about state machines are added—first, that states are determined by the values of a set of state variables, and second, that the state machines are specified in SCR. Each concept or result presented is introduced at the highest level of generality possible. The definitions, results, and techniques of this section are illustrated in the ASW example presented in Section 5.

4.1 General definitions

To establish some terminology, we begin with the (well-known) definitions of *state machine* and *invariant property* (*invariant*, for short). As is often customary, we consider predicates to be synonymous with sets; thus, "P is a predicate on set S" ≡ "$P \subseteq S$", "$P(s)$ holds" ≡ "$s \in P$", etc.

Definition 1. State machine. *A* state machine **A** *is a triple* (S_A, Θ_A, ρ_A)*, where* S_A *is a nonempty set of states,* $\Theta_A \subseteq S_A$ *is a nonempty set of* initial *states, and* $\rho_A \subseteq S_A \times S_A$ *is a set of* transitions *that contains the stutter step* (s_A, s_A) *for every* s_A *in* S_A*. A state* $s_A \in S_A$ *is* reachable *if there is a sequence* $(s_0, s_1),\ (s_1, s_2), \ldots (s_{n-1}, s_n)$ *of transitions in* ρ_A *such that* s_0 *is an initial state and* $s_n = s_A$*. A transition* $(s_A, s'_A) \in \rho_A$ *is a* reachable transition *if* s_A *is a reachable state. Reachable states/transitions of* **A** *are also called* A-reachable states/transitions.

Definition 2. One-state and two-state predicates/invariants. *Let* **A** = (S_A, Θ_A, ρ_A) *be a state machine. Then a* one-state predicate *of* **A** *is a predicate* $P \subseteq S_A$*, and a* two-state predicate *of* **A** *is a predicate* $P \subseteq S_A \times S_A$*. A one-state (two-state) predicate* P *is a state (transition)* invariant *of* **A** *if all reachable states (transitions) of* **A** *are in* P*.*

We next define two notions that describe how two state machines (e.g., two models of a system) may be related. The well known notion of *refinement* is especially useful in the context of software development because the existence of a refinement mapping from a state machine **C** to a state machine **A** at a more abstract level permits important properties—including all safety properties (and hence all one-state and two-state invariants)—proved of **A** to be inferred of **C**. A new notion, which we call *partial refinement*, is a generalization of refinement useful in situations where the approximation by a detailed system model to a model of normal system behavior is inexact.

Definition 3. Refinement. *Let* **A** = (S_A, Θ_A, ρ_A) *and* **C** = (S_C, Θ_C, ρ_C) *be two state machines, and let* $\alpha : S_C \to S_A$ *be a mapping from the states of* **C** *to the states of* **A***. Then* α *is a* refinement mapping *if 1) for every* s_C *in* Θ_C*,* $\alpha(s_C)$ *is in* Θ_A*, and 2)* $\rho_A(\alpha(s_C), \alpha(s'_C))$ *for every pair of states* s_C, s'_C *in* S_C *such that* $\rho_C(s_C, s'_C)$*.*

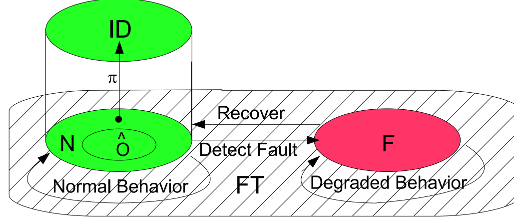

Fig. 1. Transitions in the fault-tolerant system **FT**.

Definition 4. Partial Refinement. *Let* **A** $= (S_A, \Theta_A, \rho_A)$ *and* **C** $= (S_C, \Theta_C, \rho_C)$ *be two state machines and* $\alpha : S_C \overset{\circ}{\rightarrow} S_A$ *be a partial mapping from states of* **C** *to states of* **A***. Then* α *is a* partial refinement mapping *if 1) for every* s_C *in* Θ_C*,* $\alpha(s_C)$ *is defined and in* Θ_A*, and 2)* $\rho_A(\alpha(s_C), \alpha(s'_C))$ *for every pair of states* s_C, s'_C *in the domain* $\alpha^{-1}(S_A)$ *of* α *such that* $\rho_C(s_C, s'_C)$*. When a partial refinement mapping* α *exists from* **C** *to* **A***, we say that* **C** *is a* partial refinement *of* **A** *(with respect to* α*).*

The notions of *vulnerable state* and *vulnerable transition* are useful (see Theorem 1) in describing the circumstances under which properties proved of a state machine **A** can be partially inferred for a state machine **C** that is a partial refinement of **A**.

Definition 5. Vulnerable states and vulnerable transitions. *Let* **A** $= (S_A, \Theta_A, \rho_A)$ *and* **C** $= (S_C, \Theta_C, \rho_C)$ *be two state machines, and let* $\alpha : S_C \overset{\circ}{\rightarrow} S_A$ *be a partial refinement. Then a state* s_C *in the domain of* α *is* vulnerable *if there exists a state* s'_C *in* S_C *such that* $\rho_C(s_C, s'_C)$ *but the transition* (s_C, s'_C) *does not map under* α *to a transition in* **A** *(in which case we refer to* (s_C, s'_C) *as a* vulnerable transition*).*

4.2 Concepts for fault tolerance

Our method for including fault tolerance in the software development process described in Section 3 begins with a model **ID** of the normal (software) system behavior. In the next phase, **ID** is used as a basis for constructing a model **FT** of the system that is a *fault-tolerant extension* of **ID** in the following sense:

Definition 6. Fault-tolerant extension. *Given a state machine model* **ID** *of a system, a second state machine model* **FT** *of the system is a* fault-tolerant extension *of* **ID** *if:*

- *the state set* S_{FT} *of* **FT** *partitions naturally into two sets: 1)* N*, the set of normal states, which includes* Θ_{FT} *and 2)* F*, the set of fault-handling states that represent the system state after a fault has occurred, and*
- *there is a map* $\pi : N \rightarrow S_{ID}$ *and a two-state predicate* $O \subseteq N \times N$ *for* **FT** *such that* $\pi(\Theta_{FT}) \subseteq \Theta_{ID}$ *and* $s_1, s_2 \in N \wedge O(s_1, s_2) \ \wedge \ \rho_{FT}(s_1, s_2) \Rightarrow \rho_{ID}(\pi(s_1), \pi(s_2))$*.*

The map π *and predicate* O *are, respectively, the* normal state map *and* normal transition predicate *for* **FT**.

Figure 1 illustrates the structure of **FT** and its relationship to **ID**. There are five classes of transitions in **FT**:

1. transitions from N to N that map to transitions in **ID** (Normal Behavior),
2. transitions from N to N that do not map to transitions in **ID** (not shown),
3. transitions from N to F (Fault Detection),
4. transitions from F to F (Degraded Behavior), and
5. transitions from F to N (Fault Recovery).

Remark 1. When **FT** is a fault-tolerant extension of **ID**, π is a partial refinement mapping from the state machine $(S_{FT}, \Theta_{FT}, O \cap \rho_{FT})$ to **ID**. Further, if **FT** has no transitions of class 2, class 3 represents all vulnerable transitions in **FT**, and $\pi : S_{FT} \overset{\circ}{\rightarrow} S_{ID}$ is a partial refinement mapping from **FT** to **ID**.

Even when π is not a partial refinement from **FT** to **ID**, there is still guaranteed to be a partial refinement from **FT** to **ID** whose domain can be defined in terms of the normal transition predicate O in Definition 6. In particular, given O, let $\hat{O}$ be the one-state predicate for **FT** defined by:

$$\hat{O}(s_1) \triangleq (\forall s_2 \in S_{FT} : \rho(s_1, s_2) \Rightarrow O(s_1, s_2))$$

(It is easily seen, as indicated in Figure 1, that $\hat{O} \subseteq N$.) Then, for any state $s \in S_{\mathbf{FT}}$, $\hat{O}(s)$ implies that all transitions in **FT** from s map to transitions in **ID**. Therefore, restricted to the set $\hat{O}$, the map π is a partial refinement map from **FT** to **ID**.

If (s_1, s_2) is a transition in **FT** of class 5, we refer to s_2 as a *reentry state*. Further, if (s_1, s_2) is of class 2, we refer to s_2 as an *exceptional target state*. By a simple inductive argument, we have:

Lemma 1. *If every reentry state and every exceptional target state in N maps under π to a reachable state in* **ID***, then every* **FT***-reachable state in N maps under π to a reachable state in* **ID***, and every* **FT***-reachable transition in $\hat{O} \subseteq N$ maps under π to a reachable transition in* **ID***.*

Using the notation above, we can now state:

Theorem 1. Property inheritance under partial refinement. *Let* **FT** *be a fault-tolerant extension of* **ID** *in which every reentry or exceptional target state maps under π to a reachable state in* **ID***. Then 1) for every one-state invariant P of* **ID***, $\Phi(P) \triangleq P \circ \pi$ holds for every* **FT***-reachable state in N, and 2) for every two-state invariant P of* **ID***, $\Phi(P) \triangleq P \circ (\pi \times \pi)$ holds for every non-vulnerable reachable transition of* **FT** *from a state in N (thus, in particular, for every reachable transition of* **FT** *from a state in $\hat{O}$).*

As shown below, the fault-tolerant ASW behavior is a fault-tolerant extension of the normal ASW behavior with natural definitions for N and F (see Section 5), π (see Section 4.3), and O (see Section 4.4) such that all transitions from N to N are of class 1. Further, we have proven formally that all reentry states in the fault-tolerant version of the ASW are reachable, and there are no exceptional target states. Hence, for the ASW, Theorem 1 can be used to deduce properties of **FT** from properties of **ID**.

In general, however, to supplement Theorem 1, a method is needed for establishing properties of **FT** in the case when it is difficult or impossible to establish that all reentry states and exceptional target states in **FT** map under π to reachable states of **ID**. For this purpose, we provide *compositional proof rules* analogous to those in [15]. We first define what it means for a predicate to *respect* a mapping:

$$
\begin{array}{ll}
(1) & Q \text{ is a one-state predicate for } \mathbf{FT} \text{ such that } Q \text{ respects } \pi \\
(2) & \pi(\Theta_{FT}) \subseteq \Theta_{ID} \subseteq \pi(Q) \\
(3) & s_1, s_2 \in S_{ID} \;\wedge\; \pi(Q)(s_1) \;\wedge\; \rho_{ID}(s_1, s_2) \quad \Rightarrow \quad \pi(Q)(s_2) \\
(4) & s_1, s_2 \in S_{FT} \;\wedge\; \rho_{FT}(s_1, s_2) \quad \Rightarrow \quad [(Q(s_1) \;\wedge\; \neg O(s_1, s_2)) \quad \Rightarrow \quad Q(s_2)] \\
(5) & s_1, s_2 \in N \subset S_{FT} \wedge \rho_{FT}(s_1, s_2) \quad \Rightarrow \quad [O(s_1, s_2) \quad \Rightarrow \quad \rho_{ID}(\pi(s_1), \pi(s_2))] \\
\hline
\vdash & Q \text{ is a state invariant of } \mathbf{FT}
\end{array}
$$

$$
\begin{array}{ll}
(1) & P \text{ and } Q \text{ are two-state predicates for } \mathbf{FT} \text{ such that } P \Rightarrow Q \;\wedge\; P \text{ respects } \pi \\
(2) & s_1, s_2 \in S_{ID} \;\wedge\; \rho_{ID}(s_1, s_2) \quad \Rightarrow \quad ((\pi \times \pi)(P))(s_1, s_2) \\
(3) & s_1, s_2 \in S_{FT} \;\wedge\; \rho_{FT}(s_1, s_2) \quad \Rightarrow \quad [\neg O(s_1, s_2) \Rightarrow Q(s_1, s_2)] \\
(4) & s_1, s_2 \in N \subset S_{FT} \wedge \rho_{FT}(s_1, s_2) \quad \Rightarrow \quad [O(s_1, s_2) \quad \Rightarrow \quad \rho_{ID}(\pi(s_1), \pi(s_2))] \\
\hline
\vdash & Q \text{ is a transition invariant of } \mathbf{FT}
\end{array}
$$

Fig. 2. Proof rules for state and transition invariants of **FT**.

Definition 7. *Let* $\pi : S_1 \rightarrow S_2$ *be a mapping from set* S_1 *to set* S_2*. Then 1) a predicate* Q *on* S_1 respects π *if for all* $s, \hat{s}$ *in* S_1*,* $Q(s) \wedge (\pi(s) = \pi(\hat{s})) \Rightarrow Q(\hat{s})$*, and 2) a predicate* Q *on* $S_1 \times S_1$ respects π *if for all* $s, \hat{s}, s', \hat{s}'$ *in* S_1*,* $Q(s, s') \wedge (\pi(s) = \pi(\hat{s})) \wedge (\pi(s') = \pi(\hat{s}')) \Rightarrow Q(\hat{s}, \hat{s}')$.

Figure 2 gives proof rules for establishing that a one-state (two-state) predicate Q on **FT** is a state (transition) invariant of **FT**. Note that line (5) of the first proof rule and line (4) in the second proof rule are part of the definition of fault-tolerant extension.

4.3 Fault tolerance concepts in terms of state variables

When the states of a state machine are defined by a vector of values associated with a set of state variables (as is true, for example, in SCR specifications), it is possible to interpret the concepts in Section 4.2 more explicitly. In particular, constructing a fault tolerant system model **FT** from a normal system model **ID** is usually done by adding any new variables, new values of types of existing variables, and new transitions needed to describe the triggering and subsequent handling of faults. We will refer to the original variables as *normal* variables, and the added variables as *fault-tolerance* variables; for any normal variable, we will refer to its possible values in **ID** as *normal* values, and any new possible values added in **FT** as *extended* values. In this terminology, the states in $N \subseteq S_{FT}$ are those for which all normal variables have normal values. The map $\pi : N \rightarrow S_{ID}$ can then simply be chosen to be the projection map with respect to the normal variables.

Although Definition 2 represents predicates abstractly as sets when states are determined by the values assigned to state variables, most predicates of interest can be represented syntactically as relations among state variables and constants. Further, on a syntactic level, the map(s) Φ defined in Theorem 1 will be the identity.

4.4 Modeling fault tolerance in SCR

As shown in Section 5.2 below, several aspects of an SCR specification can be used to advantage in defining **FT** as a fault-tolerant extension of a normal system specification **ID** in the sense of Definition 6. These aspects include mode classes, tables to define the behavior of individual variables, and the description of transitions in terms of events.

We call a fault-tolerant extension **FT** of **ID** obtained by the technique of Section 5.2 *simple* if any row splits in the table of any normal variable result in new rows defining updated values of the variable that are either the same as in the original row for **ID** or are among the extended values for that variable. (For example, row 3 of Table 1 is split into rows 3a and 3b of Table 6.) In the terminology of Definition 6, in a simple fault-tolerant extension, every transition from N in **FT** either maps under π to a normal transition in **ID** or is a transition from N to F (class 3). It is not difficult to prove the following:

Theorem 2. *For any simple fault-tolerant extension* **FT** *of* **ID***, the normal state map* π *is a partial refinement mapping and one can choose the normal transition predicate to be*

$$O(s_1, s_2) \triangleq N(s_1) \wedge N(s_2).$$

Thus, since the predicate N can be expressed simply as an assertion that no normal variable has an extended value, it is possible in the context of SCR to compute O for any **FT** defined as a simple fault-tolerant extension of **ID**.[2]

5 Example: Altitude Switch (ASW)

This section shows how the method presented in Section 3 can be applied using SCR to a practical system, the Altitude Switch (ASW) controller in an avionics system [20]. The goal of the ASW example is to demonstrate the specification of a system's normal behavior **ID** and the separate specification of its fault-tolerant behavior **FT** as a simple fault-tolerant extension. This is in contrast to [7], which presents an earlier SCR specification of the ASW behavior, whose goal was to demonstrate the application of Parnas' Four Variable Model to software development using SCR.

The primary function of the ASW is to power on a generic Device of Interest (DOI) when an aircraft descends below a threshold altitude. In some cases, the pilot can set an inhibitor button to prevent the powering on of the DOI. The pilot can also press a reset button to reinitialize the ASW. Fault-tolerance is supported by three sensors and the system clock. If certain events occur (e.g., all three sensors fail for some time period), the system enters a fault mode and may take some action (e.g., turn on a fault indicator lamp). Recovery from a fault occurs when the pilot resets the system.

Sections 5.1 and 5.2 describe the results of applying our method to the specification and verification of both the normal and the fault-tolerant ASW behavior. Section 5.2 also shows how the theoretical results in Section 4 can be used to prove properties of the ASW's fault-tolerant behavior **FT**. Starting from property P_2 of the normal ASW behavior, our results about property inheritance allow us to derive $\tilde{P}_2$, a weakening of P_2, which holds in **FT**, while our compositional proof rules can be used to show that $\hat{P}_2$, a different weakening of P_2, also holds in **FT**. Table 4 defines both P_2 and $\hat{P}_2$. Property $\tilde{P}_2$ is defined in Section 5.2.

[2] We have also shown that O can be automatically computed for some examples in which **FT** is not a simple fault-tolerant extension of **ID**.

Row No.	Old Mode	Event	New Mode
1	init	@F(mInitializing)	standby
2	standby	@T(mReset)	init
3	standby	@T(mAltBelow) WHEN (NOT mInhibit AND mDOIStatus = off)	awaitDOIon
4	awaitDOIon	@T(mDOIStatus = on)	standby
5	awaitDOIon	@T(mReset)	init

Table 1. Mode transition table for mcStatus.

Mode in mcStatus	cWakeUpDOI
init, standby	*False*
awaitDOIon	*True*

Table 2. cWakeUpDOI cond. table.

5.1 Specify and Verify the Normal Behavior of the ASW

To characterize the normal behavior ID of the ASW, this section presents a state machine model of the ASW's normal behavior expressed in terms of NAT and REQ, and a set of critical system properties which are expected to hold in the model.

Specify NAT and REQ. The normal ASW behavior is specified in terms of 1) controlled and monitored variables, 2) environmental assumptions, 3) system modes and how they change in response to monitored variable changes, and 4) the required relation between the monitored and controlled variables. The relation NAT is defined by 1) and 2) and the relation REQ by 3) and 4).

The ASW has a single controlled variable cWakeUpDOI, a boolean, initially false, which signals the DOI to power on, and six monitored variables: 1) mAltBelow, true if the aircraft's altitude is below a threshold; 2) mDOIStatus, which indicates whether the DOI is on; 3) mInitializing, true if the DOI is initializing; 4) mInhibit, which indicates whether powering on the DOI is inhibited; 5) mReset, true when the pilot has pressed the reset button; and 6) mTime, the time measured by the system clock. The ASW also has a single mode class mcStatus containing three system modes: 1) init (system is initializing), 2) awaitDOIon (system has requested power to the DOI and is awaiting a signal that the DOI is operational), and 3) standby (all other cases).

Table 1 defines the ASW mode transitions. Once initialization is complete (event @F(mInitializing) occurs), the system moves from init to standby. It returns to init when the pilot pushes the reset button (@T(mReset) occurs). The system moves from standby to awaitDOIon when the aircraft descends below the threshold altitude (@T(mAltBelow) occurs), but only when powering on is not inhibited, and the DOI is not powered on. Once the DOI signals that it is powered on (@T(mDOIStatus = on) occurs), the system goes from awaitDOIon to standby. Table 2 defines the value of the controlled variable cWakeUpDOI as a function of the mode class mcStatus. If mcStatus = awaitDOIon, then cWakeUpDOI is *True*; otherwise, it is *False*.

Name	System	Formal Statement
A_1	**ID**, **FT**	$(\texttt{mTime}' - \texttt{mTime}) \in \{0, 1\}$
A_2	**ID**	$\texttt{DUR}(\texttt{mcStatus} = \texttt{init}) \leq \texttt{InitDur}$
A_3	**ID**	$\texttt{DUR}(\texttt{mcStatus} = \texttt{awaitDOIon}) \leq \texttt{FaultDur}$
A_4	**FT**	$\texttt{DUR}(\texttt{cFaultIndicator} = \texttt{on}) \leq \texttt{FaultDur}$

Table 3. ASW Assumptions.

The relation NAT for ASW contains three assumptions, A_1, A_2, and A_3, each a constraint on the system timing (see Table 3).[3] The first assumption, A_1, states that time never decreases and, if time increases, it increases by one time unit.[4] Assumptions A_2 and A_3 define constraints on the time that the system remains in specified modes. To represent these constraints, we require SCR's DUR operator. Informally, if c is a condition and k is a positive integer, the predicate $\texttt{DUR}(c) = k$ holds at step i if in step i condition c is true and has been true for exactly k time units. A_2 requires that the ASW spend no more than InitDur time units initializing, while A_3 requires the system to power on the DOI in no more than FaultDur time units.

Specify the ASW Properties. Table 4 defines two required properties, P_1 and P_2, of the ASW's normal behavior. P_1, a safety property, states that pressing the reset button always causes the system to return to the initial mode. P_2, another safety property, specifies the event and conditions that must hold to wake up the DOI. A user can execute the SCR invariant generator [14] to derive a set of state invariants from an SCR specification. Such invariants may be used as auxiliary properties in proving other properties, such as P_1 and P_2. Applying the invariant generator to the specification of the normal ASW behavior (defined by Table 1, Table 2, and assumptions A_1-A_3) automatically constructs the state invariant H_1, which is defined in Table 5.

Verify the ASW Properties. The property checker Salsa [8] easily verifies that the specification of the ASW's normal behavior satisfies P_1 and P_2. Completing the proof of P_2 requires the auxiliary H_1; proving P_1 requires no auxiliaries.

Name	System	Formal Statement
P_1	**ID**, **FT**	$@\texttt{T}(\texttt{mReset}) \Rightarrow \texttt{mcStatus}' = \texttt{init}$
P_2	**ID**	$\texttt{mcStatus} = \texttt{standby} \wedge @\texttt{T}(\texttt{mAltBelow}) \wedge \neg\texttt{mInhibit}$
		$\wedge\, \texttt{mDOIStatus} = \texttt{off} \Rightarrow \texttt{cWakeUpDOI}'$
$\hat{P}_2$	**FT**	$\texttt{mcStatus} = \texttt{standby} \wedge @\texttt{T}(\texttt{mAltBelow}) \wedge \neg\texttt{mInhibit}$
		$\wedge\, \texttt{mDOIStatus} = \texttt{off} \wedge \neg\texttt{mAltimeterFail} \Rightarrow \texttt{cWakeUpDOI}'$
G_1	**FT**	$\texttt{mAltimeterFail} \wedge \texttt{mcStatus} = \texttt{standby} \Rightarrow \texttt{mcStatus}' \neq \texttt{awaitDOIon}$
G_2	**FT**	$\texttt{mcStatus} = \texttt{fault} \Rightarrow \texttt{mcStatus}' = \texttt{init} \vee \texttt{mcStatus}' = \texttt{fault}$

Table 4. ASW Properties.

[3] In Tables 3-5, assumptions and properties of *both* the normal (**ID**) and fault-tolerant (**FT**) systems are presented. Any row in these tables that applies only to **FT** is described in Section 5.2.

[4] The primed variable $\texttt{mTime}'$ in Table 3 and other primed expressions refer to the expression's value in the new state; any unprimed expression refers to the expression's value in the old state.

Name	System	Formal Statement
H_1	**ID**, **FT**	$(\texttt{mcStatus} = \texttt{awaitDOIon}) \;\Leftrightarrow\; \texttt{cWakeUpDOI}$
J_2	**FT**	$\texttt{cFaultIndicator} = \texttt{on} \;\Leftrightarrow\; \texttt{mcStatus} = \texttt{fault}$
J_3	**FT**	$\texttt{DUR}(\texttt{cFaultIndicator} = \texttt{on}) \neq 0 \;\Rightarrow\; \texttt{cFaultIndicator} = \texttt{on}$

Table 5. ASW State Invariants.

5.2 Specify and Verify the Fault-Tolerant Behavior of the ASW

This section describes how the normal behavior of the ASW can be refined to handle faults. First, the I/O devices are selected. Next, the faults that the ASW system will be designed to handle are identified, and the fault-tolerant and failure notification behavior of the ASW are specified. Finally, new ASW properties are formulated to capture the required fault-tolerant behavior, and these new properties as well as the ASW properties proven for the normal behavior, possibly reformulated, are proven to hold in the fault-tolerant specification.

Select the ASW I/O Devices. To estimate whether the aircraft is below the threshold altitude, three altimeters are selected, one analog and the other two digital. For a description of the other I/O devices selected for the ASW, see [7].

Identify Likely ASW Faults. The ASW is designed to tolerate three faults: 1) the failure of all three altimeters, 2) remaining in the initialization mode too long, and 3) failure to power on the DOI on request within some time limit. All three faults are examples of eventual masking—the system enters a fault handling state but eventually recovers to normal behavior. To notify the pilot when a fault occurs, the ASW turns on a Fault Indicator lamp. The ASW is also designed to handle a single altimeter failure; it uses the remaining two altimeters to determine whether the aircraft is below the threshold altitude. This is an example of masking where the fault is transparent at the system level—the system never enters a fault handling state when only one altimeter fails.

Specify ASW Fault-Tolerant Behavior. Generally, adding behavior related to fault detection, notification, and handling to the specification of normal system behavior requires new monitored variables to detect faults, new controlled variable to report the occurrence of faults, and new values for any variable, and, in particular, new fault modes in mode classes. To define the additional behavior in SCR, tables defining the new variables are added, and tables defining the existing variables are modified and extended.

Adding fault-handling and fault notification to the normal ASW specification **ID** requires 1) a new monitored variable `mAltimeterFail` to signal the failure of all three altimeters,[5] 2) a new controlled variable `cFaultIndicator` to notify the pilot of a fault by turning on a lamp, 3) a new mode `fault` to indicate the detection of a fault, 4) a new table defining `cFaultIndicator`, and 5) the modification and extension of two tables: the table defining the controlled variable `cWakeUpDOI` and the mode transition table defining the mode class `mcStatus`. The final step removes assumptions A_2 and A_3, thus allowing the fault-tolerant system to suffer from these faults.

[5] Because this paper focuses on the fault-tolerance aspects of the ASW, the details of how `mAltimeterFail` is computed are omitted. For these details, see [7].

	Row No.	Old Mode	Event	New Mode
†	3a	standby	@T(mAltBelow) WHEN (NOT mInhibit AND mDOIStatus = off) **AND NOT mAltimeterFail**	awaitDOIon
→	3b	standby	@T(mAltBelow) WHEN (NOT mInhibit AND mDOIStatus = off) **AND mAltimeterFail**	fault
→	6	init	@T(DUR(mcStatus = init) > InitDur)	fault
→	7	awaitDOIon	@T(DUR(mcStatus = awaitDOIon) > FaultDur) OR @T(DUR(mAltimeterFail) > FaultDur)	fault
⤳	8	fault	@T(mReset)	init

Table 6. Fault Handling Modifications for Mode Transition Table in Table 1.

Mode in mcStatus	cFaultIndicator
init, standby, awaitDOIon	off
fault	on

Table 7. New table defining cFaultIndicator.

Mode in mcStatus	cWakeUpDOI
init, standby, fault	*False*
awaitDOIon	*True*

Table 8. Revised table for cWakeUpDOI.

To define a new mode transition table capturing fault detection and recovery, the mode transition table for the ASW normal behavior, Table 1, is replaced with a new mode transition table, containing rows 1, 2, 4, and 5 of Table 1 and rows 6, 7, and 8 of Table 6, and replacing row 3 of Table 1 with rows 3a and 3b of Table 6. In the new table, a fault is detected 1) if the system takes more than InitDur time units to initialize (replaces the deleted A_2), 2) if the DOI takes more than FaultDur time units to power up (replaces the deleted A_3), or 3) if all three altimeters have failed for more than FaultDur time units. Three rows of Table 6 (rows 3b, 6, and 7), each marked by a simple arrow, indicate the detection of the three faults. The system recovers from a fault when the pilot presses mReset in response to the Fault Indicator lamp turning on. To represent recovery, a new transition from fault to init, triggered by @T(mReset), is added (row 8, marked by a squiggly arrow). To force the system to recover within some bounded time, a new assumption A_4 (see Table 3) is that the pilot always responds to a failure notification by pushing reset within some time limit. To complete the new table, row 3 of Table 1 is split into row 3a and row 3b based on whether mAltimeterFail is true. If true, the system goes to fault (row 3b); otherwise, it goes to awaitDOIon as in the normal specification **ID** (row 3a, marked by a dagger).

To indicate when the Fault Indicator lamp is on, a new table, Table 7, is defined to indicate that cFaultIndicator is on when the system is in the fault mode and off otherwise. The last step is to add the fault mode to the set of modes which assign the value *false* to the table defining cWakeUpDOI (see Table 8).

Adding the new mode fault to the specification allows a normal state in the ASW to be distinguished from a fault handling state. In particular, we define a state predicate N, where N : mcStatus $\neq$ fault, and a second state predicate F, where F : mcStatus = fault.

Verify the ASW Fault-Tolerance Properties. The safety properties, P_1 and P_2, properties of the specification **ID** of normal behavior, are also included as candidate properties of the fault-tolerant version **FT** of the ASW. In addition, safety properties, G_1 and G_2, defined in Table 4, represent part of the required fault-tolerant behavior of the ASW [9]. To support the proofs of the properties P_1, P_2, G_1, and G_2, the SCR invariant generator is applied to the fault-tolerant specification. Of the two invariants generated, the first corresponds exactly to H_1, the invariant generated previously from the normal specification **ID**; the second invariant J_2 is defined in Table 5. The third invariant J_3, also defined in Table 5, is a property of the `DUR` operator. Using J_2 and J_3 as auxiliary invariants, we used Salsa to check the fault-tolerant specification **FT** for all properties listed in Table 4. All but P_2 were shown to be invariants. Thus the required behavior represented by P_2 fails in **FT** (that is, when all altimeters fail). Applying Theorem 1 from Section 4, we can show that **FT** inherits the weakened property $\tilde{P}_2 \triangleq N' \Rightarrow P_2$ from property P_2 of **ID**. In addition, the second compositional proof rule from Section 4 with $P = P_2$ provides an alternate way to show that $\hat{P}_2$, a weakened version of P_2, holds in **FT**. (See Table 4 for the definition of $\hat{P}_2$.)

To further evaluate the ASW specifications, we checked additional properties, e.g., the property $\texttt{DUR}(\texttt{mcStatus} = \texttt{standby} \wedge \texttt{mAltimeterFail}) \leq \texttt{FaultDur}$, whose invariance guarantees that the ASW never remains in $\texttt{mcStatus} = \texttt{standby}$ too long. Failure to prove this property led to the discovery (via simulation) that the ASW could remain in mode `standby` forever—not a desired behavior. Although our specification does not fix this problem, the example shows how checking properties is a useful technique for discovering errors in specifications.

6 Related Work

Our model fits the formal notion of masking fault-tolerance of [18], but rather than expressing recovery as a liveness property, we use bounded liveness, which is more practical. Other compositional approaches to fault-tolerance describe the design of fault-tolerant detectors and correctors [4] and the automatic generation of fault-tolerant systems [18, 3]. Our notion of fault-tolerant extension is most closely related to the notion of retrenchment formulated by Banach et al. [6] and the application of retrenchment to fault-tolerant systems [5]. General retrenchment is a means of formally expressing normal and exceptional behavior as a formula of the form $A \Rightarrow B \vee C$, where $A \Rightarrow B$ is true for the normal cases, and $A \Rightarrow C$ is true for the exceptional cases. Our concept of the relation of fault-tolerant behavior to normal behavior can also be described in this form: $\rho_{\mathrm{FT}}(s_1, s_2) \Rightarrow (O(s_1, s_2) \wedge \rho_{\mathrm{ID}}(\pi(s_1), \pi(s_2)) \vee \neg O(s_1, s_2) \wedge \gamma(s_1, s_2))$, where γ is derived from the transitions of classes 2–5. The novelty of our approach is recognition that this disjunction may be expressed equivalently as the conjunction of two implications, $\rho_{\mathrm{FT}}(s_1, s_2) \wedge O(s_1, s_2) \Rightarrow \rho_{\mathrm{ID}}(\pi(s_1), \pi(s_2))$ and $\rho_{\mathrm{FT}}(s_1, s_2) \wedge \neg O(s_1, s_2) \Rightarrow \gamma(s_1, s_2)$, thus providing the basis for our theory of partial refinement and the development of compositional proof rules.

In [19], Liu and Joseph describe *classical refinement* of fault-tolerant systems as well as refinement of timing and scheduling requirements. Classical refinement is well-suited to implementation of "transparent masking fault-tolerance," often using redundancy, and contrasts with eventual masking fault-tolerance, which tolerates weaker in-

variant properties when the system is faulty (i.e., has degraded performance), and thus requires a different approach such as partial refinement.

Our extension of "normal" behavior with added fault-tolerant behavior may be viewed as a transformation of the normal system. A number of researchers, e.g. [19, 10], apply the transformational approach to the development of fault-tolerant systems. This approach is also found in Katz' formal treatment of aspect-oriented programming [16]. In addition, Katz describes how various aspects affect temporal logic properties of a system and defines a "weakly invasive" aspect as one implemented as code which always returns to some state of the underlying system. The relationship of a "weakly invasive" aspect to the underlying system is analogous to the relationship of F to N in Figure 1 when there are no exceptional target states and every entry state maps under π to a reachable state in **ID**. In this case, an analog of our Theorem 1 would hold for the augmented system.

7 Conclusions

This paper has presented a new method, based on Parnas' Four Variable Model, for specifying and verifying the required behavior of a fault-tolerant system; provided a theory of partial refinement and fault-tolerant extension, and a set of compositional proof rules, as a foundation for the method; and demonstrated how the SCR language and tools can be used to support the new method as a structured alternative to the ad hoc construction and monolithic verification of fault-tolerant systems. Like Banach's theory of retrenchment, our theory of partial refinement and fault-tolerant extension applies not only to fault-tolerant systems, but more generally to all systems with both normal and exceptional behavior.

One major benefit of the compositional approach presented here is that it separates the task of specifying the normal system behavior from the task of specifying the fault-tolerant behavior, thus simplifying the specification of such systems and making their specifications both easier to understand and easier to change. The theory in Section 4 provides the basis for formulating additional compositional proof rules and vulnerability analyses, both topics for future research. We also plan to explore the utility of our approach for fault-tolerance techniques other than masking. For example, omitting recovery results in a method which applies to fail-safe fault-tolerance.

Formal proofs of state and transition invariants capturing desired system behavior, together with properties derived from partial refinement and verified using our compositional proof rules, should lead to high confidence that the specification of a given fault-tolerant system is correct. Our new approach is supported by the SCR toolset, where increasing confidence of correctness is supported by simulation, model-checking, and proofs of invariants. In future research, we plan to explore the automatic construction of efficient source code from the **FT** specification using the SCR code generator [22] and other code synthesis techniques.

Acknowledgments

The authors thank the anonymous referees for their helpful comments as well as Sandeep Kulkarni for useful discussions on applying SCR to fault-tolerant systems. This research was funded by the Office of Naval Research.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
3. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Proc. PODC'98*, pages 173–182, 1998.
4. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Trans. Softw. Eng.*, 24(1):63–78, Jan. 1998.
5. R. Banach and R. Cross. Safety requirements and fault trees using retrenchment. In M. Heisel, P. Liggesmeyer, and S. Wittman, editors, *Proc. SAFECOMP-04*, 2004.
6. R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. *Sci. Comput. Prog.*, 67:301–329, 2007.
7. R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Proc. 19th Digital Avionics Sys. Conf.*, 2000.
8. R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Berlin, 2000.
9. A. Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State Univ., East Lansing, MI, 2005.
10. F. C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *J. Univ. Comput. Sci*, 5(10), 1999.
11. C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *Computer Systems Science and Engineering*, 20(1):19–35, Jan. 2005.
12. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.
13. K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6, 1980.
14. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Eng.*, 1998.
15. R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements. In *ESEC/FSE-11: Proc. 9th Euro. Softw. Eng. Conf./11th ACM SIGSOFT Int. Symp. on Foundations of Softw. Eng.*, pages 28–37, 2003.
16. S. Katz. Aspect categories and classes of temporal properties. *Lecture Notes in Computer Science*, 3880:106–134, 2006.
17. G. Kiczales, J. Lamping, A. Medhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Object-Oriented Programming (ECOOP97)*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.
18. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *FTRTFT'00*, volume 1926 of *LNCS*, pages 83–93. Springer, 2000.
19. Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst.*, 21(1):46–89, 1999.
20. S. P. Miller and A. Tribble. Extending the four-variable model to bridge the system-software gap. In *Proc. 20th Digital Avionics Sys. Conf.*, Oct. 2001.
21. D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.
22. T. Rothamel, C. Heitmeyer, E. Leonard, and A. Liu. Generating optimized code from SCR specifications. In *Proceedings, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006)*, June 2006.