Form Methods Syst Des (2010) 37: 265–294
DOI 10.1007/s10703-010-0106-9

Model-based construction and verification of critical systems using composition and partial refinement

Ralph D. Jeffords · Constance L. Heitmeyer · Myla M. Archer · Elizabeth I. Leonard

Published online: 28 December 2010

Abstract This article introduces a new model-based method for incrementally constructing critical systems and illustrates its application to the development of *fault-tolerant systems*. The method relies on a special form of composition to combine software components and a set of proof rules to obtain high confidence of the correctness of the composed system. As in conventional component-based software development, two (or more) components are combined, but in contrast to many component-based approaches used in practice, which combine components consisting of code, our method combines components represented as state machine models. In the first phase of the method, a model is developed of the normal system behavior, and system properties are shown to hold in the model. In the second phase, a model of the required fault-handling behavior is developed and "or-composed" with the original system model to create a *fault-tolerant extension* which is, by construction, "fully faithful" (every execution possible in the normal system is possible in the fault-tolerant system). To model the fault-handling behavior, the set of states of the normal system model is extended through new state variables and new ranges for some existing state variables, and new fault-handling transitions are defined. Once constructed, the fault-tolerant extension is shown, using a set of property inheritance and compositional proof rules, to satisfy both the overall system properties, typically weakened, and selected fault-tolerance properties. These rules can often be used to verify the properties automatically. To provide a formal foundation for the method, formal notions of *or-composition*, *partial refinement*, *fault-tolerant extension*, and *full faithfulness* are introduced. To demonstrate and validate the method, we describe its application to a real-world, fault-tolerant avionics system.

Keywords Formal specification · Refinement · Composition · Proof rules · Fault-tolerance · Theorem proving · Masking

This research is supported by the Office of Naval Research.

R.D. Jeffords (✉) · C.L. Heitmeyer · M.M. Archer · E.I. Leonard
Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC 20375, USA
e-mail: jeffords@itd.nrl.navy.mil

1 Introduction

It is widely agreed that building a critical software system that is correct (i.e., satisfies its requirements) is challenging. Many proposed approaches at both the theoretical level and the practical level apply a divide-and-conquer approach to tackling this difficult problem. Proposed at the theoretical level are a wide range of techniques, including *composition*, as proposed by Abadi and Lamport [2], for assembling individual software components into programs; refinement, such as *stepwise refinement* as proposed by Dijkstra [15] and *refinement mappings* as proposed by Abadi and Lamport [1], for developing a concrete program from an abstract one; and formal verification using, e.g., model checking [14] or theorem proving [40], for proving that models of programs satisfy properties of interest. At the more practical level are techniques for developing software incrementally, such as component-based software development (see, e.g., Szyperski [41]), model-driven engineering [17], and aspect-oriented programming as defined by Kiczales et al. [30, 31].

The goal of this article is to describe and illustrate a sound, practical component-based method, based on composition and refinement, for developing software that satisfies its requirements. However, rather than describe a general method for component-based software development, the article describes a software engineering method for constructing a critical class of software systems—fault-tolerant systems—which (1) is supported by a formal theory and (2) can be generalized and customized to build other critical systems, such as secure systems. While in practice, component-based software engineering typically combines pieces of code, using software development environments such as Eclipse, the components in our method are state machine models. An underlying assumption of our research is that such models can be automatically or semi-automatically transformed into executable code.

Similar to others [5, 9, 10, 33], our approach to developing fault-tolerant systems is to specify the required system behavior in two phases. In the first phase, a model is developed of the *normal* (also called *ideal*) system behavior, the system behavior when no faults can occur. In the second phase, the no-faults assumption is removed, and a model of the system's required fault-tolerant behavior is developed. Our approach can be viewed as a special case of the *transformational* approach, an approach which transforms a model of normal behavior into a fault-tolerant model using, for example, some form of composition [19]. As in aspect-oriented programming [30, 31], our method weaves certain aspects, specifically, the "fault-tolerant" aspects, into the original system. In contrast to process algebra-based approaches [9], which model systems in terms of abstract transitions and synchronization, our approach describes systems in terms of concrete states defined by valuations of state variables and transitions modeled as state pairs. Moreover, in contrast to those who model fault handling transparently and focus on masking fault-tolerance under particular fault hypotheses (e.g., [9]), we focus on a different notion of masking fault-tolerance, called *partial masking fault-tolerance,*[1] which includes externally visible fault detection, fault-handling and recovery behavior.

This article, a revised and expanded version of a conference paper presented at Formal Methods 2009 [29], makes five contributions. The article (1) introduces a new component-based method for developing a special class of fault-tolerant systems, called masking fault-tolerant systems, which uses composition and property inheritance to obtain high confidence of system correctness; (2) describes *partial masking fault-tolerance* as a variant of masking fault-tolerance; (3) presents formal notions of *or-composition*, *partial refinement*, and *fully*

[1] This was called "eventual masking fault tolerance" in [29].

faithful fault-tolerant extension to provide a formal foundation for developing and reasoning about fault-tolerant systems; (4) defines a set of sound property inheritance and compositional proof rules for establishing properties of a fault-tolerant extension either automatically or interactively; and (5) illustrates the method, the new formal notions, and the application of the proof rules on the real-world avionics example of [10, 37]. These contributions make it possible to develop fault-tolerant systems whose common attributes, e.g., full faithfulness and other notions defined in Sect. 4, are "correct by construction." Although the method for constructing software proposed in this article is top-down in that the user applies a forward-engineering approach, our formal foundation also provides a solid basis for a reverse engineering approach, where a model and/or code of the fault-tolerant system already exist, and the user needs to demonstrate that the fault-tolerant system satisfies the properties of a fully faithful fault-tolerant extension. Both the method of Sect. 3 and the general theory in Sect. 4 are applicable in any software development which represents systems as state machine models, including Abstract State Machines (ASMs) [12], I/O Automata (IOA) [18], Lustre [20], Requirements State Machine Language (RSML) [22], Software Cost Reduction (SCR) [23], StateCharts [21], and Temporal Logic of Actions (TLA) [34].

The article's organization is as follows. As background, Sect. 2 reviews the Four Variable Model and defines our notions of *fault*, *failure*, and *masking fault-tolerance*. Section 3 introduces our component-based method for developing fault-tolerant systems, an extension of the approach to software development introduced in [10]. To establish a formal foundation for the method, Sect. 4, motivated in part by the theory of fault tolerance in [33] and the notion of retrenchment in [7], presents our new notions of or-composition, partial refinement, fault-tolerant extension, and full faithfulness; and our proof rules. To demonstrate and validate our approach and to show how our method supports the approach, Sect. 5 applies the method to a device controller in an avionics system [37]. Section 6 discusses verification, incremental development, and other issues regarding automation of our method. Finally, Sects. 7 and 8 discuss related work and present some conclusions.

2 Background

2.1 Four variable model

Our two-phase method for developing a fault-tolerant system is an adaptation [24, 37] of Parnas' Four Variable Model (FVM) [38]. In the FVM, the required system behavior is specified in terms of two sets of environmental variables—monitored and controlled variables—and two relations on these variables—REQ and NAT. A *monitored variable* represents an environmental quantity that influences the system behavior, while a *controlled variable* represents an environmental quantity that the system controls. 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 constraints defined by NAT. In our adaptation of the FVM, two other sets of variables, *input* and *output variables*, represent the values read from input devices (for example, sensors) from which the values of the monitored variables are estimated; and the values written to output devices computed from the values of the controlled variables [10, 37]. For example, in an avionics system, such as the Altitude Switch (ASW) described in Sect. 5, the aircraft altitude may be represented as a monitored variable; to estimate the altitude, the system software may use three altimeters, each represented by an input variable.

In specifying the normal system behavior, two assumptions of NAT are that (1) no faults can occur, and (2) the system can obtain perfect values of the monitored quantities and compute perfect values of the controlled variables. In specifying the fault-tolerant behavior, these two assumptions are removed from NAT, new monitored and controlled variables are introduced to represent faults and the required system response to faults, and NAT and REQ are extended to specify the required fault handling and recovery behavior. In addition, input and output variables are introduced to represent the values read by the software from input devices and written by the software to output devices.

2.2 Faults, failures, and masking fault-tolerance

In this article, we have simplified the widely accepted definitions of Avizienis et al. [6] to describe our notions of the terms *fault* and *failure*. Unlike [6], we do not distinguish the cause of a fault from its effect (Avizienis et al. call the latter an *error*). Instead, we use the term *fault* to refer to either a hardware problem, e.g. faulty behavior by a sensor, or a symptom of a fault, e.g., the environment fails to provide the system with expected data by a given deadline. As in [6], we use the term *failure* to refer to a system failure, i.e., the failure of the system to satisfy some requirement due to a fault. The goal of a fault-tolerant design is to eliminate system failures that occur because of faults.

In our approach, similar to that of other researchers (see, e.g., [5, 19, 33, 36]), the system's normal behavior is clearly distinguished from its fault-tolerant behavior; the system's fault-handling behavior is solely responsible for detecting and recovering from faults. The form of fault-tolerance of interest in this article is *masking fault-tolerance*. Two variants of masking fault tolerance are considered. In the first, *transparent masking*, critical properties of the system are preserved even in the presence of faults, and the effect of faults on the system behavior is invisible.[2] In the second variant, which we refer to as *partial masking*, some subset of the set of critical properties is preserved during fault handling, while other critical properties guaranteed during normal behavior may be violated. Moreover, the system always recovers from a fault in bounded time.[3] In transparent masking, the component's fault-tolerant behavior is a full refinement of its normal behavior. In partial masking, the system behavior is a "partial refinement" since refinement holds for the normal system behavior but may not hold during fault-handling. For example, in a fault-handling state, the system may not respond to certain user requests for service. This article focuses mainly on partial masking fault-tolerance and its relation to our theory of partial refinement and fault-tolerant extensions. In another form of fault-tolerance, called *fail-safe fault-tolerance*, a component responds to a fault by halting in a safe state. In practice, the design of many fault-tolerant systems combines masking fault-tolerance with fail-safe behavior.

The Altitude Switch (ASW) example in Sect. 5 illustrates both variants of masking fault-tolerance. In the design of the ASW, if at least one of the three altimeters is working, the system uses the good altimeter(s) to estimate the value of the aircraft altitude. In contrast, if all three altimeters are faulty for some specified time interval, the ASW turns on a fault indicator light, initiates fault handling, and recovers in bounded time. The fault handling behavior in the first case is an example of transparent masking because the effect of the faulty sensor is hidden. In contrast, in the second case, the fault handling behavior is an example of partial masking because the system (1) enters a fault-handling mode, i.e., no

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

[3]Kulkarni's definition of masking covers both transparent and partial masking, but with unbounded recovery time [32].

longer exhibits normal behavior, (2) turns on a fault indicator light, thus making the effects of the fault externally visible, and (3) recovers in bounded time.

3 A formal method for building fault-tolerant systems

This section introduces a new incremental, model-based method for building fault-tolerant systems. Motivated by concepts in the Four Variable Model [38] and the approach to software development described in [10], our method is applied in two phases. In the first phase, a state machine model is formulated to represent the normal system behavior and verified to satisfy critical system properties, most commonly, safety properties [3]. In the second phase, I/O devices, such as sensors and actuators, are selected, hardware and other faults which may occur are identified, and an extended state machine model representing the system's *fault-tolerant* behavior is designed. This extended, fault-tolerant model, referred to as a *fault-tolerant extension*, is then shown to satisfy (1) the critical system properties verified in the first phase, typically weakened, and (2) new properties related to fault detection, fault handling, and 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 and verify 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. A state machine model $\mathbf{ID} = (S_{ID}, \theta_{ID}, \rho_{ID})$, where S_{ID} is the set of possible states of **ID**, and θ_{ID} and ρ_{ID} are **ID**'s initial state set and transition set, respectively, is formulated to represent the "normal" behavior. The model **ID** of normal behavior omits any mention of I/O devices, or of hardware failures and other system malfunctions.

3.1.1 Specify the normal behavior in terms of NAT and REQ

The state machine model **ID** of the normal system behavior is specified in terms of (1) monitored and controlled variables and (2) the two relations—REQ and NAT—defined in Parnas' Four Variable Model. Monitored and controlled variables represent the required externally visible behavior of the system, while any additional variables are considered to be internal and hidden. As stated in Sect. 2.1, in the first phase, two assumptions of NAT are (1) no faults occur, and (2) the system can obtain perfect values of the monitored quantities and compute perfect values of the controlled variables.

3.1.2 Formulate and verify system properties

Finally, critical system properties are formulated as properties of the state machine model **ID**. These properties are often safety properties. Once formulated, the properties are verified to hold in the state machine model **ID**, using special proof rules or other proof techniques, such as model checking or theorem proving.

3.2 Specify and verify the fault-tolerant behavior

In the second phase, the NAT 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 report values of controlled quantities. Also removed is the assumption in NAT 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 **FT** (see Sect. 4) of the state machine model **ID** which adds extra behavior to handle faults and which may include new externally visible behavior, e.g., operator notification of a sensor failure. The fault-tolerant extension is represented as a state machine model $\mathbf{FT} = (S_{FT}, \theta_{FT}, \rho_{FT})$, where S_{FT} is the set of **FT**'s possible states, and θ_{FT} and ρ_{FT} are the initial state predicate and set of transitions of **FT**, respectively.

3.2.1 Select I/O devices

In the second phase, the first step is to select a set of I/O devices, such as sensors and actuators, and to document device characteristics and how the I/O devices are used to estimate values of the monitored quantities and to report the values of the controlled quantities.

3.2.2 Identify likely faults

Once the I/O devices are selected, possible faults are identified. Examples of faults include a single faulty sensor, the failure of an event to occur within some time interval, and an exception raised in the system's software environment. For practical reasons, the system is designed to handle only some possible faults.

3.2.3 Design and specify the fault-tolerant behavior

Once a set of faults is selected, a design is developed that, in response to a fault, makes the system tolerant of the fault and, in the case of some faults, reports the fault so that corrective or mitigating action may be taken. A wide range of fault-tolerance techniques are used in practice. One common technique 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, each of three (or any odd number of) sensors samples a monitored quantity's value, and a majority vote determines the value of the quantity. As noted in Sect. 2.2, some fault-tolerance techniques make faults transparent. For example, if three sensors measure the value of a monitored quantity, a majority vote may be used to estimate the value and thus mask faults as long as two sensors are functioning correctly. Transparent masking of sensor faults by a similar method is described in [10]. In contrast to [10], this article focuses on partial masking, where, in response to a fault, new fault-handling behavior is required that is externally visible. For example, in response to a fault, the system may notify an operator of the fault, who in turn may take some corrective action.

Our approach to adding fault-handling behavior to a state machine model **ID** of the system's normal behavior leads to an extension **FT** of the original model that is "faithful" in the sense that every execution possible in **ID** is possible in **FT** (with essentially the same observable behavior). To construct the fault-tolerant extension, the specification of **ID** is extended in three ways:

1. *New variables are added to the set of existing variables.* These variables may include new monitored variables, e.g., to signal that a fault occurred or a time-out expired (often a symptom of a fault). Other variables may also be added—for example, a new controlled

variable to warn a system operator that a fault has been detected, or new "history variables," such as internal variables which record the time a system has been in a given state.
2. *New values may be added to ranges of existing variables.* For example, to describe a fault-handling state, the range of some existing variable may be extended to allow an extra value `fault`.
3. *New transitions are added to the existing set of transitions.* Two classes of additional transitions are possible. One class consists of brand new transitions—for example, a transition from a state in the original system to a new fault-handling state, or a transition from a new fault state back to some normal state (i.e., recovery from the fault). The other class of new transitions arise from a "split," i.e., a transformation of an original transition in **ID** into two new transitions based on the value of a given predicate: if the predicate is true, then the transition in the fault-tolerant system corresponds to the transition in the original system; if false, then the transition is to a new fault-handling state.

Once the three extensions above have been specified, the user may "compose" them with the original specification of the state machine model **ID** to obtain a specification of the extended state machine model **FT**. First, the new variables are inserted into the set of original variables to produce a new set of state variables. Next, the type sets of variables with new values are modified to include the new values. These two extensions lead to the set S_{FT} of possible states in **FT**. Finally, the new transitions are inserted into the set of transitions of the original state machine model to form a new set ρ_{FT} of transitions. The state set S_{FT} can be naturally partitioned into N, the set of normal operating states augmented with the new variables, and F, the set of fault-handling states. The faithfulness to **ID** of the extension **FT** follows because the extensions to the specification of **FT** satisfy the "non-interference" notion of Arora and Kulkarni [5], i.e., do not interfere with the original system behavior described by **ID**.

3.2.4 Formulate and verify properties of the fault-tolerant specification

In this step, the critical properties verified to hold for the normal system behavior **ID** must be shown to hold for the fault-tolerant behavior **FT**. In some cases, properties of the normal system will not hold throughout the fault-tolerant system but may only remain true during the normal system behavior. A new notion of partial refinement, defined in Sect. 4.1, describes the conditions which must be established for the fault-tolerant system to partially inherit properties of the normal system. Also in this step, 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 often be established as invariants with the aid of *property inheritance rules* and *compositional proof rules*. Examples of these rules appear in Sect. 4.3.

4 Formal foundations

This section presents formal definitions, theoretical results, and formal techniques that support the method described in Sect. 3 for developing provably correct fault-tolerant systems. The most important concepts and results include our notions of *partial refinement*, *or-composition*, and *fault-tolerant extension*, together with two proof methods for establishing properties of a fault-tolerant extension based on properties of the normal (fault-free) system behavior it extends. We identify one subclass of fault-tolerant extensions as *simple*,

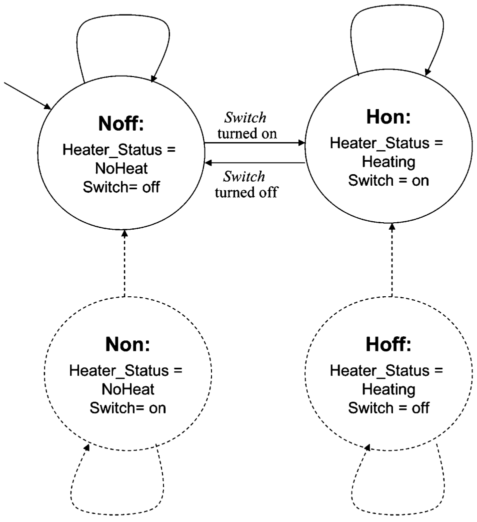

Fig. 1 State diagram for the Simple Heating System (**SHS**). Unreachable states and transitions are indicated by *dotted lines*

and establish useful properties for extensions in this subclass. We also define a *faithful* fault-tolerant extension (already discussed informally). Our first proof method describes property inheritance under partial refinement and is based on Theorems 2 and 3 (see Sects. 4.3 and 4.1). Our second proof method is based on compositional proof rules for invariants, two of which are shown in Fig. 4 of Sect. 4.3.

The section begins with general notions concerning state machines, formally defines the notion of *or-composition*, and relates or-composition to the method presented in Sect. 3.2 for adding fault-tolerant behavior to normal system behavior. It also introduces fault-tolerance concepts, and discusses additional concepts and results that arise from added assumptions about state machines—first, that states are determined by the values of a set of state variables, and second, that state transitions are triggered by a change in value of some input variable. Each concept or result presented is introduced at the highest level of generality possible. To illustrate the definitions, results, and techniques of this section, we use the running example, illustrated in Fig. 1, of a simple heating system controlled by an on-off switch:

Example 1 (Simple Heating System (**SHS**)) The behavior of the **SHS** is described in terms of a set of variables whose values determine the current state of the system:

- Heater_Status, with possible values *NoHeat* and *Heating*, and
- Switch, with possible values *on* and *off*.

The goal in this example is to extend the behavior of the **SHS** to obtain **SHS-FT**, an enhanced version of the Simple Heating System which is fault tolerant, and to relate the properties of **SHS-FT** to the properties of the **SHS**. In Fig. 1, only the states **Noff** and **Hon** are reachable; the states **Non** and **Hoff** are unreachable.

4.1 General definitions

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 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. An *execution sequence* (*execution*) of **A** is a sequence of states $s_0, s_1, \ldots, s_n$ $(s_0, s_1, \ldots, s_n, \ldots)$ in S_A such that $(s_{i-1}, s_i) \in \rho_A$ for every i with $1 \le i \le n$ $(1 \le i)$. A state $s_A \in S_A$ is *reachable* if there is an execution sequence $s_0, s_1, \ldots, s_n$ of **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. □

In the **SHS** (see Fig. 1),

- The set of states $S_{SHS} = \{$**Noff**, **Hon**, **Non**, **Hoff**$\}$
- The set of initial states $\Theta_{SHS} = \{$**Noff**$\}$
- The set of transitions $\rho_{SHS} = \{$(**Noff**, **Hon**), (**Hon**, **Noff**), (**Noff**, **Noff**), (**Hon**, **Hon**), (**Non**, **Noff**), (**Hoff**, **Hon**), (**Non**, **Non**), (**Hoff**, **Hoff**)$\}$

Only the states **Noff** and **Hon** are reachable. Four of the transitions in ρ_{SHS} are stutter steps, and only the first four transitions listed are reachable.

Definition 2 (One-state and two-state predicates/invariants) Let $\mathbf{A} = (S_A, \Theta_A, \rho_A)$ 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. □

The **SHS** has state invariants:

$$\begin{aligned} \mathsf{Heater\_status} = \mathit{NoHeat} &\quad\Leftrightarrow\quad \mathsf{Switch} = \mathit{off} \\ \mathsf{Heater\_status} = \mathit{Heating} &\quad\Leftrightarrow\quad \mathsf{Switch} = \mathit{on} \end{aligned}$$

Its transition invariants include:

$$\mathsf{Switch} = \mathit{off} \wedge \mathsf{Switch}' = \mathit{on} \quad\Rightarrow\quad \mathsf{Heater\_status}' = \mathit{Heating}$$

The above transition invariant states that in the result state of any transition in which the value of Switch changes from *off* to *on*, the value of Heater_status is *Heating*.

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 $\mathbf{A} = (S_A, \Theta_A, \rho_A)$ and $\mathbf{C} = (S_C, \Theta_C, \rho_C)$ 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 $\rho_C(s_C, s'_C)$. □

Definition 4 (Partial refinement) Let $\mathbf{A} = (S_A, \Theta_A, \rho_A)$ and $\mathbf{C} = (S_C, \Theta_C, \rho_C)$ be two state machines and $\alpha : S_C \overset{\circ}{\to} S_A$ 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

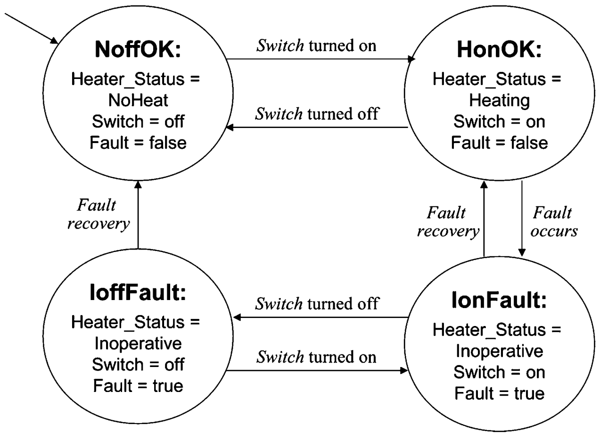

Fig. 2 State diagram for fault-tolerant version of Simple Heating System (**SHS-FT**), showing only the reachable states and non-stutter steps. By assumption, a fault in the heater cannot arise when the switch is off

(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 partial refinement mapping α). □

The state variables of the fault-tolerant state machine **SHS-FT**, whose reachable part is shown in Fig. 2, are:

- Heater_Status, with possible values *NoHeat*, *Heating* and *Inoperative*;
- Switch, with possible values *on* and *off*; and
- Fault, with possible values *false* and *true*.

The full set S_{SHS-FT} of states of **SHS-FT** has twelve elements. The set of reachable states of **SHS-FT** is {**NoffOK**, **HonOK**, **IonFault**, **IoffFault**}, and the initial state is **NoffOK**. Figure 2 shows the seven reachable transitions that are not stutter steps. The partial mapping

$$\alpha_{SHS} : S_{SHS-FT} \overset{\circ}{\rightarrow} S_{SHS}$$

defined by α_{SHS}(**HonOK**) = **Hon** and α_{SHS}(**NoffOK**) = **Noff**, with α_{SHS} otherwise undefined, is easily seen to be a partial refinement mapping.

4.2 Or-composition

In the theory of automata (that is, state machines), most notions of composition are naturally described in terms of how two (or more) state machines can be combined into one—for example, sequential composition in process algebras, and various forms of parallel composition with synchronization on actions or through message passing. The motivation for performing an *or-composition* is to create a state machine that allows two (or more) kinds of behavior. Any particular behavior of the resulting machine is *either* of one kind *or* another; hence the name or-composition. Thus, although or-composition also can be described in terms of state machines working together (in this case, through shared variables), it is most naturally thought of in terms of "behaviors" as represented by sets of possible transitions (on corresponding sets of states). Formally:

Definition 5 (Or-composition) Let $\mathbf{A} = (S_A, \Theta_A, \rho_A)$ and $\mathbf{B} = (S_B, \Theta_B, \rho_B)$ be two state machines, where any of $S_A \cap S_B$, $\Theta_A \cap \Theta_B$, and $\rho_A \cap \rho_B$ may be nonempty. Then the *or-composition* of **A** and **B** is the state machine

$$\mathbf{or}(\mathbf{A}, \mathbf{B}) \stackrel{\triangle}{=} (S_A \cup S_B, \Theta_A \cup \Theta_B, \rho_A \cup \rho_B). \qquad \square$$

Given a state machine **A** with an associated set of transitions which describe the "old behavior", suppose one wishes to produce a new state machine **C** with all the capabilities of **A** plus new capabilities—such as exception handling or fault tolerance—captured by additional "new" behavior defined in terms of "new" possible states and transitions. Then, one can use or-composition to obtain **C** as $\mathbf{or}(\mathbf{A}, \mathbf{B})$, where **B** is the state machine whose set of initial states is empty, whose transitions are the new transitions, and whose states are the new states plus all states of **A** that are either the source or target point of some new transition.

4.3 Concepts for fault tolerance

Our approach for including fault tolerance in the software development method described in Sect. 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 been detected, and
- there is a map $\pi : N \to S_{ID}$ and a two-state predicate $O \subseteq N \times N$ such that $s \in N \Rightarrow O(s, s)$, and $s_1, s_2 \in S_{FT} \wedge O(s_1, s_2) \wedge \rho_{FT}(s_1, s_2) \Rightarrow \rho_{ID}(\pi(s_1), \pi(s_2))$ and $\pi(\Theta_{FT}) \subseteq \Theta_{ID}$.

The map π and predicate O are called, respectively, the *normal state map* and *normal transition predicate* for **FT**. **FT** is a *faithful* fault-tolerant extension of **ID** if every execution in **ID** is the image under π of an execution in **FT**. **FT** is a *fully faithful* fault-tolerant extension of **ID** if for every state s_0 of **ID** and every state $\bar{s}_0$ in $\pi^{-1}(s_0)$, every execution in **ID** from s_0 is the image under π of an execution in **FT** starting from $\bar{s}_0$. $\square$

SHS-FT is easily seen to be a fault-tolerant extension of **SHS**, with

$N = \{s \in S_{SHS-FT} : \mathsf{Heater\_Status}(s) = \mathit{Heating} \vee \mathsf{Heater\_Status}(s) = \mathit{NoHeat}\}$;
$F = \{s \in S_{SHS-FT} : \mathsf{Heater\_Status}(s) = \mathit{Inoperative}\}$;
$O = N \times N$; and
$\forall s \in N, \pi(s) = \hat{s} \in S_{SHS}$,
where $\mathsf{Heater\_Status}(\hat{s}) = \mathsf{Heater\_Status}(s) \wedge \mathsf{Switch}(\hat{s}) = \mathsf{Switch}(s)$.
(Equivalently, π is α_{SHS} restricted to N.)

Noting that whenever $\bar{s}_0$ is a state of **SHS-FT** that maps under π to a state s_0 of **SHS**, every **SHS** transition from s_0 is the image under π of a transition in **SHS-FT** from $\bar{s}_0$, it is not hard to see that **SHS-FT** is a fully faithful extension of **SHS**.

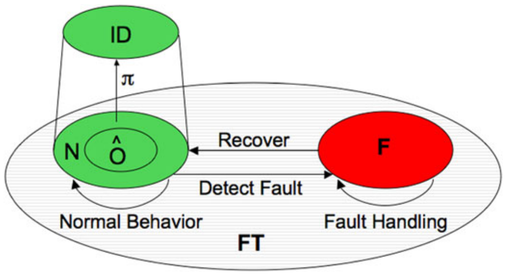

Fig. 3 Transitions in the fault-tolerant system **FT**, and relating **FT** to **ID**

Remark 1 When **FT** is a fault-tolerant extension of **ID**, the normal state map π is a partial refinement mapping from the state machine

$$\mathbf{FTO} \triangleq (S_{FT}, \Theta_{FT}, O \cap \rho_{FT})$$

to **ID**.

Clearly, there is a strong connection between the notions of fault-tolerant extension and or-composition. In fact, there are many ways to represent **FT** as an or-composition; Corollary 1 shows one straightforward representation based on the normal transition predicate O.

Corollary 1 *If* **FT** *is a fault-tolerant extension of* **ID** *with normal states* N *and fault-handling states* F, *then* **FT** *is the or-composition of the state machines*:

$$\mathbf{FTO\text{-}N} \triangleq (N, \Theta_{FT}, O \cap \rho_{FT}) \quad \textit{and} \quad \mathbf{FT\bar{O}} \triangleq (S_{FT}, \Theta_{FT}, \neg O \cap \rho_{FT}).$$

Remark 2 **FTO-N**, which is **FTO** with a restricted state set, is a refinement of **ID** with refinement mapping π.

Remark 3 The state machines **FTO** and $\mathbf{FT\bar{O}}$ may be viewed as an *or-decomposition* of **FT** with respect to O.

Figure 3 illustrates the structure of a fault-tolerant extension **FT** of **ID** 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 in Fig. 3),
3. transitions from N to F (Fault Detection),
4. transitions from F to F (Fault Handling), and
5. transitions from F to N (Fault Recovery).

Definition 7 Let the state machine **FT**, with normal states N, fault-handling states F, and normal state map π, be a fault-tolerant extension of the state machine **ID**. Then **FT** is *simple* if all its transitions from N to N map under π to transitions in **ID**, that is, if it has no class 2 transitions. □

Since every transition between normal states of **SHS-FT** maps under π to a transition of **SHS**, **SHS-FT** is a simple fault-tolerant extension of **FT**.

It is not difficult to prove the following:

Theorem 1 *For any fault-tolerant extension* **FT** *of* **ID**, *the following are equivalent*:

1. **FT** *is a simple fault-tolerant extension of* **ID**,
2. *the normal state map π is a partial refinement mapping from* **FT** *to* **ID**, *and*
3. *the normal transition predicate can be chosen to be*

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

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, provided O satisfies a certain condition. In particular, given O, let $\hat{O}$ be the one-state predicate for **FT** defined by:

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

(It is easily seen, as indicated in Fig. 3, 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, provided O includes every transition from a start state of **FT** so that $\Theta_{FT} \subseteq \hat{O}$, 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 reachable state in N maps under π to a reachable state in* **ID**, *and every reachable transition from a state in $\hat{O} \subseteq N$ maps under π to a reachable transition in* **ID**.

Before stating Theorems 2 and 3 about property inheritance, we need one further definition:

Definition 8 (Inductive property) Let $\mathbf{A} = (S_A, \Theta_A, \rho_A)$ be a state machine and P a one-state property for **A**. Then P is *inductive* (in **A**) if $\forall s_1, s_2 \in S_A,\ P(s_1) \wedge \rho_A(s_1, s_2) \Rightarrow P(s_2)$. □

Theorem 2 (One-state property inheritance) *Let* **FT** *be a fault-tolerant extension of* **ID** *with normal states N, normal state map π, and normal transition predicate O; P be a one-state invariant of* **ID**; *and $\Phi(P) \triangleq P \circ \pi$. Then $wP \triangleq (N \Rightarrow \Phi(P))$ is a state invariant of* **FT** *if either*

1. *every reentry or exceptional target state of* **FT** *maps under π to a reachable state in* **ID**, *or*
2. *every reentry or exceptional target state of* **FT** *maps under π to a state in* **ID** *that satisfies P and property P is inductive in* **ID**.

Theorem 3 (Two-state property inheritance) *Let* **FT** *be a fault-tolerant extension of* **ID** *with normal states N, normal state map π, and normal transition predicate O; P be a transition invariant of* **ID**; *and $\Phi(P) \triangleq P \circ (\pi \times \pi)$. Then $wP \triangleq (O \Rightarrow \Phi(P))$ is a transition invariant of* **FT** *if either*:

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

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

Fig. 4 Compositional proof rules for state and transition invariants of **FT**

1. *every reentry or exceptional target state of* **FT** *maps under* π *to a reachable state in* **ID**, *or*
2. $\rho_{ID} \Rightarrow P$.

As shown in Sect. 5, the fault-tolerant ASW behavior is a fault-tolerant extension of the normal ASW behavior with natural definitions for N and F (see Sect. 5.3.2.3), and π and O (see Sect. 4.4), such that all transitions from N to N are of class 1. Further, we have proven 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, the first cases in Theorems 2 and 3 can be used to deduce properties of **FT** from properties of **ID**.

In general, however, to supplement Theorems 2 and 3, a method is needed for establishing properties of **FT** in the case when it is difficult or impossible to establish either of the sufficient conditions of Theorems 2 and 3. For this purpose, we provide *compositional proof rules* analogous to those in our earlier work [28]. We first define what it means for a predicate to *respect* a mapping:

Definition 9 Let π 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 4 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 a fault-tolerant extension.

4.4 Fault tolerance concepts in terms of state variables

When we restrict our attention to state machine models whose state spaces are defined solely in terms of state variables (with associated types), i.e., when the possible states of a state machine are exactly the (type correct) assignments of values to the state variables, the concepts in Sects. 4.2 and 4.3 can be interpreted explicitly.

First, suppose we wish to construct a fault-tolerant extension **FT** of a state machine **ID**. In the restricted context of state machines whose states are defined in terms of state variables, there are two natural ways to extend the original set of states of **ID** (the "normal" states).

The first way is to add new state variables, thus obtaining in a natural way a set N and a map π that projects N onto S_{ID}. The second way is to extend the range of values of existing variables, a natural way to add a set F of new, "abnormal" states. One can represent **FT** as an or-composition of state machines **FTO-N** and **FT$\bar{\mathbf{O}}$** with state spaces N and $N \cup F$ (as with the fault-tolerant extension **FT** of **ID** in Corollary 1), once one identifies the transitions of each of **FTO-N** and **FT$\bar{\mathbf{O}}$** (and also the initial states in N shared by both). It is natural to define the transitions in **FTO-N** as those agreeing with some transition of **ID** on all old state variables and affecting none of the new state variables; these naturally map under π to transitions in **ID**. Defining the transitions of **FTO-N** to be of this form ensures that **FT** will be a fully faithful extension of **ID**. The transitions in **FT$\bar{\mathbf{O}}$** must cover all transitions in **FT** of classes 2–5.

Now, consider the context of *input-driven* state machine models in which the state variables include a set of input variables and every state transition is triggered by an "input event" (a change in the value of an input variable). This notion is not uncommon: many specification languages, including StateCharts [21], RSML [22], Lustre [20], and SCR [23], support the specification of input-driven models. For input-driven state machine models, it is possible to say more about the new transitions. For example, while the transitions in **FTO-N** (which go from N to N) may be triggered by input events of **ID**, any new transition from a state in N to a state in F requires a new triggering input event, requiring in turn the extension of the range of some input variable of **ID** or the introduction of a new input variable.

Hence, when states are defined by the values of state variables and transitions are triggered by input events, constructing a fault-tolerant system model **FT** from a normal system model **ID** using or-composition is naturally done by adding new variables, new values to types of existing variables, and new transitions to describe the triggering and subsequent handling of faults. We refer to the original variables as *normal* variables and the added variables as *fault-tolerance* variables; for any normal variable, we 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 $\pi : N \to S_{ID}$ is then chosen to be the projection map with respect to the normal variables.

More can also be said about specific predicates and predicates in general when states are determined by the values assigned to state variables. First, although Definition 2 represents predicates abstractly as sets, most predicates of interest can be represented syntactically as relations among state variables and constants. Further, on a syntactic level, the maps Φ defined in Theorems 2 and 3 are both the identity map. Finally, since, by Theorem 1, the predicate N can be expressed simply as an assertion that no normal variable has an extended value, when states are defined by the values assigned to state variables, computing O automatically is possible for any **FT** defined as a simple fault-tolerant extension of **ID**.[4]

5 Example: The Altitude Switch (ASW)

This section shows how the method and concepts presented in Sects. 3 and 4 can be applied to a practical system, an Altitude Switch (ASW) controller in an avionics system [37]. Section 5.1 briefly summarizes the ASW's normal and its fault-tolerant behavior. The goal of the ASW example is to demonstrate how one can use our method to specify a state machine

[4]In the context of SCR, 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**.

model **ID** of normal behavior and a separate state machine model **FT** that extends **ID** with fault-tolerant behavior such that **FT** is a simple fault-tolerant extension of **ID**.

In this section, both models of the ASW's required behavior, **ID** and **FT**, are specified in the tabular SCR notation. To make the SCR specifications understandable, Sect. 5.2 briefly reviews the SCR model, its constructs, and its tabular notation. Section 5.3 illustrates how we applied the method of Sect. 3, step by step, to the ASW, and shows how the theoretical results in Sect. 4, in particular, the property inheritance and compositional proof rules, can be used to prove properties of **FT**. Although many ways of specifying state machine models of systems can be used in conjunction with the method of Sect. 3, Sect. 5.4 describes how certain features of SCR have important advantages when applying our method.

5.1 Behavior of the ASW

The primary function of the ASW, i.e., its normal behavior, is to power on a generic Device of Interest (DOI) when an aircraft descends below a threshold altitude. In the ASW, the pilot can set an inhibitor button to prevent the powering on of the DOI or press a reset button to reinitialize the ASW. By design, the ASW system must tolerate a number of faults. In response to each fault, the ASW enters a fault-handling mode and turns on a Fault Indicator Lamp. In the fault-handling mode, the ASW's response to certain inputs is different from its response in one of its normal operational modes. After a specified non-zero time in the fault-handling mode, the system recovers, i.e., makes a transition back to a normal operational mode. Thus, the ASW's fault-handling behavior is an example of partial masking.

5.2 Overview of SCR

Any SCR state machine model $A = (S_A, \theta_A, \rho_A)$ is a special case of the Four Variable Model (FVM). In SCR as in the FVM, monitored and controlled variables represent the externally visible behavior of the system. SCR also has additional "hidden" variables—namely, mode classes, terms, input variables, and output variables.[5] Mode classes and terms allow concise representations of the relations NAT and REQ. Two other important SCR constructs are conditions and events; a *condition* is a predicate on a single state, while an *event* is a two-state predicate on an old state and new state indicating a change in some variable value. If condition c's values in the old and new states are denoted c and c', then the semantics of the *basic event* @T(c) is defined by $\neg c \wedge c'$, and the semantics of @F(c) by $c \wedge \neg c'$. A *conditioned event*, denoted @T(c) WHEN d, adds a qualifying condition d to an event and has the semantics $\neg c \wedge c' \wedge d$. A *monitored event* represents a change in value of a monitored variable. In SCR, each transition in ρ_A is uniquely determined by a state s in S_A and a monitored event permitted in s, and an execution, which starts in some initial state in θ_A, is driven by a nondeterministic sequence of monitored events. Each new state in the execution is defined by the new value of the monitored variable that changed, no change in the values of other monitored variables,[6] and updates to the remaining state variables deterministically determined by the SCR tables. This process is synchronous: the system completely processes one monitored event before processing the next monitored event.

[5]In the ASW specification presented in this article, the input and output variables have been omitted.

[6]SCR's One Input Assumption allows a change in only a single monitored variable.

Table 1 Condition table defining `cWakeUpDOI`

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

In specifying the required behavior of fault-tolerant systems, mode classes are especially useful because they partition the system state space in an intuitive way. Each mode corresponds to a "mode of operation" of the system; the system behaves differently in one mode than in another. Thus, for example, in flight software, the software behaves differently when an engine has failed than when all its engines are operating normally. As described later in this section, a mode class has a special role in an SCR specification of a fault-tolerant system—it is a concise means of partitioning the system into N, the set of normal states, and F, the set of fault-handling states. For more on the SCR modeling language and the tools available for representing, validating, and verifying an SCR model, see [23, 26].

5.3 Applying the method to the ASW

5.3.1 Specify and verify the normal system behavior

This section presents an SCR specification of the normal behavior **ID** of the ASW expressed in terms of the relations NAT and REQ. It also presents a set of critical system properties expected to hold in **ID** and describes how they are verified.

5.3.1.1 Specify the normal behavior in terms of NAT and REQ In SCR, the normal ASW behavior **ID** is specified in terms of (1) controlled and monitored variables, (2) environmental assumptions, and (3) the required relation between the monitored and controlled variables. The relation NAT is defined by (1) and (2) and the relation REQ by (3). Specifying the system modes, which partition the possible system states (each mode represents a subset of system states), and how the modes change in response to monitored events is useful in specifying REQ.

The ASW has a single controlled variable `cWakeUpDOI`, a boolean, initially false, which signals the DOI to power on, and five 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; and (5) `mReset`, true when the pilot has pressed the reset button. 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).

In the SCR specification of **ID**, the normal system behavior, the relation REQ is defined by two tables, Tables 1 and 2. Table 1, a condition table, defines the value of **ID**'s single controlled variable `cWakeUpDOI` as a function of the mode class `mcStatus`. The table states that if `mcStatus` = `awaitDOIon`, then `cWakeUpDOI` is *True*; otherwise, it is *False*.

Table 2, an event table, defines **ID**'s mode transitions. Figure 5, a different but equivalent representation of **ID**'s mode transitions, uses a finite state diagram to describe the transitions. In row 2 of Table 2 and in Fig. 5, the event "`@T(mReset)`" indicates that the pilot has switched the Reset button to `on`. Both Table 2 and Fig. 5 show that when this event occurs, the ASW makes a transition from mode `standby` to mode `init`. In row 3 of

Table 2 SCR table showing the mode transitions in the **ASW** normal system behavior

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`

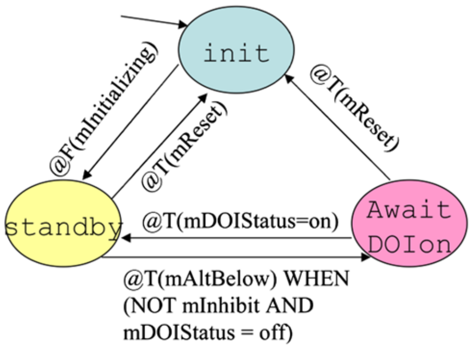

Fig. 5 Finite state diagram of mode transitions in the **ASW** normal system behavior

Table 2 and in Fig. 5, the conditioned event "`@T(mAltBelow) WHEN (NOT mInhibit AND mDOIStatus = off)`" means that the aircraft's altitude dropped below some threshold when powering on the DOI was not inhibited and the DOI was off. As shown in both Table 2 and Fig. 5, when this event occurs, the ASW makes a transition from `standby` to `awaitDOIon`.

5.3.1.2 Formulate and verify system properties Table 3 defines three required properties, P_1, P_2, and P_3, of the ASW's normal behavior.[7] Property P_1, a transition invariant, states that pressing the reset button always causes the system to return to the initial mode. Property P_2, another transition invariant, specifies the event and conditions that must hold to wake up the DOI. Property P_3, a state invariant, states that when the system is in mode `awaitDOIon`, the DOI is powered off. Applying the property checker Salsa [11] easily verifies that the specification of the ASW's normal behavior satisfies all three properties.

5.3.2 Specify and verify the fault-tolerant behavior

This section describes how the normal behavior **ID** of the ASW can be extended to handle faults, thus producing the fault-tolerant extension **FT**. 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 is designed. Finally, new ASW

[7] In this article, the primed variable `mcStatus`′ 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.

Table 3 System properties of the ASW's normal behavior **ID**

Name	System	Formal Statement
P_1	**ID**	$@T(\mathtt{mReset}) \Rightarrow \mathtt{mcStatus}' = \mathtt{init}$
P_2	**ID**	$\mathtt{mcStatus} = \mathtt{standby} \wedge @T(\mathtt{mAltBelow}) \wedge \neg\mathtt{mInhibit}$ $\wedge\, \mathtt{mDOIStatus} = \mathtt{off} \Rightarrow \mathtt{cWakeUpDOI}'$
P_3	**ID**	$\mathtt{mcStatus} = \mathtt{awaitDOIon} \Rightarrow \mathtt{mDOIStatus} = \mathtt{off}$

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.

5.3.2.1 Select I/O devices As described above, 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 [10].

5.3.2.2 Identify likely faults In addition to its transparent handling of up to two altimeter failures, the ASW is designed to tolerate three additional 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. These three additional faults are all handled using partial masking—the system enters a fault handling state but eventually recovers to normal behavior. To notify the pilot that a fault has occurred, the ASW turns on a Fault Indicator Lamp.

5.3.2.3 Design and specify the fault-tolerant behavior To describe the ASW's fault-tolerant behavior, the state machine **ID**, whose SCR specification is presented in Sect. 5.3.1.1, is extended with fault-handling behavior to produce a new state machine **FT**, the fault-tolerant extension of **ID**. Fault-handling in the ASW consists of two parts. First, when any of the three faults handled with partial masking (see Sect. 5.3.2.2) is detected, the ASW makes a transition to a new `fault` mode. Second, when the pilot hits the reset switch, the system recovers (i.e., returns to its normal behavior). Adding fault-handling behavior to the specification of normal behavior requires the three extensions described in Sect. 3.2.3:

1. *Add new variables.* Two new monitored variables are added to represent the detection of faults: `mAltimeterFail`, a boolean signaling the failure of all three altimeters,[8] and `mTime`, the time measured by the system clock. Also added are a new controlled variable, `cFaultIndicator`, which turns on a Fault Indicator Lamp to warn the pilot of a fault, and several "history variables," which record the time since an event of interest occurred.
2. *Add new values to ranges of existing variables.* To indicate that the ASW has detected a fault, a new mode `fault` is added to the mode class.
3. *Add new transitions to the existing set of transitions.* The definition of the existing controlled variable `cWakeUpDOI` is extended to indicate that when the system makes a transition to the `fault` mode, the value of `cWakeUpDOI` is *False*. In addition, several new transitions are added to the original set of mode transitions: new transitions from

[8]Because this article focuses on the partial masking fault-tolerance aspects of the ASW, it omits the details of how the value of `mAltimeterFail` is computed from the values of input variables which represent the altimeters. For these details, see [10].

Table 4 Assumptions in the specification of the fault-tolerant extension **FT**

Name	System	Formal Statement
A_1	FT	$(\mathtt{mTime}' - \mathtt{mTime}) \in \{0, 1\}$
A_2	FT	$\mathtt{DUR(cFaultIndicator} = \mathtt{on)} \leq \mathtt{FaultDur}$

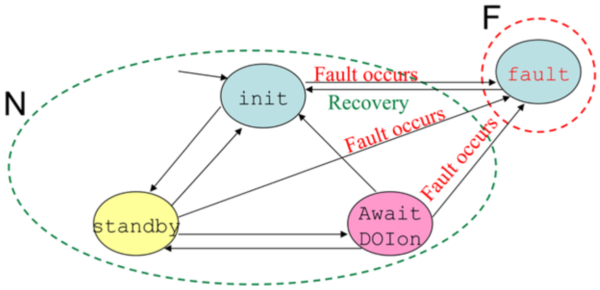

Fig. 6 Finite state diagram showing new mode transitions in the **ASW** fault-tolerant behavior

Table 5 New table defining `cFaultIndicator`

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

each normal operating mode to the new `fault` mode, and a single new transition from the `fault` mode to a normal operating mode (i.e., recovery).

Adding new variables To further define the values of the new variables, the following extensions are needed. First, `mTime` is defined as an integer with initial value zero, and an assumption A_1 is added (see Table 4) stating that time never decreases, and if time increases, it increases by one time unit. Second, to indicate when the Fault Indicator Lamp is turned on, a new table, Table 5, is defined stating that the lamp, represented by controlled variable `cFaultIndicator`, is on when the system is in `fault` mode and off otherwise. To indicate recovery in bounded time, an assumption A_2 (see Table 4) states that the lamp is on for at most `FaultDur` time units. This means that when the pilot sees that the Fault Indicator Lamp is on, he or she always responds in `FaultDur` time units by pressing the reset switch. This returns the ASW to its initialization mode, thus turning the lamp off (as shown in Table 5). Assumption A_2 is defined in terms of SCR's `DUR` operator, which used to define timing constraints. Informally, if c is a state predicate and k a positive integer, the predicate $\mathtt{DUR}(c) = k$ holds in state i of the specification if in state i predicate c is true and has been true for exactly k time units. Thus, the expression "$\mathtt{DUR(cFaultIndicator} = \mathtt{on)} \leq \mathtt{FaultDur}$" in Table 4 states that A_2 is true when the lamp has been on for at most `FaultDur` time units, and false otherwise. Associated with this expression is a (hidden) history variable, which records the time since the lamp was turned on.[9]

Adding new values to ranges In adding the new mode `fault` to the set of modes, we observe that the normal states of the fault-tolerant ASW may be described by $N: \mathtt{mcStatus} \neq \mathtt{fault}$ and the fault-handling states by $F: \mathtt{mcStatus} = \mathtt{fault}$. In Fig. 6,

[9]History variables derived from expressions containing the DUR operator are normally hidden from the SCR toolset user.

Table 6 Revised table for `cWakeUpDOI`

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

Table 7 New SCR table showing the mode transitions in the **ASW** fault-tolerant behavior

	Row No.	Old Mode	Event	New Mode
	1	init	@F(mInitializing)	standby
	2	standby	@T(mReset)	init
†	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
	4	awaitDOIon	@T(mDOIStatus = on)	standby
	5	awaitDOIon	@T(mReset)	init
→	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

the set S_{FT} of system states is naturally partitioned into N and F, where N contains the states in the original system behavior and F contains the fault-handling states.

Adding new transitions To complete the SCR specification of fault-handling in the ASW, new mode transitions are added to the existing set of mode transitions. In the SCR specification of **FT**, new transitions are added by extending two tables in the original specification of ASW, Tables 1 and 2. The first new table, Table 6, states that when the ASW enters the `fault` mode, the value of `cWakeUpDOI` is *False*. The second new table, Table 7, describes the extended set of mode transitions.

Figure 6 shows the four new mode transitions in **FT**: Three transitions, each triggered by a fault, from a normal operating mode to the new `fault` mode; and a fourth transition, triggered by a recovery action, from `fault` to a normal operating mode, `init`. To capture the four new transitions, a new mode transition table, Table 7, is created which extends Table 2. Table 7 contains rows 1, 2, 4, and 5 of Table 2 and three new rows 6, 7, and 8; and replaces row 3 of Table 2 with rows 3a and 3b. The third column of rows 3b, 6, and 7, each marked by a simple arrow, shows the three events which trigger ASW entry into the `fault` mode. Row 3b states that the ASW goes from `standby` to `fault` when the altitude falls below the threshold and other conditions are true (e.g., all altimeters have failed). The events in rows 6 and 7 of Table 7 are defined using SCR's `DUR` operator. Row 6 states that when the ASW has been in the `init` mode for more than `InitDur` time units, it enters the `fault` mode. Similarly, row 7 states that if the ASW is in `awaitDOIon` and either the time in this mode, or the time since all three altimeters failed, exceeds `FaultDur` time units, the ASW enters the `fault` mode. Row 8 of Table 7, marked by a squiggly arrow, describes recovery: The ASW makes a transition from `fault` to `init` when the pilot pushes the reset button.

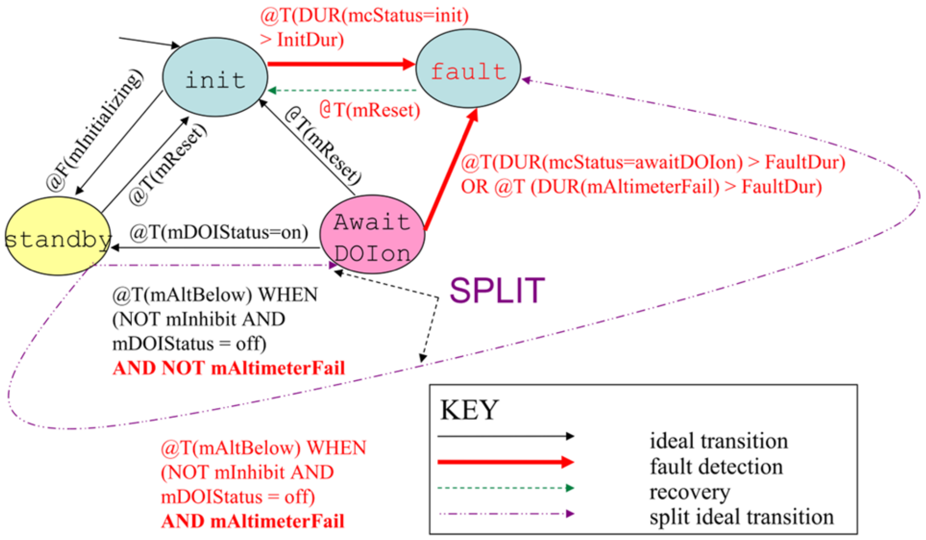
@T(DUR(mcStatus=init) > InitDur)
init
fault
@T(mReset)
@F(mInitializing)
@T(mReset)
@T(mReset)
@T(DUR(mcStatus=awaitDOIon) > FaultDur) OR @T (DUR(mAltimeterFail) > FaultDur)
standby
@T(mDOIStatus=on)
Await DOIon
SPLIT
@T(mAltBelow) WHEN (NOT mInhibit AND mDOIStatus = off) AND NOT mAltimeterFail
@T(mAltBelow) WHEN (NOT mInhibit AND mDOIStatus = off) AND mAltimeterFail
KEY
ideal transition
fault detection
recovery
split ideal transition

Fig. 7 Types of transitions in the fault-tolerant **ASW** system

Two ways to add new transitions The fault-tolerant extension **FT** of the ASW adds new transitions to the original set of transitions in the two ways described in Sect. 3.2.3. First, **FT** adds a set of brand new transitions. In Fig. 7, the transitions from `awaitDOI` to `fault`, from `init` to `fault`, and from `fault` to `init` are all brand new. **FT** also adds transitions that arise from a "split" transition. A split transforms the original transition from `standby` to `awaitDOI` (defined in row 3 of Table 2) into two new transitions, both dependent on the new monitored variable `mAltimeterFail`. As shown in Fig. 7 (and row 3a of Table 7, marked by a '†'), if the altitude drops below the threshold when the DOI is not inhibited and turned off, and all three altimeters have not failed, then the fault-tolerant behavior **FT** is the same as the normal behavior **ID**—the ASW makes the transition from `standby` to `awaitDOI`. As shown in Fig. 7 (and row 3b of Table 7), if instead the same event occurs when all three altimeters have failed, the fault-tolerant extension **FT** makes a transition from `standby` to `fault`.

5.3.2.4 Formulate and verify properties of the fault-tolerant specification Table 8 defines several desired properties of the fault-tolerant extension of the ASW. Properties P_1, P_2, and P_3, which have been shown to hold in the normal ASW behavior **ID**, are also candidate properties of the ASW's fault-tolerant extension **FT**. In addition, two new properties H_1, and H_2,[10] are desired properties of **FT**. Property H_1 states that if the ASW is currently in `fault` mode, then in the next state it either stays in `fault` mode or transitions to `init` mode. Satisfaction of this property guarantees that once the ASW enters `fault` mode, it stays there until recovery (i.e., entry into `init` mode). Property H_2 states that if the ASW is in `standby` mode and all altimeters have failed, then it is not possible for the ASW to enter `awaitDOIon` mode in the next state. Satisfaction of this property ensures that when the altimeters have all failed, the ASW does not power on the DOI. Of these properties,

[10]Properties H_1 and H_2 were formulated by Ebnenasir [16].

Table 8 Properties of the ASW's fault-tolerant extension **FT**

Name	System	Formal Statement
P_1	**FT**	$@T(\mathtt{mReset}) \Rightarrow \mathtt{mcStatus}' = \mathtt{init}$
P_2	–	$\mathtt{mcStatus} = \mathtt{standby} \wedge @T(\mathtt{mAltBelow}) \wedge \neg\mathtt{mInhibit} \wedge \mathtt{mDOIStatus} = \mathtt{off} \Rightarrow \mathtt{cWakeUpDOI}'$
$\hat{P}_2$	**FT**	$\mathtt{mcStatus} = \mathtt{standby} \wedge @T(\mathtt{mAltBelow}) \wedge \neg\mathtt{mInhibit} \wedge \mathtt{mDOIStatus} = \mathtt{off} \wedge \neg\mathtt{mAltimeterFail} \Rightarrow \mathtt{cWakeUpDOI}'$
wP_2	**FT**	$N' \Rightarrow P_2$
P_3	**FT**	$\mathtt{mcStatus} = \mathtt{awaitDOIon} \Rightarrow \mathtt{mDOIStatus} = \mathtt{off}$
H_1	**FT**	$\mathtt{mcStatus} = \mathtt{fault} \Rightarrow \mathtt{mcStatus}' = \mathtt{init} \vee \mathtt{mcStatus}' = \mathtt{fault}$
H_2	**FT**	$\mathtt{mAltimeterFail} \wedge \mathtt{mcStatus} = \mathtt{standby} \Rightarrow \mathtt{mcStatus}' \neq \mathtt{awaitDOIon}$

only P_3 is a state invariant. The other properties, P_1, P_2, H_1, and H_2, are all transition invariants.

To evaluate P_1 and P_2, both transition invariants, the compositional proof rule for transition invariants—the second rule in Fig. 4 of Sect. 4—is applied. Applying this rule to P_1 with $P = Q = P_1$ shows that P_1 also holds in **FT**. Applying the same proof rule, with $P = P_2$, shows that a weakened version of P_2, called $\hat{P}_2$ (defined in Table 8), also holds in **FT**. To evaluate P_3, a state invariant, the compositional proof rule for state invariants, the first rule in Fig. 4, is applied. Applying the proof rule shows that P_3 also holds in **FT**. Applying the property inheritance rules defined by Theorems 2 and 3 in Sect. 4 can be used to prove that **FT** satisfies other weakened properties of the original normal behavior. As an example, consider P_2. Because P_2 is a transition invariant, the property inheritance rule for transition invariants, defined in Theorem 3, is applied. This shows that **FT** inherits, from property P_2 of **ID**, the weakened property wP_2, which simplifies to $N' \Rightarrow P_2$. That is, if the new state is a normal state, then P_2 holds.

The compositional proof rules presented in Fig. 4 can also be used to prove the new properties, H_1 and H_2. Property H_1, a transition invariant, can be proven using the compositional proof rule for transition invariants, with $P = Q = H_1$. Although applying our proof rules to prove property H_1 and the other properties described above is straightforward, the proof rules in Fig. 4 are insufficient to prove H_2. To prove H_2, a more elaborate "case split" rule for proving invariants is applied: in one case we use the proof rule for transition invariants, and in the other case we use another proof method (specifically, simple determination of a transition invariant directly from the definition of the mode transitions).

To further evaluate the ASW specifications, we checked additional properties, including the property $\mathtt{DUR}(\mathtt{mcStatus} = \mathtt{standby} \wedge \mathtt{mAltimeterFail}) \leq \mathtt{FaultDur}$, whose invariance guarantees that the ASW never remains in the mode `standby` too long when all altimeters have failed. Failure to prove this property led to the discovery (via simulation using SCR's simulator [23]) that the ASW could reach a state where it remained in `standby` forever—not a desired behavior. Although the SCR specification of **FT** in this article does not fix this problem, the example shows how checking properties with verification and simulation is a useful technique for discovering errors in specifications.

5.4 Advantages of using SCR

Because SCR specifications describe the system states in terms of the values of state variables and because the system transitions are input-driven, the observations in Sect. 4.4 apply when systems are modeled using SCR. In addition, as Sect. 5.3.2 illustrates, 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 the use of mode classes, the use of tables to define the values of variables, and the description of transitions in terms of events.

As noted in Sect. 4.4, **FT**'s specification is obtained from that of **ID** by adding new input (in SCR, monitored) variables and possibly other new variables, new values of existing variables, and new fault handling transitions. SCR's notion of a mode class provides a natural way to identify the added "fault-handling states" in **FT**, namely, through new "fault-handling" modes added to the range of the mode class variable.

The SCR tabular organization makes the definition of new transitions straightforward. A table that defines the value of a variable is typically organized around a mode class, whose modes label the rows in the table describing the events (or conditions) which cause the value of that variable to change. To add new transitions for an added mode, one simply adds new rows to existing variable tables, labeling each new row with a new mode (as in Table 5). For new transitions in an existing mode, one can either split an event or condition in the original table on a condition described in terms of new variables or new variable values (as happens, for example, in rows 3a and 3b of Fig. 7), or add a new event or condition described in terms of new variables or values (e.g., rows 6, 7, and 8 of Fig. 7) .

For any fault-tolerant extension **FT** of **ID** obtained by the above techniques, establishing that **FT** is a simple fault-tolerant extension is straightforward. Suppose each row split in a table defining a normal variable produces new rows defining updated values of the variable that are either (1) the same as in the original row for **ID** or (2) among the extended values for that variable. (For example, row 3 of Table 2 splits in this way into rows 3a and 3b of Table 7.) Then, 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), and hence **FT** is simple. Conversely, if the above condition on row splits is not satisfied, then **FT** is not simple.

6 Discussion

6.1 Verifying properties

The use of property inheritance and compositional proof rules to verify properties of a fault-tolerant system has a number of important benefits.

- The structured approach to property verification which such rules provide can yield useful insights into the relationship of the extended system to the original system.
- Use of the rules can reduce the computational complexity of verifying properties of the extended system by eliminating the need to analyze the extended system monolithically. For example, applying the compositional proof rule for transition invariants, property Q need only be proven when $\neg O$ is true if Q is already a proven invariant of **ID**. This approach should be particularly effective when the new fault-handling component is relatively small compared to the original normal system component.
- The proof rules suggest a procedure, with strong potential for automated support, for proving properties of the extended system from properties of the original system.
- The proof obligations identified in the rules can be established individually using any appropriate technique, e.g., model-checking, theorem proving, or automatic invariant generation (see, e.g., [27]).

When a property of the normal system does not hold in the fault-tolerant system, weakened versions of the property may be proposed. The inheritance theorems, Theorems 2 and 3, provide standard weakenings of state and transition invariants. Another way to weaken a property is to consider the set of faults detected by **FT**, determine which faults are relevant to the property, and weaken the property by adding a restriction that the relevant faults do not occur. For example, in the ASW, property P_2 was easily weakened into property $\hat{P}_2$ by adding the restriction that the relevant fault (`mAltimeterFail`) has not yet occurred.

It is also possible that some desired properties of the fault-tolerant system capture behavior of only the fault handling portion of the system. For example, the property H_1 of the ASW describes the recovery phase of the system. In such cases, the proof effort only requires looking at the fault handling portion of the system. This is reflected in the compositional proof rules when the proof obligations for the normal behavior are vacuously true. Thus, in proving H_1 using the transition invariant proof rule, part (2) of the rule is vacuously true because $(\pi \times \pi)(H_1) = \mathit{true}$.

The SCR invariant generator [27], a tool in the SCR toolset, allows a user to automatically derive a set of state invariants from an SCR specification. One issue to be explored is the relationship between (1) the set of properties one can prove using our property inheritance and compositional proof rules and (2) the set of properties provable using automatically generated invariants. Although automatically generated invariants and other auxiliary invariants were not used to prove the properties described above, such auxiliaries will be needed to prove properties of real-world systems. Thus, new compositional proof rules need to be investigated that admit the use of auxiliaries.

A problem our method does not currently address is what to do when proofs fail, either for properties of the normal or the fault-tolerant system behavior. Many model-checkers can demonstrate the invalidity of a property with a counterexample, but for theorem provers a failed proof may simply indicate that proving the property requires additional auxiliary invariants. Hence, dealing with failed proofs requires further investigation.

6.2 Composition

The concept of or-composition of state machines in Sect. 4.2 is designed to facilitate adding new possible behaviors to an existing system model. When the model is specified in terms of state variables, a fault-tolerant extension can be defined by adding a natural set of extensions to the specification. These extensions can be used in more than one way to define a new state machine that can be or-composed with the original state machine, augmented with new state variables, to obtain a model of the extended system. The state set of any such new state machine must include, at a minimum, (1) all new states in which either some new state variable has a non-initial value or an existing state variable has a new value, and (2) any state, old or new, which is an endpoint of some added transition. To avoid problems with specifying this state set, we can simply choose it to be the full extended state set implied by the addition of new variables and new values for existing variables. The transition relation of the new state machine is the set of added transitions. The new machine need not have any initial states, but if one uses the full extended state set as its set of states one can choose as its initial state set the original initial state set extended with initial values for the new variables.

6.3 Incremental, model-based development

There are many advantages to developing a fault-tolerant system using an incremental approach. 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 [10] uncovered modeling errors previously unnoticed (see Sect. 5.3.2.4). It is also possible that errors may be detected earlier than they would have in a single phase development—some errors may be discovered during the specification and analysis of the normal system behavior, before fault handling is modeled. Finally, specifications of the fault-handling components may be reused in other systems.

The model-based approach to incrementally developing and verifying fault-tolerant systems and the formal foundation presented in Sects. 3 and 4 can be generalized to support the development and verification of other classes of critical software systems. One obvious extension of our method is to the design and verification of exception handling. Once the normal system behavior has been specified, exceptions should be identified and a design to handle these exceptions formulated. Exception handling may be viewed as a generalization of fault-tolerance: Some exceptions will be tolerated; others will result in a notification, e.g., to an operator, that an exception has occurred which needs attention; and others, which may be extremely serious, will cause the system to halt in some safe state. A second planned extension of our method is to the design and verification of secure systems. While some security properties, such as data separation [25], can be modeled at the system level and thus can be represented as part of the externally-visible normal system behavior, other security-relevant properties, e.g., the correct management by the software of context switching, are not externally visible and hence must be modeled at a lower level of abstraction. How to develop software systems which satisfy both abstract and more concrete security properties using an incremental, model-based approach, refinement, and compositional proof rules is an open research question.

6.4 Automating verification and compositional extension

In reasoning about fault-tolerant extensions, automated support is clearly desirable. The conceptual framework presented in Sect. 4 has allowed us to prove, using PVS [40], most of the theoretical results on partial refinement and fault-tolerant extension (e.g., proofs of the theorems and soundness of the compositional proof rules). Moreover, PVS was used to develop the property inheritance and compositional proofs of the properties, listed in Table 8, of the **FT** model of the ASW and to establish that the **FT** model of the ASW is a simple fault-tolerant extension of **ID**. Much of this effort should be useful in our future plan to develop tools supporting the specification and verification of fault-tolerant systems using our method. Below, we describe several obvious targets for automation.

That a fault-tolerant extension is simple and fully faithful can often be demonstrated automatically. For example, for full faithfulness, it is sufficient to show that for every transition (s_1, s_2) in ρ_{ID}, and for every $\bar{s}_1$ in S_{FT} for which $\pi(\bar{s}_1) = s_1$, there is a transition $(\bar{s}_1, \bar{s}_2)$ in ρ_{FT} such that $\pi(\bar{s}_2) = s_2$. This is usually straightforward in an input-driven system, where one can compute the needed $\bar{s}_2$ as the result state from $\bar{s}_1$ in **FT** upon the same input event that triggered the original transition in **ID**, and then check that $\pi(\bar{s}_2) = s_2$. With automatic checking of simplicity and full faithfulness, it is easy to infer that various results from Sect. 4 apply. For example, if **FT** is a simple extension of **ID**, then Theorem 1 implies that $O(s_1, s_2)$ can be computed as $N(s_1) \wedge N(s_2)$, simplifying automated support for the compositional proof rules in Fig. 4.

Automated support is also possible when the property inheritance rules are applied. In particular, the proof using the second half of Theorem 2 that a weakened version of a one-state invariant P of **ID** can be inherited by **FT** requires two automatable checks: that P is inductive in **ID** and that the images under π of all reentry and exceptional target states in **FT** satisfy P. Applying the second half of Theorem 3 requires just one automatable check: that $\rho_{ID} \Rightarrow P$. (Automated support for the use of the first parts of Theorems 2 and 3 is not as straightforward because establishing the reachability of certain states is required.)

The compositional proof rules can be used in a theorem prover to structure a proof of the conclusion of the rule by setting up subgoals for the prover that correspond to the hypotheses of the proof rule. Because each proof rule hypothesis is of a specific form, developing specialized strategies in the theorem prover to provide additional automation for each subgoal proof should be possible. For example, the proof that a property P respects π (the first hypothesis in both proof rules) is simply a check that P involves only variables and values from the original system **ID**. In some cases, such as the proof of H_1 in the ASW example, respect of π is vacuously true.

Providing automated support for the construction of an FT-extension from an ideal system is easier to perform at the specification level because templates can be designed that take advantage of the structured format of the specification. For example, to provide automated support for constructing an extension in an SCR or other tabular specification, the user could be provided with templates for defining new state variables, for augmenting the set of values of existing variables, and for adding rows to and splitting rows in the tables specifying variables (which adds transitions to the underlying model).

7 Related work

A number of researchers have developed formal approaches to the construction of fault-tolerant systems. Our model fits the formal notion of masking fault-tolerance of [5, 33], but instead of expressing recovery as a liveness property, we express recovery in terms of bounded liveness, which is more practical. Other compositional approaches to fault-tolerance describe fault-tolerant detectors and correctors that are composed with the original system [5], and the automatic generation of fault-tolerant systems from fault-intolerant ones [4, 33]. Arora, Attie, and Emerson [4] describe the generation of the entire system (both normal and fault-tolerant behavior) from temporal logic (CTL) specifications.

Concepts closely related to partial refinement have also been developed. Cau and de Roever [13] introduce the notion of relative refinement, meaning refinement restricted to states when faults do not occur. This notion is somewhat weaker than our formulation of fault-tolerant extension, which allows faults to occur as long as their effects have not caused a discrepancy in behavior in the ideal system. For example, in the ASW, failure of all altimeters has no effect until initialization is complete. Arora and Kulkarni [5] require the non-interference of the added detectors and correctors with the original system behavior. That is, the original system behavior must be preserved in the fault-tolerant system. This is similar to our construction of faithful fault-tolerant extensions.

Our notion of fault-tolerant extension is most closely related to the notion of *retrenchment* formulated by Banach et al. [8] and the application of retrenchment to fault-tolerant systems [7]. General retrenchment is a means of formally expressing normal and exceptional behavior as a formula of the form $A \Rightarrow B \vee C$, where $A \Rightarrow B$ represents normal behavior when refinement holds, but when the refinement does not hold, the formula must be extended with $A \Rightarrow C$ to include the exceptional behavior. 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 represents the added 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, $O(s_1, s_2) \wedge \rho_{\mathrm{FT}}(s_1, s_2) \Rightarrow \rho_{\mathrm{ID}}(\pi(s_1), \pi(s_2))$ and $\neg O(s_1, s_2) \wedge \rho_{\mathrm{FT}}(s_1, s_2) \Rightarrow \gamma(s_1, s_2)$, thus providing the basis for our theory of fault-tolerant extension and the development of compositional proof rules.

Many researchers only consider classical refinement of fault-tolerant systems [35, 36] or "observational equivalence" [9] (i.e., refinement in both directions). Classical refinement is well-suited to implementation of transparent masking fault-tolerance, often using redundancy. In contrast, partial masking fault-tolerance tolerates weaker invariant properties when the system is faulty, and thus requires a different approach based on the concepts of partial refinement and full faithfulness.

In many formalisms, such as process algebras [9] and the method of Kulkarni and Arora [33], faults are modeled by adding a non-deterministic choice between a normal transition and a fault transition. This should not be confused with our notion of or-composition, which is our means of introducing new fault-handling behavior. Because our models are input-driven, faults (not counting timeouts) are modeled by special monitored events.

Our extension of normal behavior with added fault-tolerant behavior may be viewed as a transformation of the normal system into a fault-tolerant one. Gärtner [19] classifies various formal transformational approaches, including [35, 36]. This approach is also found in Katz's formal treatment of aspect-oriented programming [30]. 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 Fig. 3 when there are no exceptional target states and every entry state maps under π to a reachable state in **ID**. In this case, analogs of Theorems 2 and 3 would hold for the augmented system.

8 Conclusions and future work

This article has presented a new method for specifying and verifying the required behavior of a fault-tolerant system. This method provides a structured alternative to the monolithic ad hoc construction and verification of fault-tolerant systems. Our theory of or-composition, partial refinement, fault-tolerant extension, and full faithfulness provides a formal foundation for the method and defines a set of property inheritance and compositional proof rules which facilitate proving properties of fault-tolerant extensions. Formal proofs of state and transition invariants capturing desired system behavior, together with properties inherited through partial refinement or 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 [23], where increased confidence of correctness can be obtained via simulation, model-checking, and proofs of invariants. The SCR specification of the ASW presented in Sect. 5 demonstrates how the SCR language and tools can be used to support our method. However, the method, its supporting theory, and the proof rules are generally applicable to any software development framework in which components are described using state machine models. SCR is only one such framework.

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 Sect. 4 provides the basis for formulating additional compositional proof rules, a topic for future research. We also plan to explore the utility of our approach for fault-tolerance techniques other than masking. For example, omitting recovery is a method for modeling fail-safe fault-tolerance.

In future research, we plan to extend our method for verification. As noted in Sect. 6.1, new techniques are needed to handle failed proofs and to provide compositional proof rules which allow the use of auxiliary invariants. Further, 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. Hence, as discussed in Sect. 6.3, we plan to explore the application and extension of our model-based method and its formal foundation to other classes of critical systems, including systems with exception handling and secure systems. How to develop critical software systems in general using an incremental, model-based approach, refinement, and compositional proof rules is an open research question.

We also plan to explore the use of tools which support our methods. For example, tools which help a user extend a model of normal system behavior to capture fault-handling behavior would be extremely useful. Moreover, tools which facilitate the application of the property inheritance and compositional proof rules described in Sect. 4 would also have significant utility. Especially important are tools which reduce the amount of user ingenuity needed to prove properties with our proof rules. Finally, automatic construction of efficient source code from the **FT** specification using the SCR code generator [39] and other code synthesis techniques is another planned research topic.

Acknowledgements The authors thank Sandeep Kulkarni for useful discussions on applying formal methods to fault-tolerant systems, the anonymous referees whose insightful comments have helped us to improve this article, and Ramesh Bharadwaj for his helpful comments on an earlier version of the article.

References

1. Abadi M, Lamport L (1991) The existence of refinement mappings. Theor Comput Sci 82(2):253–284
2. Abadi M, Lamport L (1993) Composing specifications. ACM Trans Program Lang Syst 15(1):73–132
3. Alpern B, Schneider FB (1985) Defining liveness. Inf Process Lett 21(4):181–185
4. Arora A, Attie PC, Emerson EA (1998) Synthesis of fault-tolerant concurrent programs. In: Proc PODC'98, pp 173–182
5. Arora A, Kulkarni SS (1998) Component based design of multitolerant systems. IEEE Trans Softw Eng 24(1):63–78
6. Avizienis A, Laprie JC, Randell B (2000) Fundamental concepts of dependability. In: Proc information survivability workshop (ISW-2000), Boston
7. Banach R, Cross R (2004) Safety requirements and fault trees using retrenchment. In: Heisel M, Liggesmeyer P, Wittman S (eds) Proc SAFECOMP-04
8. Banach R, Poppleton M, Jeske C, Stepney S (2007) Engineering and theoretical underpinnings of retrenchment. Sci Comput Program 67:301–329
9. Bernardeschi C, Fantechi A, Simoncini L (2000) Formally verifying fault tolerant system designs. Comput J 43(3):191–205
10. Bharadwaj R, Heitmeyer C (2000) Developing high assurance avionics systems with the SCR requirements method. In: Proc 19th digital avionics sys conf
11. Bharadwaj R, Sims S (2000) 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
12. Börger E, Stärk R (2003) Abstract state machines: a method for high-level system design and analysis. Springer, Berlin
13. Cau A, de Roever WP (1993) Using relative refinement for fault tolerance. In: Woodcock J, Larsen PG (eds) Formal methods Europe 1993 symp (FMS'93), Odense, Denmark. LNCS, vol 670. Springer, Berlin, pp 19–41
14. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
15. Dijkstra EW (1997) A discipline of programming. Prentice Hall PTR, Upper Saddle River

16. Ebnenasir A (2005) Automatic synthesis of fault-tolerance. Ph.D. thesis, Michigan State Univ., East Lansing, MI
17. France R, Rumpe B (2007) Model-driven development of complex software: a research roadmap. In: 2007 future of software engineering (FOSE'07). IEEE Comput Soc, Los Alamitos
18. Garland SJ, Lynch N (2000) Using I/O automata for developing distributed systems. In: Leavens GT, Sitaraman M (eds) Foundations of component-based systems. Cambridge Univ Press, Cambridge, pp 285–312
19. Gärtner FC (1999) Transformational approaches to the specification and verification of fault-tolerant systems: formal background and classification. J Univ Comput Sci 5(10)
20. Halbwachs N (1993) Synchronous programming of reactive systems. Kluwer Academic, Boston
21. Harel D (1987) StateCharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
22. Heimdahl MPE, Leveson N (1996) Completeness and consistency in hierarchical state-based requirements. IEEE Trans Softw Eng 22(6):363–377
23. Heitmeyer C, Archer M, Bharadwaj R, Jeffords R (2005) Tools for constructing requirements specifications: the SCR toolset at the age of ten. Comput Syst Sci Eng 20(1):19–35
24. Heitmeyer C, Bharadwaj R (2000) Applying the SCR requirements method to the light control case study. J Univers Comput 6(7):650–678
25. Heitmeyer CL, Archer MM, Leonard EI, McLean JD (2008) Applying formal methods to a certifiably secure software system. IEEE Trans Softw Eng 34(1):82–98
26. Heitmeyer CL, Jeffords RD, Labaw BG (1996) Automated consistency checking of requirements specifications. ACM Trans Softw Eng Methodol 5(3):231–261
27. Jeffords R, Heitmeyer C (1998) Automatic generation of state invariants from requirements specifications. In: Proc 6th ACM SIGSOFT symp on foundations of software eng
28. Jeffords RD, Heitmeyer CL (2003) 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, pp 28–37
29. Jeffords RD, Heitmeyer CL, Archer M, Leonard EI (2009) A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In: Cavalcanti A, Dams D (eds) Proc FM2009: formal methods, 2nd world congress, Eindhoven, The Netherlands. LNCS, vol 5850. Springer, Berlin, pp 173–189
30. Katz S (2006) Aspect categories and classes of temporal properties. In: LNCS, vol 3880, pp 106–134
31. Kiczales G, Lamping J, Medhekar A, Maeda C, Lopes CV, Loingtier J.-M., Irwin J. (1997) Aspect-oriented programming. In: Object-oriented programming (ECOOP97). LNCS, vol 1241. Springer, Berlin
32. Kulkarni S (2010) Personal communication
33. Kulkarni SS, Arora A (2000) Automating the addition of fault-tolerance. In: Joseph M (ed) FTRTFT'00. LNCS, vol 1926. Springer, Berlin, pp 83–93
34. Lamport L (1994) The temporal logic of actions. TOPLAS 16(3):872–923
35. Liu Z, Joseph M (1992) Transformations of programs for fault-tolerance. Form Asp Comput 4(5):442–469
36. Liu Z, Joseph M (1999) Specification and verification of fault-tolerance timing, and scheduling. ACM Trans Program Lang Syst 21(1):46–89
37. Miller SP, Tribble A (2001) Extending the four-variable model to bridge the system-software gap. In: Proc 20th digital avionics sys conf
38. Parnas DL, Madey J (1995) Functional documentation for computer systems. Sci Comput Program 25(1):41–61
39. Rothamel T, Heitmeyer C, Leonard E, Liu A (2006) Generating optimized code from SCR specifications. In: Proc ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems (LCTES), pp 135–144
40. Shankar N, Owre S, Rushby JM, Stringer-Calvert DWJ (2001) PVS Prover Guide, Version 2.4. Tech. rep., Computer Science Lab, SRI International, Menlo Park, CA
41. Szyperski C (2002) Component software: beyond object-oriented programming. Addison–Wesley Longman, Boston