Applying a Compositional Method to Incrementally Prove Critical Properties of an Airlock System

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

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375
{leonard,jeffords,archer,heitmeyer}@itd.nrl.navy.mil

Abstract. Providing assurance that a software system satisfies its critical properties is difficult, particularly when the system must satisfy many classes of properties, such as safety, fault-tolerance, timing, and security. This paper describes the application of an incremental development and verification approach [13, 14], based on composition and refinement, to an airlock system. Initially, a model of the basic functional behavior of the system is developed and proved to satisfy a set of safety properties. This basic model is then extended with timing behavior. This timed model is a full refinement of the basic model, and related safety properties are shown to still hold for the timed model. In the third and last step, the timed model is extended with fault handling behavior. This "fault-tolerant" model, a partial refinement of the other two models, is shown to satisfy both weakened versions of the safety properties and additional fault-tolerance properties.

1 Introduction

Providing assurance that a software system satisfies its critical properties is difficult, particularly when the system must satisfy different classes of properties, such as safety, fault-tolerance, timing, and security. Two well-known theoretical approaches to developing high assurance software are refinement [1] and composition [2]. In a refinement-based development, an abstract model of the system is developed and critical properties are proved to hold for the model. Then, the model is iteratively refined, adding more detail about the system behavior during each iteration. Each model is shown to be a refinement of the model from the previous iteration, and thus, many properties which hold in the previous model may be inherited by the new model. (This is true for a large class of properties, including all safety properties.) A major difficulty with refinement-based approaches is that maintaining a refinement throughout the development process is difficult, unless the original model contains some mention of every possible system behavior.

Compositional approaches to software construction generally rely on proof rules that allow properties of individual components to be used to infer properties of the composite system. The proof rules require each component and its properties to satisfy a set of conditions before the rules can be applied. Applying the rules in practice is challenging because it is difficult either 1) to find components whose composition is a system with the desired composite behavior and that also satisfy the requirements for applying the rule, or 2) to decompose the system into components that satisfy the requirements for applying the rule.

In [13, 14], we describe a software construction and verification method that combines refinement and composition. Like other development methods, our method builds a system model (in the form of a state machine) iteratively, starting with an abstract model of the functional behavior (e.g., services) required of the basic system. In each new iteration, the model is extended by composing it with a new component that describes a different aspect of the system's behavior (e.g., fault-tolerance). This extended model is a *partial refinement* of the previous model because it inherits the behavior of the basic model but may also add new externally visible behavior. For example, the basic system model may assume that no faults occur; the extended system model removes the no-faults assumption, thus allowing faults to occur (e.g., notification of an engine failure in a flight control system), and adds externally visible behavior (e.g., an alarm to notify the pilot of an engine failure). Using the *property inheritance rules* that are part of our method, and assuming that the extended model has been constructed appropriately, we have shown that the extended model inherits weakened versions of the properties proven to hold in the basic model [13, 14]. Using the *compositional proof rules* associated with our method, we can prove properties of the extended model from properties already proven of the basic model. These proof rules can be used to prove additional properties about the extended model, for example, to prove fault-tolerance properties when the model has been extended to support fault-tolerance.

In [13, 14], an example is presented to illustrate how our method might be applied in two phases; the basic model was developed in the first phase, and the extended model, a fault-tolerant model, in the second phase. This paper extends that work by showing how our method can be applied in many steps (in our new example, three steps), where at step $n+1$, additional behavior is added to the model developed at step n. The paper provides an example where it makes sense to use a full refinement step in addition to the partial refinement step used in [13, 14]. In particular, this paper describes how our method can be applied to an airlock system, based on the example introduced in [16], by 1) specifying a model of the basic safety-critical behavior of the system and proving properties of this basic model, 2) extending the basic model with timing behavior and proving properties about the timed model, and 3) extending the timed model with fault-tolerance and proving more properties about this fault-tolerant model. In developing these three models of the airlock system, some new theoretical results, including a new compositional proof rule, were needed. This paper presents these new results and discusses some lessons learned in applying our method to the airlock example.

The paper is organized as follows. Section 2 reviews the key theoretical results of [13, 14]. Section 3 presents the airlock example, Section 4 discusses lessoned learned, and Section 5 presents some conclusions and our plans for future work.

2 Iterative Software Construction and Verification Process

References [13, 14] describe a method for compositionally constructing and verifying fault-tolerant systems, illustrate how the method can be applied to a two-phase specification of a fault-tolerant system, and present a formal theory to support the method, based on composition and partial refinement. Many popular state-machine-based formalisms are suitable for use with the method, including Abstract State Machines (ASM) [5], I/O Automata (IOA) [7], Lustre [8], Requirements State Machine Language (RSML) [10], Software Cost Reduction (SCR) [11], StateCharts [9], and Temporal Logic of Actions (TLA) [15].

In [13, 14], our application of the method to a fault-tolerant system was applied in two phases. In the first phase, the normal behavior (assumes no faults are possible) is specified, and critical system properties are proved to hold for the model. In the second phase, the no-faults assumption is removed, and the model is extended to include fault detection, handling, and recovery. The extended model is then shown to satisfy the critical system properties (possibly weakened) and additional fault-tolerance properties.

In this paper, we extend the method to allow multiple development/verification steps, where at step $n+1$ additional behavior is added to the model developed at step n. In each step, the additional behavior captures some new aspect of the system behavior (e.g., security, fault-tolerance, timing). A system model in our method is a state machine (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 $1 \leq i \leq n$ $(1 \leq i)$. 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 our extended method, the new system model created at each step is required to be a *partial refinement* of the previous system model. A special case of a partial refinement is (ordinary) *refinement*. Specifically, we define a refinement and a partial refinement as follows:

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

An extended model that is a refinement of an existing model will inherit many properties of the existing model, including all safety properties; hence, it is useful to establish that the extended model is a refinement. One natural approach to constructing a refinement is to add detail to an existing state machine model by describing individual transitions as being implemented by sequences of smaller transitions. The more detailed state machine thus has additional (intermediate) states and additional (intermediate) transitions. When the additional states can be mapped to existing states in such a way that all transitions in the detailed state machine map to transitions in the original state machine, the detailed state machine will be a refinement of the original state machine. Note that including stutter steps in our definition of state machine helps to make this possible by allowing intermediate transitions in the detailed state machine to map to stutter steps in the original state machine.

In a (proper) partial refinement step in our method, we extend the current model by composing it with a new component model, where transitions to, from, and inside the new component represent visible system behavior. In this case, the extended model has visible behavior not present in the previous model, and cannot be a proper refinement of that model. The partial refinement relation is still useful to establish, however, for verification purposes, since it allows a weakened version of property inheritance. Finally, we have also proved that the composition of partial refinements is a partial refinement. Hence, every system model obtained by applying our compositional method will be a partial refinement of the original, basic system model.

In [14], we define a special relationship between two models called *fault-tolerant extension*.

Definition 3. 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 $\pi : N \rightarrow S_{ID}$ 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**. *When $O = N \times N$,* **FT** *is a* simple *fault-tolerant extension of* **ID**.

If **FT** is a simple fault-tolerant extension of **ID**, then π is a partial refinement mapping from **FT** to **ID**. A construction method is provided in [14] which, when used to obtain **FT**, guarantees **FT** to be a fault-tolerant extension of **ID** that is "faithful" in the sense that every execution possible in **ID** is possible in **FT** (with essentially the same visible behavior). The construction method builds a fault-tolerant extension by extending a model in three ways:

1. *New variables are added to the set of existing variables.* These variables may include new input 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 output 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., fault recovery). 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 predicate involving new variables or new values of existing variables: if the predicate is true, then the transition in the fault-tolerant system corresponds to the original system transition; 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 [3], i.e., do not interfere with the original system behavior described by **ID**. In Section 3, we construct a fault-tolerant extension of the airlock as the final step in the software development process.

Reference [14] defines a set of property inheritance rules for fault-tolerant extensions that allow **FT** to inherit weakened versions of properties proved for **ID**, and two compositional proof rules that can be used to prove properties of **FT** using properties already proved for **ID**. Those property inheritance and compositional proof rules only allow a property P proved for **ID** to be used in establishing that a property Q holds in **FT**. Sometimes, this information is not sufficient to prove the desired property. In such cases, allowing the proof to use an auxiliary invariant is sometimes all that is necessary to obtain a proof. Since the publication of [14], additional property inheritance rules have been developed that allow an auxiliary invariant to be used in proving transition invariants and state invariants of **FT**.

3 Case Study: Applying the Method to an Airlock System

This section shows how the method described in Section 2 can be applied to a practical system, an airlock, which, for example, allows divers to exit and enter a submarine or astronauts to exit and enter a space vehicle. An airlock is a chamber connecting two areas with differing pressures. Each area has a door connecting it to the airlock, and the door between the chamber and one of the areas should only be opened after the pressure in the chamber has been equalized with the pressure of that area. In applying the method, the initial specification describes the safety-critical behavior of the airlock. In the second and third phases, the specification is extended first with timing behavior and then with fault-tolerance behavior.

All three of the specifications presented in this paper are represented in the SCR tabular notation [11]. In SCR a set of tables are used to define a state machine model $A = (S_A, \theta_A, \rho_A)$. The set of states S_A is determined by the values assigned to the set of state variables which specify the required system behavior. In SCR, monitored and controlled variables represent the externally visible input and output behavior of the system. SCR also has additional "hidden" variables—namely, mode classes and terms, which are used to make the specification of the relationship of the monitored and controlled variables more concise. SCR tables are used to specify how the value of each controlled variable, mode class, or term changes in response to changes in the monitored variables. In SCR tables, two other constructs are important: 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 $\neg c \wedge c'$, the semantics of @`F`(`c`) by $c \wedge \neg c'$, and the semantics of @`C`(`c`) by $c \neq c'$. 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[1], and updates to the remaining state variables deterministically defined by the SCR tables. This process is synchronous: the system completely processes one monitored event before processing the next monitored event.

3.1 Modeling and Verifying the Safety-Critical Behavior

The basic airlock specification and some of the properties proved of it are based on the example given in [16]. The basic airlock is designed to control two doors and the pressure in the chamber between them. These two doors and the chamber pressure are represented in the SCR specification by the controlled variables `cInDoor`, `cOutDoor`, and `cChPres`. In the initial specification, no timing constraints on the opening and closing of the doors are assumed. Nor are there constraints on the time needed to equalize the pressure in the chamber. Thus, each door has the value of either `closed` or `open`, and the chamber pressure is either equal to the pressure on the other side of the inside door (`InPres`) or on the other side of the outside door (`OutPres`). The airlock responds to operator commands, represented as values of a monitored variable `mCmd`. The possible commands are `OpenOutDoor` and `CloseOutDoor` for operating the outside door; `OpenInDoor` and `CloseInDoor` for operating the inside door; and `ToOutPres`

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

Table 1. Event table defining cInDoor in **ID**

Variable	Event	Event
	@T(mCmd = OpenInDoor) WHEN cInDoor = closed AND cChPres = InPres	@T(mCmd = CloseInDoor) WHEN cInDoor = open AND cChPres = InPres
cInDoor′ =	open	closed

Table 2. Event table defining cChPres in **ID**

Variable	Event	Event
	@T(mCmd = ToInPres) WHEN cChPres = OutPres AND cInDoor = closed AND cOutDoor = closed	@T(mCmd = ToOutPres) WHEN cChPres = InPres AND cInDoor = closed AND cOutDoor = closed
cChPres′ =	InPres	OutPres

and ToInPres for changing the pressure in the chamber to match the pressure outside or inside. Tables 1 and 2 are SCR event tables describing how the values of the controlled variables cInDoor and cChPres change in response to events. The table for cOutDoor, which is analogous to that for cInDoor, is omitted.

Table 3 lists six safety properties required of the basic airlock model. Each property was proved for the basic model using the property checker Salsa [4]. The proofs of properties P_2 through P_6 required no auxiliary invariants. In contrast, to complete the proof of property P_1, properties P_3 and P_5 were used as auxiliaries.

3.2 Adding Timing Behavior

In the second phase, the assumption that doors open and close unconstrained by time is replaced by a requirement that the doors open and close in a fixed amount of time. (The assumption about the chamber pressure changing unconstrained by time can be replaced in a similar manner.) A new monitored variable mtime, a monotonically non-decreasing integer, is added to the specification, and two constants ClosingDur and OpeningDur are introduced to represent the time required to close and open a door. To represent this, the representation of the doors in the timed model has one of four possible values: open, closed, opening, or closing. When the command to open a door is invoked, the door is assigned the value opening. If the door is opening, then when OpeningDur time has passed, the door is assigned the value open. The process is similar for closing a door. (The timed behavior of the airlock is similar to that described in [6].) The event tables for cInDoor and cOutDoor are modified to reflect these additional values; Table 4 is the modified table for cInDoor. The table for cOutDoor is analogous. The table for cChPres does not change.

We can define a mapping α from the timed specification **T** to the original specification **ID** by ignoring the value of mtime and mapping a state of **T** to a state of **ID** based on the values of the remaining variables: (1) the values opening and closing for the doors in the timed specification are mapped to open and closed in the untimed specification, while values open and closed remain unchanged; (2) the value of cChPres is preserved by the mapping (the mapping for mCmd is technical and omitted here). In the mapping, in response to a command to open a door, a door in **ID** immediately opens and then stutters while the corresponding door in **T** has the intermediate value opening (i.e., all the variables except mCmd remain unchanged). The case for closing a door is analogous. This mapping was proved to be a refinement using PVS.

Table 3. Safety Properties for the basic airlock **ID**

Name	Formal Statement	Informal Statement
P_1	NOT(cInDoor = open ∧ cOutDoor = open)	Both doors cannot be open at the same time.
P_2	cInDoor ≠ cInDoor′ ⇒ cChPres = InPres ∧ cChPres′ = InPres	Pressure is equal during door movement.
P_3	cInDoor = open ⇒ cChPres = InPres	If door open, pressure same as in indoor area.
P_4	cOutDoor ≠ cOutDoor′ ⇒ cChPres = OutPres ∧ cChPres′ = OutPres	Pressure is equal during door movement.
P_5	cOutDoor = open ⇒ cChPres = OutPres	If door open, pressure same as in outdoor area.
P_6	cChPres ≠ cChPres′ ⇒ cInDoor = closed ∧ cOutDoor = closed	Pressure only changes when both doors closed.

Table 4. Event table defining cInDoor in **T**

Variable	Event	Event	Event	Event
	@T(mCmd = OpenInDoor) WHEN cInDoor = closed AND cChPres = InPres	@C(mtime) WHEN DUR(cInDoor = opening) = OpeningDur	@T(mCmd = CloseInDoor) WHEN cInDoor = open AND cChPres = InPres	@C(mtime) WHEN DUR(cInDoor = closing) = ClosingDur
cInDoor′ =	opening	open	closing	closed

While the properties proved for **ID** may still hold for **T**, in some cases those properties are now weaker than what we actually desire to prove as a result of the refinement. In particular, properties that are conditioned on a door being open can be strengthened to be conditioned on the door being either partially or fully open (i.e., having value opening, open, or closing). For example, when a door is open fully or partially, the pressure in the chamber must be the same as the pressure on the other side of the door. Any property that explicitly mentions values for the variables whose values were refined, in this case cInDoor and cOutDoor, can be modified to take the value refinement into account. Thus, properties P_1, P_3, and P_5 are modified, replacing all references to cInDoor = open by cInDoor $\neq$ closed (shown as properties tP_1, tP_3, and tP_5 in Table 5).

Because **T** is a refinement of **ID**, **T** can inherit properties of **ID**. If P is a state invariant of **ID**, then $P \circ \alpha$ is a state invariant of **T**. Likewise, if P is a transition invariant of **ID**, then $P \circ (\alpha \times \alpha)$ is a transition invariant of **T**. Unfortunately, in some cases the inherited property is not identical to the property proved for **ID**. For example, the inherited property corresponding to P_2 is

$$(\texttt{cInDoor} = \texttt{open} \vee \texttt{cInDoor} = \texttt{opening}) \wedge \neg(\texttt{cInDoor}' = \texttt{open} \vee \texttt{cInDoor}' = \texttt{opening}) \vee$$
$$(\texttt{cInDoor}' = \texttt{open} \vee \texttt{cInDoor}' = \texttt{opening}) \wedge \neg(\texttt{cInDoor} = \texttt{open} \vee \texttt{cInDoor} = \texttt{opening}) \Rightarrow$$
$$\texttt{cChPres} = \texttt{cInPres} \wedge \texttt{cChPres}' = \texttt{cInPres}$$

Because we want to prove the stronger properties shown in Table 5 (including P_2, P_4, and P_6 which are syntactically identical to the untimed properties of the same names in Table 3), rather than the weaker properties obtained via inheritance, the desired properties were verified using Salsa. Proving properties tP_3, tP_5, and P_6 required no auxiliary invariants. In contrast, P_2 required tP_3 as an auxiliary, P_4 required tP_5 as an auxiliary, and tP_1 required both tP_3 and tP_5 as auxiliaries.

3.3 Adding Fault Tolerance

In the final iteration, the system is modified to handle the case in which opening a door is faulty—i.e., either door opens outside of user control. In such cases, both doors may be open for some nonzero period of time. Detecting this fault leads to the sounding of an alarm; a Warning alarm indicates that the other door is closed, and a Danger alarm indicates that the other door is open. The recovery for this fault is to force the offending door to close within a set time limit. The system only tolerates one faulty door at a time.

Using the method for constructing a fault-tolerant extension described in [13, 14], we add several new variables to the specification to represent fault detection and fault handling. Two new monitored variables, mOpenInDoor and mOpenOutDoor, are used to signal the system when one of the doors opens outside of user control. A new controlled variable cAlarm is introduced to model the status of the alarm; its value can be None, Warning, or Danger. A new integer constant HazardDur is introduced to represent the time required to close a door that has opened outside

Table 5. Safety Properties for the timed airlock **T**

Name	Formal Statement
tP_1	NOT(cInDoor $\neq$ closed $\wedge$ cOutDoor $\neq$ closed)
P_2	cInDoor $\neq$ cInDoor′ $\Rightarrow$ cChPres = InPres $\wedge$ cChPres′ = InPres
tP_3	cInDoor $\neq$ closed $\Rightarrow$ cChPres = InPres
P_4	cOutDoor $\neq$ cOutDoor′ $\Rightarrow$ cChPres = OutPres $\wedge$ cChPres′ = OutPres
tP_5	cOutDoor $\neq$ closed $\Rightarrow$ cChPres = OutPres
P_6	cChPres $\neq$ cChPres′ $\Rightarrow$ cInDoor = closed $\wedge$ cOutDoor = closed

Table 6. Table defining the mode transitions in **FT**

Old Mode	Event	New Mode
Normal	@C(mOpenOutDoor) WHEN (cOutDoor = closed)	FaultyOutDoor
Normal	@C(mOpenInDoor) WHEN (cInDoor = closed)	FaultyInDoor
FaultyOutDoor	@T(cOutDoor = closed)	Normal
FaultyInDoor	@T(cInDoor = closed)	Normal

Table 7. Event table defining cChPres in **FT**

Mode mcStatus	Event	Event
Normal	@T(mCmd = ToInPres) WHEN [cChPres = OutPres AND cInDoor = closed AND cOutDoor = closed]	@T(mCmd = ToOutPres) WHEN [cChPres = InPres AND cInDoor = closed AND cOutDoor = closed]
cChPres′ =	InPres	OutPres

of user control. Finally, and most importantly, a new mode class mcStatus is added to indicate the status of the fault-tolerant airlock system. The status is Normal when both doors are operating properly (all behavior that was part of **T** has this value of mcStatus in **FT**), FaultyOutDoor when the outside door is opening because of a glitch, and FaultyInDoor when the inside door is opening because of a glitch. The mode transition table for mcStatus is shown in Table 6. The second step in constructing a fault-tolerant extension is to extend the ranges of existing variables. In the example presented in [13, 14], the mode variable was extended with a new value fault to indicate when the system was in fault-handling mode. In the airlock system, there was no need for a mode class variable in the original and timed specifications because the system only had one mode of operation. Thus, when we added fault-tolerance to the airlock, we added a mode class variable, rather than extending the range of an existing mode class. In the airlock example, there is no need to extend the range of any variables.

The final step in constructing a fault-tolerant extension is to add transitions. In our SCR specification, the set of transitions is extended by adding rows to the tables which define the values of the controlled variables. Tables 7, 8, and Table 9 show the modified tables defining cChPres, cInDoor, and cAlarm. The tables for cInDoor and cChPres has been extended by first making the new values of the variables depend on the value of the mode class variable mcStatus. All transitions in the tables for cInDoor and cChPres in **T** are present in the tables for FT and are represented in the rows where the mode of mcStatus=Normal. Additional transitions are added when the value of mcStatus is either FaultyOutDoor or FaultyInDoor by adding rows to the table for cInDoor.

FT is a fault-tolerant extension of **T** in which

$N = \{s \in S_{\mathbf{FT}} : \texttt{mcStatus}(s) = \texttt{Normal}\}$;
$F = \{s \in S_{\mathbf{FT}} : \texttt{mcStatus}(s) = \texttt{FaultyInDoor} \vee \texttt{mcStatus}(s) = \texttt{FaultyOutDoor}\}$; $O = N \times N$; and
$\forall s \in N$: $\pi(s) = \hat{s} \in S_T$, where $\texttt{cInDoor}(\hat{\mathbf{s}}) = \texttt{cInDoor}(\mathbf{s}) \wedge \texttt{cOutDoor}(\hat{\mathbf{s}}) = \texttt{cOutDoor}(\mathbf{s}) \wedge \texttt{cChPres}(\hat{\mathbf{s}}) = \texttt{cChPres}(\mathbf{s}) \wedge \texttt{mCmd}(\hat{\mathbf{s}}) = \texttt{mCmd}(\mathbf{s}) \wedge \texttt{mtime}(\hat{\mathbf{s}}) = \texttt{mtime}(\mathbf{s})$.

Because **FT** is a fault-tolerant extension of **T**, **FT** inherits weakened forms of **T**'s properties. Properties wP_1–wP_5, shown in Table 10, are inherited via property inheritance rules. A new property inheritance rule allowing the use of an auxiliary invariant in the proof was necessary in proving properties wP_2 and wP_4. In this case, the necessary auxiliary invariant is

$$\begin{aligned}\texttt{AUX} = {} & [\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{cOutDoor} = \texttt{closed} \wedge \texttt{cInDoor} = \texttt{closed}] \vee \\ & [\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{cOutDoor} \neq \texttt{closed} \wedge \texttt{cInDoor} = \texttt{closed} \wedge \texttt{cChPres} = \texttt{OutPres}] \vee \\ & [\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{cOutDoor} = \texttt{closed} \wedge \texttt{cInDoor} \neq \texttt{closed} \wedge \texttt{cChPres} = \texttt{InPres}] \vee \\ & [\texttt{mcStatus} = \texttt{FaultyOutDoor} \wedge \texttt{cOutDoor} = \texttt{opening} \wedge \texttt{cInDoor} \neq \texttt{closed} \wedge \texttt{cChPres} = \texttt{InPres}] \vee \\ & [\texttt{mcStatus} = \texttt{FaultyOutDoor} \wedge \texttt{cOutDoor} = \texttt{opening} \wedge \texttt{cInDoor} = \texttt{closed}] \vee \\ & [\texttt{mcStatus} = \texttt{FaultyInDoor} \wedge \texttt{cOutDoor} \neq \texttt{closed} \wedge \texttt{cInDoor} = \texttt{opening} \wedge \texttt{cChPres} = \texttt{OutPres}] \vee \\ & [\texttt{mcStatus} = \texttt{FaultyInDoor} \wedge \texttt{cOutDoor} = \texttt{closed} \wedge \texttt{cInDoor} = \texttt{opening}],\end{aligned}$$

Table 8. Event table defining cInDoor in **FT**

Mode mcStatus	Event	Event	Event	Event
Normal	(@T(mCmd = OpenInDoor) WHEN cInDoor = closed AND cChPres = InPres) OR @C(mOpenInDoor) WHEN cInDoor = closed	@C(mtime) WHEN DUR(cInDoor = opening) = OpeningDur	@T(mCmd = CloseInDoor) WHEN cInDoor = open AND cChPres = InPres	@C(mtime) WHEN DUR(cInDoor = closing) = ClosingDur
FaultyOutDoor	FALSE	@C(mtime) WHEN DUR(cInDoor = opening) = OpeningDur	FALSE	@C(mtime) WHEN DUR(cInDoor = closing) = ClosingDur
FaultyInDoor	FALSE	FALSE	FALSE	@C(mtime) WHEN (DUR(cInDoor = opening) = HazardDur OR DUR(cInDoor = closing) = ClosingDur)
cInDoor′ =	opening	open	closing	closed

Table 9. Condition table defining cAlarm in **FT**

Mode mcStatus			
Normal	TRUE	FALSE	FALSE
FaultyOutDoor	FALSE	cInDoor ≠ closed	cInDoor = closed
FaultyInDoor	FALSE	cOutDoor ≠ closed	cOutDoor = closed
cAlarm =	None	Danger	Warning

proved by the compositional proof rule for state invariants in [14]. The properties wP_1, wP_3, and wP_5 hold by the following rule, "If $P \Rightarrow Q$ holds by propositional reasoning and if P is a state invariant, then Q is a state invariant," and because AUX $\Rightarrow wP_3 \wedge wP_5 \Rightarrow wP_1$. Property P_6 was proved using the compositional proof rule for transition invariants in [14]. Table 10 also shows new fault-tolerance properties, FT_1–FT_3, generated automatically by our invariant generator [12], and an interesting timing property, FT_4, proved using Salsa.

4 Lessons Learned

Incremental development and verification. Before developing the airlock as a series of three specifications, **ID**, **T**, and **FT**, we tried deriving **FT** from **ID** in a single step that included both the timing and fault-tolerance behavior. Developing **FT** directly from **ID** was extremely difficult because the combination of timing and fault-tolerance behavior required the addition of a significant amount of detailed behavior. It was much easier to get the final behavior correct by concentrating on the timing and fault-tolerance behavior individually in separate steps. This is one major argument for incremental development, and it held true in our case study. This experiment confirmed that the basic premise of our approach, adding only one additional aspect of the system's behavior in each step, was beneficial.

In addition, when we attempted to add both timing and fault-tolerance in a single step, proving the relationship between the models was difficult. Using the incremental approach, proving that **T** was a refinement of **ID** was the difficult part; establishing that **FT** was a partial refinement of **T** was easy (in fact, it follows by construction; see Section 2). Including both timing and fault-tolerance in one step obscured the individual pieces and made proving the relationship difficult.

Finally, we also found that it was important that each step extend the model from the previous step in a manner that was suitable for the new behavior being added. This requires the construction and verification method to be flexible, allowing some steps to be full refinements of the previous step and others to be just partial refinements. In the airlock example, a full refinement that refined the transitions of the original specification was the appropriate way to add

Table 10. Safety Properties for the fault-tolerant airlock **FT**

Name	Formal Statement
wP_1	$\texttt{mcStatus} = \texttt{Normal} \Rightarrow \texttt{NOT}(\texttt{cInDoor} \neq \texttt{closed} \wedge \texttt{cOutDoor} \neq \texttt{closed})$
wP_2	$\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{mcStatus}' = \texttt{Normal} \wedge \texttt{cInDoor!} = \texttt{cInDoor}'$ $\Rightarrow \texttt{cChPres} = \texttt{InPres} \wedge \texttt{cChPres}' = \texttt{InPres}$
wP_3	$\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{cInDoor} \neq \texttt{closed} \Rightarrow \texttt{cChPres} = \texttt{InPres}$
wP_4	$\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{mcStatus}' = \texttt{Normal} \wedge \texttt{cOutDoor!} = \texttt{cOutDoor}'$ $\Rightarrow \texttt{cChPres} = \texttt{OutPres} \wedge \texttt{cChPres}' = \texttt{OutPres}$
wP_5	$\texttt{mcStatus} = \texttt{Normal} \wedge \texttt{cOutDoor} \neq \texttt{closed} \Rightarrow \texttt{cChPres} = \texttt{OutPres}$
P_6	$\texttt{cChPres} \neq \texttt{cChPres}' \Rightarrow \texttt{cInDoor} = \texttt{closed} \wedge \texttt{cOutDoor} = \texttt{closed}$
FT_1	$\texttt{cAlarm} = \texttt{Danger} \Leftrightarrow \texttt{cInDoor} \neq \texttt{closed} \wedge \texttt{mcStatus} = \texttt{FaultOutDoor} \vee$ $\texttt{cOutDoor} \neq \texttt{closed} \wedge \texttt{mcStatus} = \texttt{FaultyInDoor}$
FT_2	$\texttt{cAlarm} = \texttt{None} \Leftrightarrow \texttt{mcStatus} = \texttt{Normal}$
FT_3	$\texttt{cAlarm} = \texttt{Warning} \Leftrightarrow \texttt{cInDoor} = \texttt{closed} \wedge \texttt{mcStatus} = \texttt{FaultyOutDoor} \vee$ $\texttt{cOutDoor} = \texttt{closed} \wedge \texttt{mcStatus} = \texttt{FaultyInDoor}$
FT_4	$\texttt{DUR}(\texttt{cOutDoor} \neq \texttt{closed} \wedge \texttt{cInDoor} \neq \texttt{closed}) \leq \texttt{HazardDur}$

timing behavior. In contrast, adding fault tolerant behavior was better handled by using a compositional construction that produced a fault-tolerant extension that was only a partial refinement of the timed model. This need for flexibility will likely hold for modeling other systems. Adding behavior such as exception handling or a new security component that performs monitoring will likely be best handled using composition and partial refinement, while adding more detailed security behavior to an abstract model that only captures security at a high level would be better handled using full refinement.

Importance of modes. Mode variables play an important role in the construction of a fault-tolerant extension. In the example specification in [13, 14], a mode variable that existed in the original specification was extended with a new mode to indicate when the extended system was handling a fault. In the airlock example, no mode class was defined in either the basic specification or in the timed specification. When the timed version is extended with fault-tolerant behavior to create **FT**, one of the new variables is the mode class `mcStatus`. This new mode class distinguishes all of the previously existing behavior (now described by mode `Normal`) from the states in which fault handling occurs (modes `FaultyInDoor` and `FaultyOutDoor`). Fault detection is indicated by transitions from a `Normal` state to one of the fault handling states, and recovery by transitions from the fault handling states to a `Normal` state. The mode variable clearly partitions the set of states based on its value into **N** and **F**, thus simplifying the process of establishing that **FT** is a fault-tolerant extension of **T**.

Use of auxiliary invariants in proving properties. Using the proof rules to establish properties of the airlock system revealed that in some cases the proof rules developed in [14] were not strong enough to produce the desired proof. In these cases, the proofs required additional information in the form of an auxiliary invariant to establish that the desired properties were satisfied. This led to the development of additional proof rules that allow the use of an auxiliary invariant in the proof. One important question is how to find the needed invariants. Discovering the auxiliary invariant needed to prove properties wP_2 and wP_4 of the fault-tolerant model (see Section 3.3) required user ingenuity. More systematic techniques are needed for discovering the needed auxiliary invariants.

Proving refinement may be difficult. In proving the refinement mapping from **T** to **ID**, an additional environmental constraint on `mCmd` was needed: The value of `mCmd` must always return to the inactive value `None` between each setting to an active value. Including such artificial constraints in the initial formal specifications is undesirable because this makes the specifications less understandable. However, such artifacts are sometimes necessary to establish required relationships (such as refinement) between two specifications. Further, the refinement mapping α seemed overly complex given the relative simplicity of the Airlock example. In the future, we shall explore alternate ways of establishing refinement in the Airlock example.

5 Conclusions and Future Work

This paper has presented a case study of incremental development of an airlock using composition and (partial) refinement to add timing and fault-tolerance to the original specification of the system's safety-critical behavior. In the process, additional theory, including a new compositional proof rule that allows invariants to be used in the proofs of properties of an extended system from properties of the basic system, was developed.

In the future, we plan to continue developing theory and methods to support different types of behavioral extensions beyond fault-tolerant extensions (for example, security extensions and error handling extensions). We also plan to develop tools to support use of the methods. Such tools include: 1) a construction tool that would guide the user in extending a model so that the resulting extended model is guaranteed to be a behavioral extension; 2) verification support for applying the property inheritance and compositional proof rules; and 3) tools for transferring the confidence developed for the model to the actual code (e.g., automatic code generation from the models or model-based testing).

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.
3. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Trans. Softw. Eng.*, 24(1):63–78, Jan. 1998.
4. R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Berlin, 2000.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
6. D. Esp. Environment-based specification of real-time interlock and control systems. pages 173 –177, jul. 1988.
7. S. J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge Univ. Press, 2000.
8. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Boston, MA, 1993.
9. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8(3):231–274, June 1987.
10. M. P. E. Heimdahl and N. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Eng.*, 22(6):363–377, 1996.
11. C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science and Engineering*, 1:19–35, 2005.
12. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Eng.*, 1998.
13. R. D. Jeffords, C. L. Heitmeyer, M. Archer, and E. I. Leonard. A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 173–189. Springer, 2009.
14. R. D. Jeffords, C. L. Heitmeyer, M. Archer, and E. I. Leonard. Model-based construction and verification of critical systems using composition and partial refinement. *Formal Methods in System Design*, to appear.
15. L. Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, May 1994.
16. I. Lopatkin, A. Iliasov, and A. Romanovsky. On fault tolerance reuse during refinement. Technical Report CS-TR-1188, University of Newcastle upon Tyne, Feb. 2010.