Formal Specification of Button-Related Fault-Tolerance Micropatterns

Mu Sun(✉) and José Meseguer

University of Illinois at Urbana-Champaign, Champaign, IL, USA
{musun,meseguer}@illinois.edu

Abstract. Fault tolerance has been a major concern in the design of computing platforms. However, currently, fault tolerance has been done mostly with just heuristics, high level probabilistic analysis and extensive testing. In this work, we explore how we can use formal patterns to achieve fault-tolerance designs and methods. In particular, we look at faults that occur in mechanical button interfaces such as button bounce, button stuck, and phantom button faults. Our primary goal is the safety of such interfaces for medical devices [7], but the methods are more widely applicable. We formally describe corresponding patterns to address these faults including button debouncing, button stuck detection, and phantom press filtering. We prove stuttering-bisimulation results for some patterns showing their fault-masking capabilities. Furthermore, for patterns where fault-masking is not possible, we prove fault-detection properties. We also instantiate these patterns to a simple instance of a button-press counter and perform execution and model checking as further validation.

1 Introduction

Idealized abstractions of computing systems allow us to build more complex applications and for more complex scenarios. One can think in terms of binary values instead of continuous voltages, and in terms of objects and messages instead of assembly-level instructions. Given the complexities of the real world, it is remarkable how accurate these abstractions can be. However, sometimes the real world behavior violates the expectation of idealized models and we refer to this type of behavior as faults.

In order to maintain the behavior of ideal models in the presence of faults, fault tolerance techniques are essential. We would like faults to be completely contained within the lower levels of design and never be exposed to the upper layers; this is the notion of *fault masking*. However, there are many cases where fault masking is impossible. In these cases, faults will inevitably be exposed to the upper layers, either by explicit fault detection or as behavioral anomalies such as extra delays and nondeterminism.

In this paper, we explore *fault-tolerance micropatterns* for button related faults including button bounce, phantom button presses, and stuck buttons.

Research partially supported by NSF Grant 13-19109.

S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 263–279, 2014.
DOI: 10.1007/978-3-319-12904-4_15

These micropatterns provide specific levels of safety for medical device interfaces in the presence of faults [7], and can be likewise applied to devices in other areas. All of these faults and fault-tolerance patterns are quite well known, but our contribution is in the formalization of these fault-tolerance models including:

(1) defining a model for button interfaces;
(2) modeling faults as a relation from ideal environments to faulty environments;
(3) describing fault tolerance methods as a design transformation pattern using parameterized modules;
(4) proving fault-tolerance results about our models using appropriate bisimulation relations; and
(5) validating of our models with execution and model checking.

Since we are dealing with faults on the interface, we mainly focus on faults in the environment. There are also other classes of faults such as internal faults (e.g. bit flips, memory corruption, computation errors). However, environmental faults and internal faults are generally handled orthogonally in the design of a system, so we focus only on environmental faults. The fault tolerance patterns that we describe in this paper all have a similar structure that is captured in Fig. 1. All fault tolerance designs have a goal, an ideal abstraction that it is trying to provide (left-hand side of Fig. 1). An ideal environment, and the ideal design will give the correct behavior of the system. However, the challenge comes when we have a faulty environment (right-hand side of Fig. 1). Just using an ideal design with a faulty environment will most likely lead to undesirable deviations in the behavior of the system. The goal then is to provide a design transformation for the system along with the fault model that will have behavior similar to the ideal. The notion of *correspondence in behavior* is an important one. In this paper, this correspondence is expressed as a *bisimulation*.

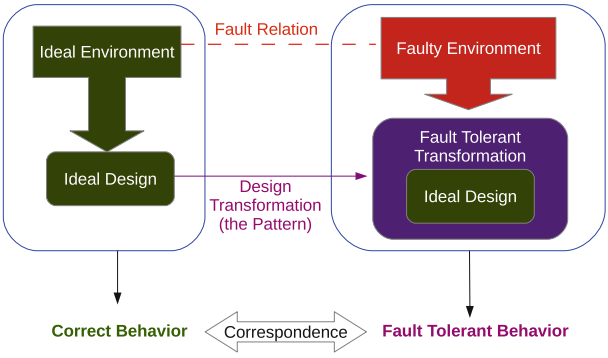

Fig. 1. Fault Modeling

The rest of the paper is organized as follows. Section 2 covers the basics of rewriting logic and the subset of Maude that we use to describe our models.

Section 3 describes how we model buttons in order to describe button-related faults. Sections 4, 5, and 6 describe in detail our patterns to handel button bounce, phantom button presses, and stuck buttons respectively. We conclude in Sect. 7 with a summary and a discussion of potential future work.

2 Background on Parameterized Formal Specifications and Real-Time Maude

We use the Maude rewriting logic language [2] to define formal specifications for our fault-tolerance wrappers for medical systems. We present some of the basic concepts behind rewriting logic, its real-time extensions, and parametrization.

2.1 Membership Equational Logic and Rewriting Logic

Membership equational logic (MEL) [5] describes the most general form of the equational components of a Maude rewrite theory. These are called functional modules in Maude [2].

A MEL signature is a tuple (K, F, S) where S is a set of sorts (i.e. types), K is a set of kinds (i.e. super types or error types for data), and F is a set of typed function symbols (and constants). A MEL theory is a pair (Σ, E) where Σ is a MEL signature, and E a set of sentences (equations and memberships) expressing (possibly conditional) membership or equality constraints. If an MEL theory is convergent (satisfies properties of confluence, termination, and sort-decreasingness), Maude provides efficient execution of its initial model semantics.

Rewriting logic [1] describes the most general form of modules defined in Maude. A rewrite theory in Maude is defined in the form of a tuple: (Σ, E, ϕ, R), where (Σ, E) is an underlying MEL theory, ϕ defines the frozen positions of operators (positions where no rewrites are allowed to occur below), and R is a set of rewrite sentences (possibly conditional on equality and membership sentences). If a rewrite theory satisfies the properties of coherence, and the underlying MEL theory of a rewrite theory is convergent, then Maude provides efficient execution of the initial model semantics for the rewrite theory. This includes efficient execution for simulation, searching and LTL model checking.

2.2 Full Maude and Real-Time Maude

Full Maude [3] is a Maude interpreter written in Maude, which in addition to the Core Maude constructs provides syntactic constructs such as object oriented modules. Object oriented modules implicitly add in sorts `Object` and `Msg`. Furthermore, OO-modules add a sort called `Configuration` which consists of a multiset of terms of sort `Object` or `Msg`. Objects are represented as records:

```
< objectID : classID | Attribute 1 : Value 1, ... Attribute n : Value n >
```

Rewriting logic rules are then used to describe state transitions of objects based on consumption of messages. For example, the following rule expresses the fact that a surgical-laser object consumes a message to set the power to 50 Watts:

```
rl setPower(sl1, 50) < sl1 : SurgeryLaser | power : P >
=> < sl1 : SurgeryLaser | power : 50 > .
```

Real-time Maude [6] is a real-time extension for Maude developed on top of Full Maude. It adds syntactic constructs for defining timed modules. Timed modules automatically import the `TIME` module, which defines the sort `Time` (which can be instantiated as discrete or continuous) along with various arithmetic and comparison operations on `Time`. Timed modules also provide a sort `System` which encapsulates a `Configuration` and implicitly associates with it a time stamp of sort `Time`. After defining a time-advancing strategy, Real-time Maude provides timed execution (`trew`), timed search (`tsearch`), which performs search on a term of sort `System` based on the time advancement strategy, and timed and untimed LTL model checking commands.

Real-time Maude provides useful constructs for specifying real-time systems, including basic semantics of time and time advancement. We use the model of linear time provided by Real-Time Maude. For time advancement, we have used the conventional best practice where only one timed rewrite rule is used and is fully determined by the operators *tick* and *mte* [6].

The *tick* operator advances time over a configuration by some time duration. For example, with timer (and time units being seconds): $tick(timer(10), 3) = timer(7)$. That is, a timer with 10 sec remaining ticked by 3 sec will become a timer with 7 sec remaining.

The *mte* operator computes the maximum time that can elapse in a system before an interesting event occurs. Interesting events include all state transitions in which messages are generated in a configuration. Again, with the timer example, we assume that components only react when the timers expire, so the maximum time elapsable for a timer would be the time it takes the timer to expire: $mte(timer(10)) = 10$.

Real-Time Maude also includes models of time that have infinity, `INF`, as a possible time value. Although, `INF` will never be used to advance time in any system, it is useful to have `INF` to describe unbounded time. For example, $mte(\texttt{stableSys}) = \texttt{INF}$.

2.3 Parameterized Modules

Modules in Maude have an *initial model semantics*. Maude also supports *theories* which have a *loose semantics* (that is, not just the initial mode, but all the models of the theory are allowed). Theories can be instantiated by *views* (i.e., theory interpretations) to other theories or modules. In particular, a theory can be instantiated by a view to any module whose initial model satisfies all equational, membership, and rewrite sentences of the theory.

Parametrized modules [2] are modules which take theories as input parameters and define operations (parametrically) in terms of the input theory. Parametrized modules are instantiated by providing views to concrete modules for the corresponding input theories. Once instantiated, the parametrized module is given the free extension semantics for the initial models of the targets of the input views. Core Maude, Full Maude, and Real-Time Maude all support parameterized modules. For our pattern, we will exploit in particular the Real-Time Maude parameterization mechanisms.

3 Modeling Buttons

Before we describe specific patterns, we should describe the problem domain that we are addressing. Many cyber-physical systems, including many medical devices, use buttons as an input interface. We need a general abstraction that can capture the important details of any button interaction with the system. This abstraction must be detailed enough to model faulty button behavior.

For the cases that we are considering, it is sufficient to use a 2-state button abstraction. A button model can be in one of two states, either *pressed* or *not pressed*, at any instant in time. Button behavior is then a function $button_{state} : Time \rightarrow \{on, off\}$. Here, $Time$ is some ideal continuous physical time, which can be represented by the positive real numbers $\mathbb{R}_{\geq 0}$. $Time$ can also be reasoned about from the perspective of a system clock that ticks (advances time) in discrete intervals, in which case we can model it using the natural numbers $\mathbb{N}$. It is desirable to prove results about our system using continuous time as it is more general. However, some of our proved results later use a discrete time model as it allows for cleaner proofs using induction and is still general enough to cover the behaviors of systems running on a system clock.

Realistic button press behaviors will have additional constraints such as buttons cannot toggle faster than a certain frequency, and we can also make some mathematical simplifications such as making all the button press intervals left-closed [7]. With these assumptions, we can model continuous button behavior with a discrete timed model, since in each finite interval of time, given a button function, b, there are only a finite number of press and release events in b. For example, if the button behavior is $b(t) = on$ for $t \in [0, 1) \cup [2, 5)$ and $b(t) = off$ otherwise. This can be represented discretely without any loss of information as a list of pairs describing when a button gets pressed and released, e.g., $(press, 0).(release, 1).(press, 2).(release, 5)$. We can easily specify this type of list structure in Maude with its expressive typing system [7].

3.1 Button Behavior Semantics in a System

The behavior of a button we have just defined is a purely mathematical one. By itself, it has no behavior semantics. To capture the behavior of the list of button press events over time, we simply convert the list of press and release events over time into a set of delayed messages:

```
op to-msgs : PressReleaseList Oid -> Configuration .
msgs press release : Oid -> Msg .
```

The `to-msgs` operator homomorphically maps each element of the list to a message.

```
eq to-msgs(nil, O) = none .
eq to-msgs(L press(T), O) = to-msgs(L,O) delay(press(O), t(T)) .
eq to-msgs(L release(T), O) = to-msgs(L,O) delay(release(O), t(T)) .
```

The object reacting to this button press event will then receive each button-related message at the appropriate time according to the semantics of the delay operator.

4 A Pattern to Address Button Bounce Faults

With our current model of the environment (button presses as delayed messages), we are now ready to discuss how to model faults. Faults essentially add additional behavior to the environment or system. In general, we would like to capture a fault in full generality in order to check all cases, but we also need to make enough assumptions to restrict in a realistic way the faulty behavior. Otherwise, it may become impossible to correctly design a fault-tolerant system.

4.1 Button Bounce

When a button is pressed, the button may "bounce." A button bounce is a mechanical phenomenon that occurs due to oscillations when a button is pressed. The contact voltages of the button may oscillate between high and low thresholds multiple times before stabilizing. This results in multiple erroneous button press events for only one intended button press event. Since oscillatory phenomena are usually dampened pretty quickly, there is a short time window, T^{max}_{bounce}, within which a button may bounce after it is pressed.

Of course, the basic model of button bouncing behavior can be described in the continuous time model as a relation $F_{bounce} \subseteq I_{valid} \times I_{valid}$ (implicitly parameterized by a maximum bounce time T^{max}_{bounce}) where $(b, b_f) \in F_{bounce}$ means that given an ideal input b, the faulty input b_f could result from the button bouncing fault [7]. However, with proper assumptions on the spacing of events to avoid zeno behavior, we can use F_{bounce} to define a corresponding relation on the discrete list-like representation of button press and release events. This is represented as the binary predicate `bounce-fault`. The first argument is the ideal input, and the second argument is the nonideal faulty input. The predicate returns true iff the faulty model is a possible result of button bounce faults applied to the ideal model.

```
op bounce-fault : Input Input -> Bool .
eq bounce-fault(nil,nil) = true .
```

If the last press events match, then we can remove it and look for earlier faults.

```
eq bounce-fault(I press(T), I' press(T)) = bounce-fault(I,I') .
```

If a press event occurs in the faulty model, which is later than the corresponding press event in the ideal model, then it is possibly a bounce event if it is within the T_{bounce}^{max} duration, `bounce-duration`. We can remove this event and analyze the earlier times for more faults.

```
ceq bounce-fault(I press(T), I' press(T'))
   = bounce-fault(I press(T), I')
   if T' le (T plus bounce-duration) /\ T' gt T .
```

Release events should match the ideal ones, but there might be extraneous release events generated by the bounce fault, which we can just remove and reason about the corresponding press event earlier (using the equations above). Anything that does not match the patterns described above could not have been generated by a bounce fault.

```
eq bounce-fault(I release(T),I' release(T)) = bounce-fault(I,I') .
ceq bounce-fault(I press(T), I' release(T')) = bounce-fault(I
press(T),I')
if T lt T' .
eq bounce-fault(I,I') = false [owise] .
```

The current fault model is purely declarative. It is a binary relation that can be used to check whether one button input is a faulty version of another. However, this gives no means for generating a faulty model directly from a nonfaulty one. In order to have some degree of completeness in model checking analysis later, we need to have a more executable fault model; one that specifies faults as transitions and not just by a predicate. Of course, if we choose $Time$ to be the real numbers, we have no hope of obtaining a set of possible faults manageable for execution purposes as there are uncountably many. However, for most practical purposes, we can obtain a fairly complete analysis just by using discrete time, mostly because systems operate based on discrete clocks anyway. Assuming a natural number model of time, a more executable fault model can be defined [7].

4.2 A Button Debouncer Pattern

Finally, we come to the most important part of our specification, namely, a formal pattern for correctly handling faulty button bounce behavior. Figure 2 shows the intuitive structure of the button debouncer. Essentially, all button inputs are filtered through a wrapper, and by properly timing button press events, we can ignore exactly the faulty bounced button press events (assuming proper spacing between normal button press events).

We must first describe the input theory `oth DEBOUNCED` that is required for a button debouncer. This includes the original class that the button debouncer will modify, and also parameters of the system and of the fault in order to adjust the pattern's behavioral parameters accordingly. The parameters of the theory `DEBOUNCED` can be intuitively described as follows. The class `Wrapped` is

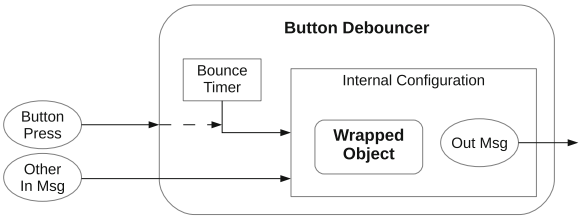

Fig. 2. The Button Debouncer Pattern

the class for the internal object that is wrapped by the button debouncer. An operator `|dest|` needs to be provided in order to know whether a message should be forwarded outside of the wrapped configuration. The constant `|t-bounce|` should be mapped to an appropriately measured constant T^{max}_{bounce}. Furthermore, another constant `t-space` is required to define the minimal time spacing between two intentional button presses. The message `press` is of course the special button press message that we want to debounce. We also add an equation in the theory specifying that time should not be allowed to advance when a press message has not yet been handled.

```
class |Wrapped| .
op |dest| : Msg -> Oid .
op |t-bounce| : -> Time .
op |t-space| : -> Time .
eq |t-bounce| lt |t-space| = true .
msg |press| : Oid -> Msg .
eq mte(|press|(O:Oid)) = zero .
```

Now, the actual pattern itself is quite straightforward. The debouncer pattern is a wrapper enclosing an object that modifies its behavior by filtering messages. Besides the internal configuration, it also adds a timer attribute, which is needed to filter the debouncing actions correctly. Note that we use parameter `|O|` as the parameter label of the theory `DEBOUNCED`.

```
(tomod DEBOUNCER{|O| :: DEBOUNCED} is
  pr RT-COMP .
  pr DELAY-MSG .

  class !Debouncer{|O|} |
     inside : NEConfiguration,
     timer : Timer .
```

The tick and mte equations are the intuitive ones, where we must tick the internal configuration according to its defined semantics as well as the timer stored in the wrapper object.

```
eq tick(< O : !Debouncer{|O|} | inside : C, timer : TM >, T)
   = < O : !Debouncer{|O|} | inside : tick(C, T), timer : tick(TM, T) > .
eq mte(< O : !Debouncer{|O|} | inside : C, timer : TM >)
   = minimum(mte(C), mte(TM)) .
```

Finally, we have the behavioral rules for the object. For receiving messages, all messages that are not a button press message are forwarded to the internal

configuration. Also, all messages output from the internal object are forwarded to the external wrapper:

```
crl [forward-in] : IM < O : !Debouncer{|O|} | inside : C >
   => < O : !Debouncer{|O|} | inside : IM C >
   if |dest|(IM) == O /\ IM =/= |press|(O) .
crl [forward-out] : < O : !Debouncer{|O|} | inside : OM C >
   => < O : !Debouncer{|O|} | inside : C > OM
   if |dest|(OM) =/= O .
```

When a button press message is received, the behavior will differ based on the timer. If the timer is not set, then we have an initial button press event, which is immediately forwarded to the internal configuration. Furthermore, the timer is set for the maximum bounce duration.

```
rl [set-timer] : |press|(O) < O : !Debouncer{|O|} | timer : no-timer, inside : C >
  => < O : !Debouncer{|O|} | timer : t(|t-space|), inside : |press|(O) C > .
```

If the timer is set, then the system is within a bounce duration, and the incoming button press event is ignored.

```
crl [ignore-press] : |press|(O) < O : !Debouncer{|O|} | timer : TM, inside : C >
   => < O : !Debouncer{|O|} | inside : C >
   if TM =/= timer0 /\ TM =/= no-timer .
```

Finally, when the timer expires, the timer is removed. This is a model-specific construct that allows the time to advance.

```
crl [reset-timer] : < O : !Debouncer{|O|} | timer : TM >
   => < O : !Debouncer{|O|} | timer : no-timer >
   if TM == timer0 .
endtom)
```

4.3 Proof of Correctness of the Debouncer Pattern

The button debouncer should essentially mitigate button bounce faults, but we must make clear this notion and what it means. We essentially need to define a correspondence between ideal behavior and the debounce pattern behavior under a faulty input. We must define the two transition systems of interest and express their correspondence. First, we define appropriate projection operations. We need a message filter and a wrapper remover. π_{nf} only projects the nonfaulty messages. π_w projects the object on the inside of the wrapper. In Maude, they can be defined as follows:

```
vars C C' : NEConfiguration .
eq pi-nf(C) = pi-nonpress(C) pi-press(C, get-time(C)) .

eq pi-w(< I:Oid : PressDebouncer | inside : C >) = C .
eq pi-w(C C') = pi-w(C) pi-w(C') .
eq pi-w(C) = C [owise] .
```

Here all these operators are frozen. `pi-nonpress` projects all the components of the configuration that are not `press` messages, and `pi-press` filters all press messages that are not faulty using the defined times `T-bounce` and `T-space`, and also the timer set on the debounce wrapper to filter initial times.

Definition 1. *States of the transition system S_{ideal} are system configurations with a single instance of a wrapped object, and such that the input button press messages are spaced by at least the assumed minimal time spacing.*

States of the transition system $S_{wrapped}$ are system configurations with a single instance of a wrapped object in a wrapper object, and such that input button press messages are related to an ideal button press configuration by the button press fault F_{bounce}.

We define a relation $H \subseteq S_{ideal} \times S_{wrapped}$ by the equivalence $s_i H s_f$ iff $\pi_{nf}(\pi_w((s_f))) = s_i$ and $time(s_f) = time(s_i)$.

We now come to the theorem that shows that H defines a bisimulation between an ideal system and a faulty system with our pattern applied. Since H preserves all the states of the object, this theorem essentially states that our pattern fully masks button bounce faults for our model of input (with proper spacing between successive button presses). The full proof of the theorem can be found in [7].

Theorem 1. *The relation H is a well-founded bisimulation, and thus H defines a stuttering bisimulation between S_{ideal} and $S_{wrapped}$ when considering natural number time.*

Note that if we do not have natural number time, then it is not guaranteed that we have a bisimulation. A simple counter-example would be one where a button bounces an infinite number of times in a finite time period. Of course, this is due to Zeno behavior. In order to remove Zeno behavior, we can make the assumption that all events are spaced at least Δt apart. This means that if we convert all times t into the natural number $\lceil t/\Delta t \rceil$, then the relation is still well founded, and the bisimulation result would still hold.

Notice that any atomic proposition AP defined on a state s_i can be lifted to a property of s_f by labelling s_f according to $\pi_{nf}(\pi_w((s_f)))$.

In addition to proving these theorems, we have also performed some model checking for simple instantiations of this pattern as an extra level of validation [7].

5 A Pattern to Address Phantom Faults

5.1 Phantom Faults

Slight disturbances in the environment (e.g. EMI, moving parts, etc.) can lead to a button being unintentionally pressed for a very short time.

The domain model is exactly the same as that for button bounce. We consider button inputs that we model as discrete messages, and an object that reacts to button inputs by consuming these messages.

A phantom button fault is a relation $F_{phantom} \subseteq I_{valid} \times I_{valid}$ (implicitly parameterized by a phantom press duration $T_{phantom}$) where faulty button presses of very short durations may occur. More precisely, $(b, b_f) \in F_{phantom}$ iff

1. $b(t) = 1 \implies b_f(t) = 1$ (an intentional button press is always registered)
2. if $b_f(t) = 1$ and $b(t) = 0$, then $t - init(b_f, t) < T_{phantom}$ (the duration of all phantom presses are bounded by $T_{phantom}$)

We can similarly construct the discrete definition of the $F_{phantom}$ relation and also the executable fault generation definitions when we are working in discrete time.

5.2 Dephantom Pattern

The pattern for handling phantom button events first requires describing the necessary parameters to fully define its behavior in the parameter theory `PHANTOMABLE`.

Like the button debouncer pattern, the dephantomizer pattern is parameterized, in this case by the `PHANTOMABLE` input theory that describes the nature of the phantom button press fault and the object which will be wrapped by the pattern. This includes a class `|Wrapped|` which specifies which object is subject to the phantom press fault. The `|dest|` operator which is again used to find which messages to forward to the outside configuration. The `|press|` and `|release|` messages which describe the actual button press events subject to phantom press faults.

```
(oth PHANTOMABLE is pr TICK-MTE-SEM .

   class |Wrapped| .
   op |dest| : Msg -> Oid .
   op |t-phantom| : -> Time .

   msg |press| : Oid -> Msg .
   msg |release| : Oid -> Msg .

   var O : Oid .
   eq mte(|press|(O)) = zero .
endoth)
```

The dephatomizer pattern takes a `PHANTOMABLE` theory as input and describes a wrapper pattern to mitigate phantom button press faults. The wrapper structure is very similar to the button debouncer, except for the logic of handling button presses, which is of course necessary since the fault behavior is different for the pattern.

```
(tomod DEPHANTOMIZER{|O| :: PHANTOMABLE} is
   pr RT-COMP .
   pr DELAY-MSG .

   class !PhantomIgnore{|O|} |
      inside : NEConfiguration,
      timer : Timer .

   op init-timer : -> Timer .
   eq init-timer = no-timer .

   vars T : Time .
   var O : Oid .
   var TM : Timer .
   var C : Configuration .
```

The equations below define the wrapper class and the time advancement semantics. This is exactly the same as in the button debouncer case. However, here the timer is used slightly differently to eliminate a different set of faults. The logic for the timer will be shown later.

```
eq tick( < O : !PhantomIgnore{|O|} | inside : C, timer : TM >, T)
= < O : !PhantomIgnore{|O|} | inside : tick(C, T), timer : tick(TM, T) > .

eq mte( < O : !PhantomIgnore{|O|} | inside : C, timer : TM >)
= minimum(mte(C), mte(TM)) .
```

The rule `set-timer` below sets the timer whenever a button press event is received. The timer is then used to make sure that the button is pressed for sufficiently long before it is actually recognized as an intentional button press event. The rule `non-phantom-release` decides the behavior when the system receives a release after sufficient time has elapsed, and hence the timer is disabled to `no-timer`. The rule `phantom-release` is applied when a release message is received before the timer expires. This means that insufficient time has elapsed before a button is released and it is considered a phantom event. Thus, the button press and the release events are hidden from the internal object. Furthermore, the timer is reset. The last rule `reset-timer` is specified when the timer expires. This means that the button press duration has just passed the threshold to be registered as a valid press. The press event is forwarded to the internal configuration.

```
rl [set-timer] : |press|(O) < O : !PhantomIgnore{|O|} | timer : no-timer >
=> < O : !PhantomIgnore{|O|} | timer : t(|t-phantom|) > .

rl [non-phantom-release] : |release|(O) < O : !PhantomIgnore{|O|} |
   timer : no-timer, inside : C >
=> < O : !PhantomIgnore{|O|} | inside : |release|(O) C > .

crl [phantom-release] : |release|(O) < O : !PhantomIgnore{|O|} | timer : TM >
=> < O : !PhantomIgnore{|O|} | timer : no-timer >
if TM =/= timer0 /\ TM =/= no-timer .

crl [reset-timer] : < O : !PhantomIgnore{|O|} | timer : TM, inside : C >
=> < O : !PhantomIgnore{|O|} | timer : no-timer, inside : |press|(O) C >
if TM == timer0 .
```

The last two rules for forwarding messages in and out from the internal configuration are similar to the forwarding rules for the debouncer pattern. Indeed, any wrapper that selectively filters certain messages will have forward rules of this form.

```
var IM OM : Msg .
crl [forward-in] : IM < O : !PhantomIgnore{|O|} | inside : C >
   => < O : !PhantomIgnore{|O|} | inside : IM C >
   if |dest|(IM) == O /\ IM =/= |press|(O) /\ IM =/= |release|(O) .
crl [forward-out] : < O : !PhantomIgnore{|O|} | inside : OM C >
   => < O : !PhantomIgnore{|O|} | inside : C > OM
   if |dest|(OM) =/= O .
endtom)
```

5.3 Proof of Correctness of the Dephantomizer Pattern

As with the button debouncer, we would like to establish a correspondence between the execution of an ideal system and that of a system with input faults but with the pattern applied. Again, the key is to define a projection relation between the two systems. However, in this case, in addition to the projection operations, we also need to define a *time translation* on button press messages to capture the delays of the pattern.

The first transformation operation of interest is the `delay-press`, which delays all press messages by a time duration `T`. This is useful as the dephantom pattern introduces delays in processing the press messages. Because of this, a delay transformation is required to show an equivalent execution between an ideal system and a delayed system. The projection $\pi_{phantom}$ from a phantom input system with a wrapper to an ideal input system with no wrapper would be the composition `remove-small ; remove-wrapper ; delay-press`. Where `remove-small` is applied first and removes all messages whose durations are too small; `remove-wrapper` removes the pattern wrapper and exposes the internal object; and `delay-press` shifts the time of all button press events by a specific duration. Full details about each of these operator definitions can be found in [7].

Again, we use the same definitions as with the button bounce case defining the states of systems S_{ideal} and $S_{wrapped}$, but this time using the phantom fault $F_{phantom}$ to provide faulty button inputs.

Definition 2. *Define a relation* $H \subseteq S_{ideal} \times S_{wrapped}$ *such that* $s_i H s_f$ *iff* $\pi_{phantom}(s_f) = s_i$ *and* $time(s_f) = time(s_i)$.

We again have a bisimulation result, for which the full proof can be found in [7].

Theorem 2. *The relation* H *is a well-founded bisimulation, and thus* H *defines a stuttering bisimulation between* S_{ideal} *and* $S_{wrapped}$ *when considering natural number time.*

Notice that in this case, H still preserves all the attributes of objects but only by making the button press delivery times later in the ideal model. This means that H adds a delay into the system, which is to be expected as detecting for faulty short button presses requires the system to wait before registering the button press event.

6 A Pattern to Address Stuck Faults

6.1 Stuck Faults

When a button is pressed, it may become stuck. This may be caused by deterioration in the spring or sudden increase in friction due to deformation or adhesives. This results in a persistent logical 1 signal, even though the button was already released.

We again have another device-button interaction, and the model is entirely similar to the button bounce and phantom press cases.

A button stuck fault is a relation $F_{stuck} \subseteq I_{valid} \times I_{valid}$ such that a faulty button may be held down for longer durations than intended, or more precisely, $(b, b_f) \in F_{stuck}$ iff:

1. $b(t) = 1 \implies b_f(t) = 1$ (a button appears pressed when it is physically pressed, regardless of being stuck)
2. If $b_f(t) = 1$ and $b(t) = 0$, then there is a $t' < t$ s.t. $b(t') = 1$ and $b_f(t'') = 1$ for all $t'' \in [t', t]$ (a button can only become stuck after it has been pressed, and stays stuck for a continuous time interval).

6.2 Stuck Detection Pattern

Like the button debouncer pattern, the stuck detector pattern takes an input theory that describes the nature of the stuck button press fault. This includes a class `Wrapped` which specifies which object is subject to the stuck button press fault. The `dest` operator is again used to find which messages to forward to the outside configuration. The `press` and `release` messages describe the actual button press events subject to stuck button press faults. Furthermore, we have `t-stuck` to describe the minimal time that the button will remain stuck. The input theory for the stuck detector pattern is given as follows.

```
(oth STUCKABLE is
   pr TICK-MTE-SEM .

   class |Wrapped| .
   op |dest| : Msg -> Oid .
   op |t-stuck| : -> Time .

   msg |press| : Oid -> Msg .
   msg |release| : Oid -> Msg .

   var O : Oid .
   eq mte(|press|(O)) = zero .
endoth)
```

The stuck detector pattern is defined in the `STUCK-DETECT` module below. It takes a `STUCKABLE` theory as input and describes a wrapper pattern to detect stuck button press faults. The wrapper structure is again very similar to the button debouncer wrapper.

```
(tomod STUCK-DETECT{|O| :: STUCKABLE} is
   pr RT-COMP .
   pr DELAY-MSG .

   class !StuckDetect{|O|} |
      inside : NEConfiguration,
      timer : Timer,
      stuck-err : Bool .

   op init-timer : -> Timer .
   eq init-timer = no-timer .
   op init-stuck-err : -> Bool .
   eq init-stuck-err = false .
```

We first define the necessary attributes of the wrapper object. Besides the internal configuration, we have a timer for keeping track of when the button has been pressed passed its stuck duration. The `stuck-err` bit, when set to true represents detection of the error. The other constants define initialization values for each of the attributes.

The tick and mte rules are again similar to those for the other patterns and work by propagating the operations homomorphically to the internal configuration and timers. Their behavior on objects are defined by the equations below.

```
eq tick( < O : !StuckDetect{|O|} | inside : C, timer : TM >, T)
= < O : !StuckDetect{|O|} | inside : tick(C, T), timer : tick(TM, T) > .

eq mte( < O : !StuckDetect{|O|} | inside : C, timer : TM >)
= minimum(mte(C), mte(TM)) .
```

The rules for the behavior under button press events is just forwarding all button press and release messages normally, but setting and resetting the timers appropriately. The last rule, `stuck-event`, is applied whenever a button press event is not followed by a release within `t-stuck` time units. When this happens, the `stuck-err` is set to true to indicate detection.

```
rl [set-timer] : |press|(O) < O : !StuckDetect{|O|} | timer : no-timer, inside : C >
=> < O : !StuckDetect{|O|} | timer : t(|t-stuck|), inside : |press|(O) C > .

rl [release-event] : |release|(O) < O : !StuckDetect{|O|} | inside : C >
=> < O : !StuckDetect{|O|} | inside : |release|(O) C, timer : no-timer, stuck-err : false > .

crl [stuck-event] : < O : !StuckDetect{|O|} | timer : TM >
=> < O : !StuckDetect{|O|} | timer : no-timer, stuck-err : true >
if TM == timer0 .
```

The forward in and out rules are again similar to the previous two patterns.

```
   var IM OM : Msg .
   crl [forward-in] : IM < O : !StuckDetect{|O|} | inside : C >
      => < O : !StuckDetect{|O|} | inside : IM C >
      if |dest|(IM) == O /\ IM =/= |press|(O) /\ IM =/= |release|(O) .
   crl [forward-out] : < O : !StuckDetect{|O|} | inside : OM C >
      => < O : !StuckDetect{|O|} | inside : C > OM
      if |dest|(OM) =/= O .
endtom)
```

6.3 Proof of Correctness of the Stuck Detection Pattern

The stuck fault is inherently lossy, so the correctness of the pattern is shown in two parts. First, if no stuck faults occur then we show that the behavior with the pattern is bisimilar to the ideal system. Second, if a stuck fault occurs, we can no longer guarantee any correspondence in behavior to the ideal case, but we can guarantee *detection* of the fault within a certain time bound.

The projection π_{stuck} from a wrapped system for stuck detection to an ideal input system with no wrapper is just simply a function `remove-wrapper`, which removes the pattern wrapper and exposes the internal object to the external configuration.

Again, we use definitions analogous to those for the button bounce case for states of S_{ideal} and $S_{wrapped}$. Although stuck faults will ruin any possibility of

behavioral correspondence (since the system becomes unresponsive), we can still show that without faults our pattern does not alter the behavior of the system.

Definition 3. *Define a relation* $H \subseteq S_{ideal} \times S_{wrapped}$ *such that* $s_i H s_f$ *iff* $\pi_{stuck}(s_f) = s_i$ *and* $time(s_f) = time(s_i)$.

We can show that under a strict relation H that does not allow for differences in the faulty model (i.e. no stuck faults occur), then the behavior of the wrapped system in a faulty environment is bisimilar to that of the ideal system, that is, the added wrapper does not essentially change to the behavior of the system. Proof in [7].

Theorem 3. *The relation* H *is a well-founded bisimulation, and thus* H *defines a stuttering bisimulation between* S_{ideal} *and* $S_{wrapped}$ *when considering natural number time.*

However when a button does become stuck, we can no longer give any guarantees about correct behavior, but we can still detect a fault. The following theorem proves that any stuck faults will be detected by our pattern. Proof in [7].

Theorem 4. *Consider a system in* $S_{wrapped}$. *If we have a stuck fault such that there exist two consecutive press and release events on the input* `delay(press,` t`)` `delay(release,` t'`)` *such that* $t' - t > T_{stuck}$ *then the wrapper attribute* `stuck-err` *will be set after* $t + T_{stuck}$ *time units.*

7 Conclusion and Future Work

The goal of this work has been to define *formal patterns*, as parameterized real-time rewrite theories, that provide provably correct guarantees of fault tolerance for commonly occuring faults in button interfaces of manually-operated devices, including medical equipment. The general technique of well-founded bisimulations [4] has been used to obtain the desired guarantees for each pattern. Since the formal specifications are executable, formal analysis by model checking has also been performed.

For future work, an important next step is to analyze the compositional behavior of multiple patterns together. Although each of the patterns have bisimulation results which is by itself composable, some of the bisimulations are conditional (such as introducing delays or adding additional fault-detection messages). In these cases the order of pattern composition can result in different system behaviors. This highly nontrivial problem of pattern composition is one of the major challenges that must be addressed before these patterns can be used for larger scale systems.

References

1. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1), 386–414 (2006)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS. Springer, Heidelberg (2007)
3. Durán, F., Meseguer, J.: The Maude specification of Full Maude. Technical report, SRI International (1999)
4. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. J. Log. Algebr. Program **79**(2), 103–143 (2010)
5. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, Francesco (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
6. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order Symbolic Comput. **20**(1–2), 161–196 (2007)
7. Sun, M.: Formal patterns for medical device safety. Doctoral Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign (2013). https://dl.dropboxusercontent.com/u/54321762/mu-thesis.pdf