FORMAL PATTERNS FOR MEDICAL DEVICE SAFETY

BY

MU SUN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Lui Sha, Chair
Professor José Meseguer
Professor Gul Agha
Doctor Raoul Jetley, ABB

Abstract

Formal methods have revolutionized software reliability and safety, and design patterns has revolutionized software reusability and modularity. However, the preciseness required for formal methods and the flexibility inherent in design patterns has rendered these two concepts somewhat disjoint and applied to different application domains. Currently, new uses of software in medical device plug-and-play systems has pointed to a need for creating systems that are both flexible and safe. In this dissertation, we describe significant advancements towards the development of formal patterns to achieve greater assurance about medical device safety. We consider three levels of safety and associated case studies in the medical device domain: device interface safety, medical requirement safety, and network safety. For device interface safety we look at various button-related faults and describe pattern solutions for addressing each fault. For medical requirement safety we focus on a particular class of stress-relax safety and present the Command-Shaper pattern to address this. Finally, in the network safety area we look at the particular case of message loss and describe an active message repeater pattern. For each of these patterns: (i) we formally define them in the Maude rewriting logic framework; (ii) we show their correctness by rigorously proving the required properties based on their rewriting logic specification; and (iii) we also show practicality of each pattern with execution, model checking, and emulation.

To My Mother

Acknowledgments

The work in this dissertation was supported in part by funding from ONR grant N000140810896 and by NSF grants 0720482 and 0834709, and NSF CNS grant 13-19109.

I owe much of the completion of my dissertation to the amazing people that have helped me through so many different aspects of the grad program. First and foremost, I must thank my advisors Professor Lui Sha and Professor José Meseguer for providing me with the path of my research and guiding me throughout the way. I must also give my sincerest thanks to my two advisors and the CS department including Professor Roy Campbell and Rhonda McElroy for helping me through difficult times and giving me the final push needed to completing my dissertation.

I would also like to thank all people at the FDA and the MD PnP: Raoul Jetley, Yi Zhang, Sandy Weininger, Julian Goldman and Susan Whitehead. They introduced me to the vision for the future of medical systems and gave many suggestions on how to make my research more relavant for the real world. I also give a big thanks to Doctor Richard Berlin for taking the time to sit in meetings with us giving me the inside perspectives of medical practice and how to best design medical systems for use by real people. I should also thank Doctor Raoul Jetley and Professor Gul Agha for being on my dissertation committee, and providing me with additional suggestions and feedback. My dissertation is also just a small piece of work built on the shoulder's of giants. I owe my thanks to all the people who helped to develop Maude rewriting logic framework in its current state. In particular, I would like to thank Peter Ölveczky for creating Real-Time Maude and answering my many questions about it.

Of course no graduate school experience would be complete without the many people to share the struggles and complaints of grad school life. I would like to thank my mentors from when I had just joined the program: Ajay Tirumala, Hui Ding, and Tanya Crenshaw. I would also like to thank my casual friends, Cheolgi Kim, Kyungmin Bae, Maryam Rahmaniheris, Michael Katleman, Abdullah Al-Nayeem, Heechul Yun, and Dongyun Jin who were always available for a fun chat and to discuss many nerdy topics.

Lastly, I must thank my mother for supporting me in everyway possible in order for me to succeed in grad school. No words can express my gratitude.

Table of Contents

List of Figures . . . . . **viii**

Chapter 1 Introduction **1**
1.1 Formal Patterns for Medical Systems 2
1.2 Practicality of Our Formal Patterns for Real-World Systems 4

Chapter 2 Background on Rewriting Logic, Maude, and Parameterized Specifications . **5**
2.1 Membership Equational Logic 5
2.2 Rewriting Logic and Kripke Structures 8
2.3 Simulations and Stuttering Simulations 9
2.4 Full Maude and Real-Time Maude 11
2.5 Parameterized Modules 13
2.5.1 Actor Reflection in Rewriting Logic 13
2.6 Socket Programming in Maude 14

Chapter 3 Patterns for Fault-Tolerant Device Interfaces **15**
3.1 Modeling Buttons 16
3.2 Modeling the System 18
3.3 Pattern to Address Button Bounce Faults 19
3.3.1 Button Bounce 19
3.3.2 Button Bounce Fault Generation 20
3.3.3 A Button Debouncer Pattern 22
3.3.4 Proof of Correctness of the Debouncer Pattern 25
3.3.5 Model Checking Case Study for the Buttion Debouncer 28
3.3.6 Model Checking Constructs 30
3.4 Pattern to Address Phantom Faults 32
3.4.1 Phantom Faults 32
3.4.2 Dephantom Pattern 33
3.4.3 Proof of Correctness of the Dephantomizer Pattern 37
3.5 Pattern to Address Stuck Faults 41
3.5.1 Stuck Faults 41
3.5.2 Stuck Detection Pattern 42
3.5.3 Proof of Correctness of the Stuck Detection Pattern 46

Chapter 4 A Device Level Safety Pattern **50**
4.1 Safety of Life-Critical Medical Devices 50
4.2 A Wrapper Pattern for Safety Monitoring 51
4.2.1 Patterns as Parameterized Specifications 51
4.2.2 Overall Idea of the Command-Shaper Pattern 51
4.2.3 Formal Models of Event Streams 53
4.2.4 Abstract Safety Definition 58
4.3 Command Shaper Pattern as a Parametrized Specification 63

4.3.1 Wrapped Object Theory . . . 63
4.3.2 Parameterized Wrapper Object . . . 65
4.3.3 Safety Envelope Calculations for the Wrapper . . . 66
4.3.4 Wrapper Execution . . . 67
4.3.5 Some Pattern Instantiations . . . 70
4.4 Model Checking Completeness and Verification . . . 81
4.4.1 Completeness of Compositional Nested Systems . . . 81
4.4.2 Model Checking of Instantiations . . . 82
4.5 Correctness of the Command Shaper Pattern . . . 91
4.5.1 Stability of the Safety Property . . . 91
4.5.2 Safety Preserving Property of the norm Operator . . . 92
4.5.3 Persistence of Safety using Safety Envelopes . . . 96
4.5.4 System Execution Assumptions . . . 97
4.5.5 Wrapper Dispatch and Well-Behaved Rule Applications . . . 97
4.5.6 Correctness of the Command Shaper Pattern . . . 99
4.5.7 Model Checking Completeness of Nested Object Configurations . . . 103

Chapter 5 A Pattern for Network Safety . . . 105
5.1 Airway Laser Fires - A Case Study . . . 105
5.2 The Issue of Open Loop Safety . . . 105
5.3 The Heartbeat Pattern . . . 106
5.3.1 Modeling Network Faults . . . 106
5.3.2 Transient Commands and Fail-Safe Modes . . . 107
5.3.3 The Heartbeat Protocol . . . 108
5.3.4 An Intuitive Yet Flawed First Attempt . . . 110
5.3.5 Ironing Out The Kinks . . . 111
5.3.6 Proof of Correctness . . . 115
5.3.7 Proof of Robustness . . . 120
5.4 A Safe Laser Surgery Protocol - An Application of the Heartbeat Pattern . . . 122

Chapter 6 Formal Model Based Device Emulation . . . 129
6.1 Distributed Emulation of Safe Medical Devices . . . 129
6.2 Mapping Internal Messages to External I/O . . . 130
6.2.1 One-Round Communication Clients . . . 131
6.3 Mapping Logical Time to Physical Time . . . 132
6.3.1 Synchronous Timed Execution . . . 133
6.3.2 Handling Asynchronous External Events . . . 136
6.4 Case Studies . . . 137
6.4.1 Pacemaker Simulation Case Study . . . 137
6.4.2 Syringe Pump Case Study . . . 139

Chapter 7 Related Works . . . 144
7.1 Formal Methods and Medical Systems . . . 144
7.2 Formal Methods and Software Patterns . . . 145

Chapter 8 Conclusion and Future Work . . . 146

Appendix A Complete Maude Specifications of Models Used . . . 148
A.1 Basic Modeling . . . 148
A.1.1 Time Advancement Semantics . . . 148
A.1.2 Real-Time Components . . . 149
A.1.3 Delayed Messages . . . 150
A.1.4 Event Logs . . . 151
A.2 Button Related Patterns . . . 153

A.2.1 Button Press Model 153
A.2.2 Button Fault Models 154
A.2.3 Debouncer Pattern 155
A.2.4 Dephantomizer Pattern 157
A.2.5 Stuck Detection Pattern 159
A.3 Command-Shaper Pattern 161
A.4 Heartbeat Pattern 165

References 169

List of Figures

1.1 ICE System 2

3.1 The Button Debouncer Pattern 23
3.2 The Dephantomizer Pattern 33
3.3 The Stuck Detection Pattern 43

4.1 Command-Shaper Wrapper Pattern for the Pacemaker 52
4.2 Event Example 55
4.3 Stress Relax Event Log 57
4.4 Characterization of the Stress Envelope 68
4.5 A state change in the system may not be immediately reflected in the patient. There may be some delay. 83

5.1 Simple Laser Protocol 106
5.2 Network Loss and Failures 107
5.3 The Heartbeat Protocol 108

6.1 Real-Time Model Execution Wrapper 130
6.2 From Ideal Time Advancement Semantics to Physical Time Advancement 133
6.3 Handling Interrupts and Asynchronous Communication Semantics 136
6.4 Trace from Pacemaker Simulator 139
6.5 Model Execution Jitter Distribution 140
6.6 Pacing periods recorded by the pacemaker simulator (jitter effects are reflected by noise on the curve) 140
6.7 Multi-Phaser NE-500 Syringe Pump 143
6.8 Infusion Volume over Time 143

Chapter 1

Introduction

In everyday life we naturally use patterns as a powerful form of abstraction that not only serves to concisely represent large amounts of information around us, but also provides a reasoning mechanism to deal with a wide range of examples that we have not yet encountered. In the same spirit, engineers have also realized that, after successfully designing many systems, a common part of a design can be extracted as a pattern. Future engineers can readily use these patterns as a starting point in design and benefit from the tried and true experience from successful designs in the past. The usage of pattern designs originated in (building) architectures [5] and was later introduced to software engineers in the famous gang of four book [21].

There is no dispute that patterns have been enormously useful in software engineering. However, current design patterns are used mostly to ensure modularity, portability, scalability, and maintainability of code. But in the present practice of patterns there is often no clear way to attach to patterns formal conditions for their applicability and formal guarantees for their behavior when such conditions are met. For example, a design pattern may be successful in many cases, but, if its applicability conditions are not precisely specified, there may be pattern instances that fail in unexpected ways. Clearly, to harness the full power of patterns in safely critical systems, patterns must become mathematically precise entities, which we call formal patterns. A formal pattern must come with:

1. Formally specified preconditions on the environment and the system in order to have correct application of the pattern.

2. Formal safety and correctness guarantees provided by the pattern, assuming that the pattern application satisfies all the preconditions.

3. Furthermore, it is desirable to have formal definitions of the pattern that are *as generic as possible* and have an *executable semantics* that can be easily translated into an implementation.

To define patterns with all these properties in mind in the context of medical devices (or just cyber physical systems in general), we need a semantic framework that naturally supports modeling real-time concurrent

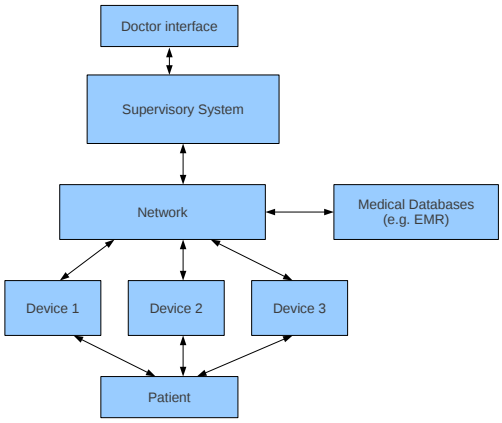

Figure 1.1: ICE System

systems and logical and actor reflection capabilities. For this, we use the Maude rewriting logic framework [16] to formally specify, simulate, and analyze our patterns.

1.1 Formal Patterns for Medical Systems

The formal patterns we will consider in this dissertation are software patterns to improve the safety of medical device operation. Safety in designs of medical devices and systems is an important issue that, when left unaddressed or addressed only partially by testing an implementation (which of course should also be done in any case), can cause severe and perhaps deadly accidents for patients. The Integrated Clinical Environment (Figure 1.1) [23] shows all the components in a typical medical system: the patient, individual devices, device supervisor(s), the network, the clinical database, and medical personnel.

Patterns for Safe Medical Device Interfaces: The first class of safety issues we consider is the inputs to the system, interaction of medical personnel with these devices. This is where all the behavior of the medical system is set and guided. If this interface is faulty, then the system may not faithfully capture the intentions of medical personnel. We call patterns in this first class *patterns for safe device interfaces* in Chapter 3. In particular, we look at various button related faults including button bounce, stuck buttons, and phantom button presses. We address each of these problem with corresponding solution patterns including formalizing button debouncing logic, button stuck detection, and button dephantomizers. For each of these patterns we prove bisimulation results that these patterns do not change system behavior under normal cercumstances. Furthermore, we show that under faulty conditions our patterns can provide additional robustness either by fault masking or fault detection.

Patterns for Safe Medical Devices: The second class of safety issues we consider are the interactions of individual devices with the patient. Over the years, medical functionalities for each device have been nicely abstracted and decoupled into different classes: e.g. infusion pumps, cardiac pacemakers, medical ventilators, etc. Each device responsibility gives a precise but limited control of the patient. The necessary properties of the device are of course successfully and safely providing its dimension of control. We call patterns in this second class *patterns for safe device/patient interaction*. Many times safety of an individual device administering treatment involves rate and time constraints. We discuss a specific pattern for enforcing these types of constraints in Chapter 4. In particular, we focus on a special class of safety properties called stress-relax safety characterizing to when the medical devices put patients into stressful, potentially unsafe, states. Furthermore, we discuss a command reshaper pattern that can modify potentially unsafe commands before they are executed by a medical device. The pattern allows many modifiable parameters such as range limitations, rate of change limitations, and also time limitations that pertain to stress-relax safety. We further prove that our pattern will indeed satisfy all of these parameterized safety properties once instantiated. We also apply our pattern to many different medical device models such as infusion pumps, pacemakers, and ventilators.

Patterns for Safe Medical Device Networking: The third class of safety issues we consider are more global: they have to do with safe device coordination. That is, some important safety properties are interaction safety properties and must be satisfied by several devices simultaneously. For example in airway-laser surgery, the flow of oxygen must be reduced before the laser can be allowed to be turned on. This is a property that must always hold in order to prevent a fire. Furthermore, coordinated safety properties can naturally conflict with local safety. For example, the oxygen cannot turn off for too long, while the global safety property needs to be coordinated for the laser. Addressing these issues while considering the possibility of network disconnects makes the problem even more difficult. We call patterns in this third class *patterns for safe networked device communication*. We discuss some specific patterns for ensuring global system safety under message loss in Chapter 5. In particular, we describe the heartbeat pattern, which transforms a design that assumes a reliable network to operate in an unreliable network but still maintain safety. We prove that our heartbeat pattern preserves all ideal behavior when no network faults occur. We also prove that under failure conditions our heartbeat pattern can preserve the safety of a networked medical system by ensuring that all devices fall to a safe independent mode of operation.

1.2 Practicality of Our Formal Patterns for Real-World Systems

In this work we demonstrate, through many examples, the technique of using formal patterns to provide provably correct templates for software design and reuse in the realm of medical device safety. Indeed, all patterns in this dissertation are described in the context of medical systems. For each of our patterns we consider a few well understood and precisely defined faults, and discuss a provably safe software pattern under this limited fault model. Of course, in real systems the set of faults and hazards is much larger, and our patterns should not be used directly to handle specific faults unless all the other faults are shown to be orthogonal. For example, for the button bounce fault, we can use a button debouncer (Section 3.3.3), but this may now introduce new timing faults to the system due to the presence of additional software. Thus, in general, we need to be careful about how patterns are applied and composed. While the guarantees provided by a pattern are precisely defined and formally proven, any required behavior that is not guaranteed must still be verified separately.

The fault model is continuously evolving as we understand more about a system through its deployment. For example, the NTSB, which investigates accidents, constantly influences certification criteria for aircraft development, and these certification criteria must be revised based on new incidents and use of new technology. Practical formal patterns need to also adapt and change with increased understanding of a system. We need domain experts to identify faults as they are found, and more time needs to be spent formalizing these faults, possibly extending existing software models. The formal patterns in this dissertation were carefully abstracted to cover the concrete examples instances we have considered, and we acknowledge the need to improve these existing pattern definitions as we learn of examples that fall outside of the current abstractions.

Thus, we view this work as an important step among others towards feasible use of formal patterns for safe medical system development. Some of the problems not addressed in this dissertation but important for future work on feasible use of formal patterns include: how to identify when a pattern can be used in an extended context of new faults; and how to compose multiple patterns that handle nonorthogonal faults.

Chapter 2

Background on Rewriting Logic, Maude, and Parameterized Specifications

We use the Maude rewriting logic language and framework [16] in order to define formal specifications for our software patterns for medical systems. We present some of the basic concepts behind rewriting logic, its real-time extensions, and parametrization.

2.1 Membership Equational Logic

Membership equational logic (MEL) [34] describes the most general form of the equational components of a Maude rewrite theory. These are called functional modules in Maude [16].

Formally, 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.

More practically, to illustrate the syntax of functional modules in Maude, we take a look at a commonly used module `TIME` used throughout this dissertation to model time. We present a simplified version of the module to just highlight the main points.

```
fmod TIME is
   sort Time .


   op zero : -> Time .
   op _plus_ : Time Time -> Time [comm assoc] .
   op _le_ : Time Time -> Bool .
   op _lt_ : Time Time -> Bool .
```

```
   eq zero plus R:Time = R:Time .
   eq R:Time le R':Time = (R:Time lt R':Time) or (R:Time == R':Time) .
endfm
```

This module is used to define the theory of time. The first line of the module defines the sort `Time`, which semantically will correspond to the set of terms that represent time. The next set of lines define operators: a 0-ary operator (i.e. a constant) `zero`; a binary operator `plus` taking two arguments of sort `Time` and returning a term of sort `Time` (it also has attributes indicating that it is a commutative and associative operator); binary operators `le` (less than or equal to) and `lt` (less than) taking two arguments of sort `Time` and returning a boolean term[1]. The next part contains equations which define semantics of the operators. The first equation expresses the fact that zero is an identity element for the set of terms of sort `Time` under the operator `plus`. The second equation shows that the less than, `le`, operator is essentially just syntactic sugar as it can be defined as a disjunction of strict less than and equality.

Now, this defines a theory of time with addition and comparison, but what about the sematics? Maude uses inital model semantics for functional modules, which is the unique model (up to isomorphism) that has unique homomorphisms to all of models satisfying the theory. Intuitively, the inital model means the "minimal" model that satisfies the theory (it introduces no additional constant terms or operators and does not equate different terms unless they are forced equal by the equations). The particular inital model that Maude uses is the *canonical term algebra*, which associates with each sort a set of constructor terms (terms constructed from constants and operators that are irreducible by the equations) and associates with each operator a function that constructs terms using that operator and reduces them by the equations to some canonical form with only constructors.

Currently the initial model for our module `TIME` is quite uninteresting as it only has one constructor term `zero`. In order to make our model of time more realistic, we need to extend it.

Time can have many different models including discrete and continuous ones. Discrete time is generally modelled with natural numbers. We can extend our model of time by defining another functional module `NAT-TIME-DOMAIN`.

```
fmod NAT-TIME-DOMAIN is
  including TIME .
  protecting NAT .
```

[1] All modules in Maude implicitly import the functional module `BOOL` which defines the sort `Bool`, constants `true` and `false`, and the common boolean operators.

```
  subsort Nat < Time .

  vars N N' : Nat .

  eq zero = 0 .
  eq N plus N' = N + N' .
  eq N lt N' = N < N' .
endfm
```

This new module specifies that it is including `TIME`, which means that it will use all the sorts, operators, and equations from `TIME` but will add more constants and equations. It also specifies that it is protecting `NAT`, which means that it will use the model of natural numbers (but will not modify the semantics of the natural numbers in any way). The next line specifies that `Nat` is a subsort of `TIME`. This means that all terms that were of sort `Nat` are now also of sort `Time` (this effectively extends the ground terms of sort `Time` with all natural numbers). The next line defines some variables for the equations introduced later. The equations identify `zero` with the natural number 0, and map the `plus` and `lt` operators to the corresponding operators for natural numbers. With this new module the semantics of time now becomes an algebra with the set of natural numbers along with addition and comparison operators. Natural number time is a very useful model of time that we will often use later in order to do finite-state model checking of timed systems.

Note that having `Nat` be a subsort of `Time` creates a partially ordered set for sorts. The connected components for sorts are called *kinds*. To have simple reasoning about sorts, Maude modules requires a property called preregularity, where every term in a kind has a least sort. For example, the term `3` would be of sort `Nat` and `Time`, but since `Nat` is subsumed by `Time`, `Nat` is the least sort. Maude's type system in full generality is *membership equational logic*, which subsumes order-sorted logic, but for practical purposes we normally do not use the full power of membership equational logic in the models we define in this dissertation, and we mostly restrict ourselves to order-sorted logic. Interested readers can consult [34] for an explanation of membership equational logic.

The semantics of modules in Maude has algebraic semantics, and algebra is essentially a characterization of structure. This structure could be as simple as the interactions natural numbers under addition, or as complex as the structure an entire software system. Either way, the hierarchical sort system in Maude together with its module structure allows us to build more complex structures on top of existing models. For example, from our model of time, we can easily define a structure for clocks.

```
fmod CLOCK is
```

```
   protecting NAT-TIME-DOMAIN .

   sort Clock .

   op c : Time -> Clock .
   op tick : Clock Time -> Clock .
   op over? : Clock -> Bool .

   vars T T' : Time .

   eq tick(c(T), T') = c(T plus T') .
   ceq over?(c(T)) = true if 24 le T .
   eq over?(c(T)) = false [owise] .
endfm
```

Here, we have just defined a new sort `Clock` that encapsulates `Time` with the `c` operator, and we define an operator `tick` that will advance the time of the clock. In the equations defining the operator `over?`, we have introduced a conditional equation to indicate that `over?` is true if the time in the clock exceeds 24. The last equation has the attribute `[owise]` which means it is only applied when no other equations can be applied.

2.2 Rewriting Logic and Kripke Structures

We just described functional modules in Maude. Functional modules define algebras which have a static structure. However, structure is not enough to define most systems. We also need the dynamics or the behavior of a system. In Maude, this is defined by a system module, whose semantics is expressed by the the idea of rewriting logic.

Formally, rewriting logic [12] 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.

To illustrate rewriting, we go back to our clock example. We define it's behavior over time with a system module.

```
mod CLOCK-EXEC is
   inc CLOCK .

   var C : Clock .
   rl [advance-time] : C => tick(C,1) .
   crl [reset] : C => c(0) if over?(C) .
endm
```

We imported all the sorts, operators, and equations defining clocks, but now we added two additional rules. The first rule states that we can tick the clock by one time unit. The second rule states that if the clock's time has gone over, then we can reset it. Rules are taken nondeterministically, and thus, semantically, our clock could tick to arbitrary time values before being reset.

For any sort, such as `Clock`, in our system module, we have the semantics of a transition system. A *transition system* is a pair $\mathcal{A} = (A, \rightarrow)$, where A is a set of states, and $\rightarrow \subseteq A \times A$ is a binary relation for transitions. In our `Clock` sort, our set A will be all possible values of clocks $\{c(0), c(1), c(2), ...\}$. Note that reducible terms, such as $c(1+1)$, are represented by a canonical element, $c(2)$. The transition relation would then have $a \rightarrow a'$ iff the term a can be rewritten into a' with one rewrite rule. Thus, in our example, $c(0) \rightarrow c(1)$, $c(4) \rightarrow c(5)$, $c(30) \rightarrow c(0)$, but $c(6) \not\rightarrow c(8)$.

Furthermore, given that we have our transition system, it is natural, to label our states with atomic propositions. For example, we have already defined the predicate `over?` on clocks, and we can thus label all states in our transition system with o whenever the predicate `over?` holds for clocks. Thus, given some set of atomic propositions, AP with $o \in AP$, we can define a labeling function $L : A \rightarrow \mathcal{P}(AP)$. In our clock example, we would have $o \in L(c(30))$, $o \in L(c(40))$, but $o \notin L(c(20))$. A transition system with this labelling function is called a *Kripke structure*.

2.3 Simulations and Stuttering Simulations

Given two transition systems (defined by Maude systems modules), it is natural to ask whether they have similar behavior. This brings us to the notion of simulations.

Definition. Given transition systems $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$, a simulation from $\mathcal{A}$ to $\mathcal{B}$ is a relation $H \subseteq A \times B$ such that if $a \rightarrow_{\mathcal{A}} a'$ and aHb then there is b' such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$.

If both H and H^{-1} are simulations, then we call H is a bisimulation.

A simulation from $\mathcal{A}$ to $\mathcal{B}$ means that $\mathcal{B}$ must exhibit all the transition behavior of $\mathcal{A}$ and possibly more. Any transition $\mathcal{A}$ can take, $\mathcal{B}$ should be able to "simulate."

In addition to similar transition behavior given by H it is also desirable to have H preserve important properties of states of $\mathcal{A}$ and $\mathcal{B}$. Thus, we can furthermore relate the Kripke structures of the two systems in order have simulation of transitions as well as a correspondence of labels on states.

Definition. Given Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$, both over the same set AP of atomic propositions, an AP-simulation from $\mathcal{A}$ to $\mathcal{B}$ is given by a simulation H from the transition systems of $\mathcal{A}$ to $\mathcal{B}$ such that if aHb, then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.

However, for the purposes of reasoning in our system modules, the notion of simulation is a bit too strict. Mostly it requires lockstep transitions between two transition systems. Most of the time we can relax these constraints of lock step behavior, since often we have intermediate transitions that don't really affect the important parts of the state of a system (or more precisely, the labelling of atomic propositions in a Kripke structure of interest). It is useful to define a more loose notion of simulation.

Definition. Given transition systems $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ and a relation $H \subseteq A \times B$. Given a path π in $\mathcal{A}$ and a path ρ in $\mathcal{B}$, we say that ρ H-matches π if there are strictly increasing functions $\alpha, \beta : \mathbb{N} \rightarrow \mathbb{N}$ with $\alpha(0) = \beta(0) = 0$ such that, for all $i, j, k \in \mathbb{N}$, if $\alpha(i) \leq j < \alpha(i+1)$ and $\beta(i) \leq k < \beta(i+1)$, it holds that $\pi(j) H \rho(k)$.

That is we divide pairs of paths in two transition systems into chunks, so that each chunk in one transition system should match a chunk in the other transition system and in progressive order.

Definition. Given transition systems $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$, a stuttering simulation from $\mathcal{A}$ to $\mathcal{B}$ is a relation $H \subseteq A \times B$ such that if aHb, then for each path π in $\mathcal{A}$ starting at a there is a path ρ starting at b that H-matches π.

If both H and H^{-1} are stuttering simulations, then we say that H is a stuttering bisimulation.

Intuitively, a stuttering simulation from $\mathcal{A}$ to $\mathcal{B}$ under the relation H says that for any path $\mathcal{A}$ can take, $\mathcal{B}$ can also take a similar path. However, the paths only need to be loosely related by chuncks, and lock step simulation is not required.

Of course, reasoning about all possible paths would be quite hard in practice, especially since rewrite rules only represent each individual step. It would be convenient to have an equivalent notion of stuttering simulation but based on each transition step instead of on infinite paths. This is conveniently provided by the notion of a well-founded simulation.

Definition. Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ be transition systems. A relation $H \subseteq A \times B$ is a well-founded simulation of transition systems from $\mathcal{A}$ to $\mathcal{B}$ if there exist functions $\mu : A \times B \rightarrow W$ and $\mu' : A \times A \times B \rightarrow \mathbb{N}$, with $(W, <)$ a well founded order, such that whenever aHb and $a \rightarrow a'$, either:

1. there is b' such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$, or
2. $a'Hb$ and $\mu(a', b) < \mu(a, b)$, or
3. there is b' such that $b \rightarrow_{\mathcal{B}} b'$, $aH'b'$, and $\mu'(a, a', b') < \mu'(a, a', b)$.

For purposes in this dissertation, we use the set $\mathbb{N}$ for W and furthermore, we also use the functions $\nu : B \times A \rightarrow W$ and $\nu' : B \times B \times A \rightarrow \mathbb{N}$ to show well founded simulation in the other direction from $\mathcal{B}$ to $\mathcal{A}$.

The equivalence between well-founded simulations and stuttering simulations are stated in a Theorem in [36].

Theorem 2.3.1. *Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ be Kripke structures over AP, and $H \subseteq A \times B$. Then, H is a well-founded AP-simulation iff it is a stuttering AP-simulation.*

It is worth noting that practically, we are interested in stuttering AP-bisimulations. However, it is unclear which set AP of atomic propositions that we should be reasoning over. However for all relations H we define in our stuttering bisimulation proofs, H will preserve: (1) all values of attributes in common objects, (2) the model time. Thus in our bisimulation proofs, all atomic propositions based on object attributes and current time are preserved by H, and this set of atomic propositiotns cover most properties of interest.

2.4 Full Maude and Real-Time Maude

We have covered the core of Maude with functional module and systems modules and given a brief overview of their semantics. We have also shown how these semantics can be used to reason about simulations. However, creating complex systems from just basic primitives may be difficult and hard to preserve modularity. We now discuss some higher level Maude constructs and the tools that exist to model real-time object-oriented actor systems.

Full Maude [19] 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 sorts `Object` or `Msg`. Objects are represented as records:

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

where *objectID* is the object's name of class *classID*, and each attribute *AttributeName* `j`; has a corresponding value *Value* `j`.

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 pacemaker object consumes a message to set the pacing period to T:

```
rl setPeriod(pm, T)
   < pm : Pacing-Module | pacing-period : PERIOD >
   => < pm : Pacing-Module | pacing-period : T > .
```

Real-time Maude [43] is a real-time extension of 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`. We have show a very small subset of this earlier as an illustration of equational modules. 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 defining 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* [43].

The *tick* operator advances time over a configuration by some duration of time. 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 gives 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, $mte(\texttt{stableSys}) = \texttt{INF}$.

2.5 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). Normally, theories are instantiated via *views* to other theories or to 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 [16] 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 (i.e. theory interpretations) 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 formal patterns, we will exploit in particular the Real-Time Maude parameterization mechanisms.

2.5.1 Actor Reflection in Rewriting Logic

The representation of objects in Maude allows for arbitrarily complex term types to be used as object attributes, including other objects or even entire actor configurations. This allows for simple specifications of meta-objects. Meta-objects are objects that encapsulate other objects and have the capability to control/mediate/adapt the behaviors of these encapsulated objects. Expressing patterns as meta-objects, this allows us to separate concerns and easily compose a solution by nesting meta-objects inside each other. In this dissertation, most of our patterns use onion-skin meta-objects (objects that wrap a single object) [3]. For example, a meta-object to filter out of range requests to an infusion pump can be specified with a rule:

```
crl setRate(pump, R)
   < pump : RateFilter | inner: < pump : Pump | ... > >
```

```
=> < pump : RateFilter | inner: < pump : Pump | ... > >
if R > max-rate .
```

Precise formal semantics have been defined for onion-skin meta-objects in rewriting logic [17], and also for a more general notion of meta-objects that may contain entire configurations [37].

2.6 Socket Programming in Maude

Maude supports the Berkley sockets API for TCP communication. This is done by having a special gateway object, denoted `<>`, to consume all the messages responsible for setting up sockets and communicating to an external environment (e.g. `createClientTcpSocket`, `send`, `receive`). The gateway object will also generate messages upon status updates from the socket (e.g. `sent`, `received`, `closedSocket`). Consuming and generating messages from the gateway object is achieved by external rewrite rules which can be executed using the `erew` command in Core Maude. An important thing worth pointing out about external rewrite rules is that *external rewrite rules are only applied when no internal rewrite rules can be applied.* Also, using external rewrite rules with Real-Time Maude specifications (built on top of Full Maude) requires reifying the specification down to a Core Maude module before executing it.

Chapter 3

Patterns for Fault-Tolerant Device Interfaces

Fault tolerance has been a major area of study in embedded systems since their creation. General notions of faults cover everything from anomalies of the environment to misbehavior of internal system components due to insufficient measurements or degradation of hardware over time.

For our inital work in describing patterns for fault tolerant design, we exclusively focus on external faults (e.g. input noise, interference, message loss, etc.). The reason for this is that external faults are much more easy to model, since they can be seen as variations of the input. While internal system faults (e.g. bit flips, memory corruption, computation error, etc.) are also interesting, they are much more difficult to handle completely, and usually, we must settle for probabilistic guarantees or make probabilistic assumptions of the fault model. We believe that at the current time, internal system faults are usually better addressed by extensive testing and probabilistic analysis.

In terms of solving the problems of external faults. We take a 4 step approach to describing solution patterns for fault-tolerance or sometimes even full fault masking.

1. As with any application of formal methods, the domain model that we are using must be extensively studied through use cases and case studies and encoded in a model.

2. The domain model is separated into the external environment, E that the system receives input from and outputs actuation to.

3. The fault model is described as a relation of ideal environmental inputs to potential faulty behaviors of the environment. More precisely, a relation $F \subseteq E \times E$ s.t. if eFe' then e' is a possible faulty environmental model of e.

4. The desired behavior of a system can be described as an ideal design s s.t. the execution of the environment-system pair (e, s) generates ideal behavior.

5. A fault tolerance technique is then a transformation $T : Sys \to Sys$ s.t. the following property holds. Given a fault-tolerance characterization H. For all possible models of e, for all e' s.t eFe', and for all system designs s, then the execution of the system $(e', T(s))$ H-corresponds to (e, s).

In this section, we use this idea to describe 3 different fault tolerant patterns for buttons. The external environment in this case will be the buttons and their associated behavior. The fault relations will be various common button related faults such as button bounce, button stuck, or accidental presses. The fault tolerance techniques will then be common design practices for handling button related faults such as button debouncing.

3.1 Modeling Buttons

We first describe the model of the environment in detail. That is the formal model of a button. Our goal in modeling a system is to have some reasonable abstraction of the real world. For many device interfaces, buttons are used to communicate useful and important information to a system. Thus, it makes sense to look at how a button works in an abstract but general sense. Furthermore, we will also be able to use this abstraction to model faulty button behavior.

We model a button as something that can be in one of two states, either *pressed* or *not pressed*, at any instant in time. Thus, we can model the behavior of a button as a function $button_{state} : Time \to \{0, 1\}$, where $b(t) = 0$ means the button is not pressed, and $b(t) = 1$ means the button is pressed. $Time$ is a totally ordered set. $Time$ can represent 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 $\mathbb{N}$. In later definitions and theorems, we let $Time = \mathbb{R}_{\geq 0}$ whenever possible, since this provides the most general definitions and results.

Although this model captures everything we want about a button, we would like to have an equivalent model that can be discretized. In order to do this, we need to make some assumptions. When a button is held down, it is natural to talk about the time when it was initially pressed, and also the time when it was released. Thus, we define $init(b, t) = inf(\{t' \in Time | \forall t'' \in [t', t], b(t'') = b(t)\})$ and $end(b, t) = sup(\{t' \in Time | \forall t'' \in [t, t'], b(t'') = b(t)\})$. We make the following assumptions:

1. $Time$ starts from 0, $init(b, t)$ is always defined, and $b(init(b, t)) = b(t)$.
2. A button is always pressed or released for at least a minimal finite duration T_{min}, i.e. for all t, $end(b, t) - init(b, t) \geq T_{min}$.

Let I_{valid} be the subset of button functions $b \in [Time \to \{0, 1\}]$ that satisfy these assumptions.

These assumptions allow us to model continuous button behavior as a discrete timed model, since in each finite interval of time, given a buttion function, b, there is only a finite number of press and release

events in b. For example, if the button behavior is $b(t) = 1$ for $t \in [0, 1) \cup [2, 5)$ and $b(t) = 0$ 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., $(press, 0).(release, 1).(press, 2).(release, 5)$. This model is captured in the following Maude model:

```
(fmod PRESS-RELEASE is inc LTIME-INF .
```

The first structural property of a valid button model must have alternating press and release events. We enforce this in button press models by using the powerful typing system provided by Sorts in Maude.

```
...
sorts Press Release .
sort PressReleaseList .
sorts MtList PressLastList ReleaseLastList .
subsort MtList < PressLastList ReleaseLastList .
subsorts PressLastList < PressReleaseList .
subsorts ReleaseLastList < PressReleaseList .
...
```

With the sorts defined, we can define the actual structure of what we want to model for a button. Press and release events are all associated with a time stamp, and concatenating press and release events in alternating order (enforced by the sorts) will generate valid button press models.

```
op press : Time -> Press .
op release : Time -> Release .
op nil : -> MtList .
op __ : PressLastList Release -> ReleaseLastList .
op __ : ReleaseLastList Press -> PressLastList .
...
```

Finally, even with sort constraints for button press models. We may want some further constraints on button press models we consider as valid user input. For example, the assumption that button presses are spaced far enough apart. However, we will not make these constraints part of the type of button press models, as faults can easily create a button press model that is not a valid user input.

```
   op valid? : PressReleaseList -> Bool .
   eq valid?(nil) = true .
   eq valid?(L press(T)) = valid?(L) and (t-last(L) lt T) .
   eq valid?(L release(T)) = valid?(L) and (T monus t-last(L)) geq T-min .
endfm)
```

To summarize, we have a list of press and release events over time. The press and release events must alternate, and their time stamps must also be strictly increasing in order for it to be a valid model. The constant T_{min} spaces out the events to avoid Zeno-like behavior in valid non-faulty button press models.

3.2 Modeling the System

The formal model of our timed system can be found in Appendix A. The model starts by defining the notion of time advancement (`TICK-MTE-SEM`), primitive real-time components (`RT-COMP`), and timed messages (`DELAY-MSG`).

The behavior of a button we have defined before 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 just simply convert the list of press and release events over time into a set of delayed messages:

```
(tomod PRESS-RELEASE-MSGS is inc PRESS-RELEASE .
   inc DELAY-MSG .

   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)) .
endtom)
```

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.

3.3 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.

3.3.1 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 errorneous button press events of signals to be generated. Since oscillatory phenomena are usually dampened pretty quickly, there is a short maximum time window, T^{max}_{bounce}, within which a button may bounce after it is pressed.

We can formally define a bounce fault to be a binary relation F_{bounce} on our button input model, where, intuitively, we use b_f as the faulty button function, which may exhibit bouncing behavior, and b is as *ideal* button function not subject to any faults.

A *button bounce fault* is a relation $F_{bounce} \subseteq I_{valid} \times I_{valid}$ (implicitly parameterized by a maximum bounce time T^{max}_{bounce}) where $(b, b_f) \in F_{bounce}$ iff:

1. $b(t) = 1 \implies b_f(init(b, t)) = 1$ (all initial button press events are preserved), and
2. if $b_f(t) = 1$, then $init(b_f, t) - init(b, t) \leq T^{max}_{bounce}$ (all bouncing behavior occurs within T^{max}_{bounce} time of a button press).
3. if $b(t) = 1$ and $t - init(b, t) \geq T^{max}_{bounce}$ then $b_f(t) = 1$ (all bouncing behavior stabilizes after T^{max}_{bounce} time).

Of course, this is just defined based on the continuous model of time. However, since the definition is based on the use of the *init* function, we can easily use F_{bounce}, represented as the binary predicate `bounce-fault`, to define the same semantics for bounce faults in the list-like representation of button functions. Note that the arguments of `bounce-fault` has the first argument as the faulty input and the second argument as the non-faulty input (this reverses the arguments for the relation F_{bounce} but everything else about the relation is the same).

```
op bounce-duration : -> Time .
```

```
op space-duration : -> Time .
op bounce-fault : Input Input -> Bool .
eq bounce-fault(nil,nil) = true .
eq bounce-fault(I press(T), I' press(T)) = bounce-fault(I,I') .
ceq bounce-fault(I press(T), I' press(T')) = bounce-fault(I, I' press(T'))
   if T le (T' plus bounce-duration) /\ T gt T' .
eq bounce-fault(I release(T),I') = bounce-fault(I,I') .
eq bounce-fault(I, I' release(T)) = bounce-fault(I,I') .
eq bounce-fault(I,I') = false [owise] .
```

The equations describe the F_{bounce} relation as the operator `bounce-fault`. Here, the first argument to `bounce-fault` is a possible faulty button input model of the second argument (an ideal button input model). This mostly amounts to checking that the faulty press messages are sufficiently close in time to nonfaulty presses. Furthermore, we ignore the precise time for button releases. The important equations are those that capture the requirements on the spacing between press events. Either they are the same press event, or a press event is within T^{max}_{bounce} of the nonfaulty one.

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.

3.3.2 Button Bounce Fault Generation

As mentioned above, our next step is to generate a set of faulty inputs for each set of nonfaulty inputs. 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 natural number 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.

The intuition is to create a sliding window for the faults. All input events after the window are assumed to be finalized, and all inputs inside the window can still have faulty behavior. Each button press event in the window can generate bounced button presses within the T^{max}_{bounce} time intervals. Furthermore, the length and spacing between bouncing presses can also vary and are captured by further rewrite rules. It can be

shown that these rewrite rules will always terminate, since the time window decreases for each faulty button press input generated, and, furthermore, each faulty button press can only decrease in duration for each rewrite rule (and this must terminate for a natural number model of time).

The sliding window is captured by a 3 tuple: the input events before the sliding front and within the window, the time of the sliding window front, the input event after the sliding window. Initially the window covers all the events and is slowly shrunk. The equations capture when events are moved outside the window because the sliding time has already moved beyond them. The multiple equations are mostly to distinguish different events. The faulty and nonfaulty events need to be distinguished (e.g. `pressf` and `releasef`) as to not generate more faults on top of faults.

```
(mod BOUNCE-FAULT is
   inc BUTTON-FAULTS .
   pr NAT-TIME-DOMAIN .

   op bounce-fault : Input Nat Input ~> Input .

   op pressf : Time -> Press .
   op releasef : Time -> Release .

   vars I I' : Input .
   vars T T' T'' : Time .

   eq bounce-fault(nil, T, I) = I .
   ceq bounce-fault(I press(T), T', I')
      = bounce-fault(I, T' monus 1, press(T) -> I')
      if T' <= T .
   ceq bounce-fault(I pressf(T), T', I')
      = bounce-fault(I, T' monus 1, press(T) -> I')
      if T' < T .
   eq bounce-fault(I release(T), T', I')
      = bounce-fault(I, T monus 1, release(T) -> I') .
   eq bounce-fault(I releasef(T), T', I')
      = bounce-fault(I, T monus 1, release(T) -> I') .
```

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

The actual sliding behavior of the window is defined by rules. These rules describe, respectively, how to shift the window by one step, generate a faulty button press event with maximum duration, increase the starting time of a bounced press, and decrease the ending time of a bounced press. All of these rules together allow us to generate all possible bounced button press events at all times and of all durations.

```
   rl bounce-fault(I, T, I') => bounce-fault(I, T monus 1, I') .
   crl bounce-fault(I press(T), T', I')
      => bounce-fault(I press(T) releasef(T + 1) pressf(T'), T', I')
      if T' > T + 1 /\ T' <= T + bounce-duration .
   crl bounce-fault(I releasef(T) pressf(T'), T'', I')
      => bounce-fault(I releasef(T + 1) pressf(T'), T'', I')
      if T' > T + 1 .
   crl bounce-fault(I releasef(T) pressf(T'), T'', I')
      => bounce-fault(I releasef(T) pressf(T' monus 1), T'', I')
      if T' > T + 1 .
endm)
```

3.3.3 A Button Debouncer Pattern

Finally, we come to the most important part of all of our specification, namely, a pattern for correctly handling faulty button bounce behavior. Figure 3.1 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 first must describe the input theory 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.

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

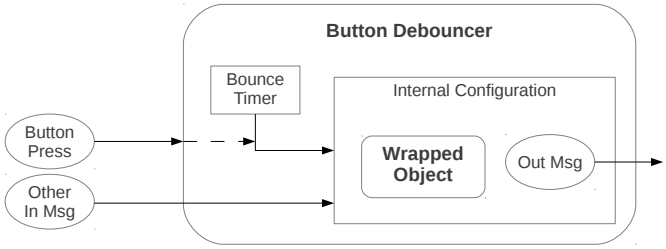

Figure 3.1: The Button Debouncer Pattern

```
   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 .

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

The parameters of the theory `DEBOUNCED` can be intuitively thought as follows. The class *Wrapped* is 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. The last equation in the theory `DEBOUNCED` says that time should not be allowed to advance when a press message has not yet

been handled.

Now, the actual pattern itself is quite straight forward. The debouncer pattern is a wrapper around 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, we use parameter `|O|` as an instance 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)
```

3.3.4 Proof of Correctness of the Debouncer Pattern

The button debouncer should essentially mitigate button bounce faults, but we must make precise 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. These two projection operators are defined in Maude as follows:

```
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. 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}.

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

Definition. Consider a system with a wrapped object, which in the most general case is a system configuration of the form

$S = \{\ C$ `<` O `:` `Wrapper | inside :` C' `, timer :` TM `>` $\}$ `in time` T.

We define the following functions:

- $nmsgs_{to}(S)$ is the number of messages in configuration C being sent to oid O.
- $nmsgs_{from}(S)$ is the number of messages in configuration C' which is being sent to an oid other than O.
- χ_{timer} is 1 when the timer TM is set, and 0 when TM is just `no-timer`.

Theorem 3.3.1. *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.*

Proof. By Theorem 2.3.1, we have that H is a well-founded bisimulation if we can find relations μ and μ' in both directions. Furthermore, since we are only considering natural number time, if $mte > 0$, then we *tick* the system by 1 time unit as an atomic transition.

First, we show that a well-founded simulation from S_{ideal} to $S_{wrapped}$ exists.

Define

$\mu(s_i, s_f) = 0$ and

$\mu'(s_i, s_i', s_f) = 2(nmsgs_{to}(s_f) + nmsgs_{from}(s_f)) + \chi_{timer}(s_f)$.

Suppose that $s_i H s_f$. We consider possible cases for the transition $s_i \to s_i'$. For convenience, we denote the object ID of the internal wrapped object to be O_w.

(1) $mte(s_i) = 0$ because a message M needs to be delivered in s_i, we consider the following cases:

(1.a) M is not a message to O_w, and M is in the external configuration of s_f, then $s_f \to s_f'$ takes a corresponding transition by using the same rule, and we have $s_i' H s_f'$.

(1.b) M is a message to O_w, and M is in the internal wrapped configuration of s_f, then, again, we can just take a corresponding transition $s_f \to s'_f$ in the wrapped configuration using the same rule, and we have $s'_i H s'_f$.

(1.c) M is a message to O_w, and M is in the external configuration of s_f, and M is not a press message, then the `[forward-in]` rule is used to transition $s_f \to s'_f$, and we have $s_i H s'_f$, and $\mu(s_i, s'_i, s'_f) < \mu(s_i, s'_i, s_f)$ since $nmsgs_{to}$ decreases.

(1.d) M is a message not to O_w but in the internal configuration of s_f. Then the `[forward-out]` rule is used to transition $s_f \to s'_f$, we have $s_i H s'_f$ and $\mu(s_i, s'_i, s'_f) < \mu(s_i, s'_i, s_f)$ since $nmsgs_{from}$ decreases.

(1.e) M is a message to O_w in the external configuration of s_f, and M is a press message. Since $s_i H s_f$, this means that the timer is not set, and the rule `[set-timer]` is used to transition $s_f \to s'_f$. We have $s_i H s'_f$, and $\mu(s_i, s'_i, s'_f) < \mu(s_i, s'_i, s_f)$ since $2nmsgs_{to} + \chi_{timer}$ decreases.

(2) if $mte(s_i) > 0$, then $s_i \to s'_i$ is a tick rule advancing time by one time unit, and we have a few subcases to consider for possible transitions $s_f \to s'_f$

(2.a) $mte(s_f) > 0$, we have a corresponding tick rule $s_f \to s'_f$ by one time that preserves the relation H.

(2.b) $mte(s_f) = 0$ because the timer in the wrapper has expired. In this case, the only rule to take is to stop the timer, so $s_f \to s'_f$ only changes the timer from `t(0)` to `no-timer`. We have $s_i H s'_f$ and $\mu(s_i, s'_i, s'_f) < \mu(s_i, s'_i, s_f)$, since the value of χ_{timer} decreases and all other values stay the same.

(2.c) $mte(s_f) = 0$ because a message needs to be delivered in s_f. Since $s_i H s_f$, and $mte(s_i) > 0$, the message in s_f must have been a button press message, and the wrapper timer must be set to some positive time. This means, that only the rule `[ignore-press]` can be used. Thus, in the transition $s_f \to s'_f$, we still have $s_i H s'_f$, but $\mu(s_i, s'_i, s'_f) < \mu(s_i, s'_i, s_f)$, since one message to O_w is removed and $nmsgs_{to}$ decreases.

Now, we show the other direction, namely, that a well-founded simulation exists from S_{ideal} to $S_{wrapped}$.

Define

$\nu(s_f, s_i) = 2(nmsgs_{to}(s_f) + nmsgs_{from}(s_f)) + \chi_{timer}(s_f)$ and

$\nu(s_f, s'_f, s_i) = 0$.

Let $H' = H^{-1}$, and suppose $s_f H' s_i$. We consider the following cases:

(1') $mte(s_f) > 0$, and $s_f \to s'_f$ is a tick rule advancing time by one time unit. By the homomorphic definition of mte, if $s_f H' s_i$, then $mte(s_i) > 0$, and a corresponding one time unit tick step can be taken $s_i \to s'_i$ with $s'_f H' s'_i$.

(2') $s_f \to s'_f$ uses the `[forward-in]` rule. In this case, $s'_f H' s_i$, and $\nu(s_f, s_i) > \nu(s'_f, s_i)$, since $nmsgs_{to}$ decreases.

(3') $s_f \to s'_f$ uses the `[forward-out]` rule. Again, we have $s'_f H' s_i$, and ν decreases, since $nmsgs_{from}$

decreases.

(4') $s_f \to s'_f$ uses the `[ignore]` rule. Since $s_f H' s_i$, we know that the button press message that was ignored was not originally in s_i, and $s'_f H' s_i$, and ν decreases, since $nmsgs_{to}$ decreases.

(5') $s_f \to s'_f$ uses the `[set-timer]` rule. Since $s_f H' s_i$, we know that the press message forwarded also exists in s_i, and again, $s'_f H' s_i$, and ν decreases, since $2nmsgs_{to} + \chi_{timer}$ decreases.

(6') $s_f \to s'_f$ uses the `[reset-timer]` rule. This does not change anything with the projected configuration, and $s'_f H' s_i$ and ν decreases, since χ_{timer} decreases.

(7') $s_f \to s'_f$ has $mte(s_f) = 0$, and it transitions by a rule not in the debouncing pattern module. We assume that all zero-time rules not in the debouncing module are about consuming messages by objects, and additionally that the debouncer wrapper object does not appear on either side of these rules. Since $s_f H' s_i$, we can take a corresponding transition $s_i \to s'_i$ using the same rule with $s'_f H' s'_i$.

This shows that H and H^{-1} are well-founded simulations, and therefore that we have a bisimulation between the two systems, as desired. □

Note that if we don't 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 $\pi_{nf}(\pi_w((s_f)))$.

3.3.5 Model Checking Case Study for the Buttion Debouncer

We have proved that there is a bisimulation between an ideal system and our fault-tolerant system. We would like to model check this property for a specific instantiation of the pattern and for a specific subset of execution paths. The model contains the following parts:

- a simple button counter that counts the number of times the button is pressed,
- the debounced version of the button counter obtained by instantiating our debouncer pattern with the simple button counter as the wrapped object,
- a system that has parallel execution over time of two different models,
- predicates to relate the simple button counter to the debounced button counter

The model-checking should then check that, given equivalent initial states, the simple button counter with no faults and the debounced button counter with bounce faults executed in parallel should satisfy the same predicates over time. Here, the predicate that we are most interested in would be that the value of the counter is the same for both systems.

For our case study, we use a simple counter that counts the number of times a button has been pressed. To make the problem more interesting, we also count the number of times the button has been released.

```
(tomod DUAL-COUNTER is
   ...
   class Counter | press-count : Nat, release-count : Nat .

   var O : Oid .
   var N : Nat .
   var T : Time .

   op dest : Msg -> Oid .
   eq dest(press(O)) = O .
   eq dest(release(O)) = O .

   eq tick(< O : Counter | >, T)
      = < O : Counter | > .
   eq mte(< O : Counter | >)
      = INF .

   rl press(O) < O : Counter | press-count : N >
      => < O : Counter | press-count : s N > .
   rl release(O) < O : Counter | release-count : N >
      => < O : Counter | release-count : s N > .

   op counter : -> Oid .
   op init-counter : -> Configuration .
   eq init-counter = < counter : Counter | press-count : 0, release-count : 0 > .
```

```
   op get-press : Object ~> Nat .
   eq get-press(< O : Counter | press-count : N >) = N .
endtom)
```

In order to apply the pattern to this problem, we must set the appropriate parameters. The view from the theory of a wrapped object for debouncing and the dual counter example is as follows:

```
(view Counter from DEBOUNCED to DUAL-COUNTER is
   class |Wrapped| to Counter .
   op |dest| to dest .
   op |t-bounce| to bounce-duration .
   op |t-space| to space-duration .
   msg |press| to press .
endv)
```

With this view, we can instatiate the pattern, and define an initial state:

```
(tomod PRESS-DEBOUNCER is
   pr DEBOUNCER{Counter}*(class !Debouncer{Counter} to PressDebouncer) .

   op init-press-debounce-counter : -> Configuration .
   eq init-press-debounce-counter =
      < counter : PressDebouncer | inside : init-counter, timer : init-timer > .
endtom)
```

3.3.6 Model Checking Constructs

We now must consider the question of what exactly is the safety property that we are trying to verify. In this case, we want to show that the pattern provides an ideal abstraction of system execution while the system is running. To this end, we are really verifying a *correspondence* between two systems. An ideal system, and the nonideal system with our pattern applied. The easiest way to compare two system is of course to execute them simultaneously and define an equivalence relation that should hold on the two states as a safety invariant. We first must define what we mean by a simultaneous system, and also specify in Real-Time Maude how to execute it.

```
(tomod MATCHING-EXEC is
```

```
   pr TICK-MTE-SEM .

   sort ConfPair .
   subsort ConfPair < Configuration .
   op _;;;_ : Configuration Configuration -> ConfPair .
   op tick : ConfPair Time -> ConfPair .
   op mte : ConfPair -> Time .

   vars C C' : Configuration .
   var T : Time .
   eq tick(C ;;; C', T) = tick(C, T) ;;; tick(C', T) .
   eq mte(C ;;; C') = minimum(mte(C), mte(C')) .
endtom)
```

Now, in order to verify by model-checking that, regardless of faults, we always have a correct correspondence between the ideal system and the faulty one wrapped by the pattern, we put everything together, the input model, the ideal press counter, the fault input model, and the debounce pattern:

```
(tomod MATCHING-EXEC-TEST is
   inc MATCHING-EXEC .
   inc DUAL-COUNTER-TEST .
   inc BOUNCE-TEST .
endtom)
```

We then model check that the number of button presses captured by both the ideal system and the pattern based system is the same:

```
(tsearch [1] in MATCHING-EXEC-TEST :
{ (to-msgs(test-input,counter) init-counter) ;;;
(to-msgs(bounce-fault(test-input,100,nil),counter) init-press-debounce-counter) }
=>*
{ (C:Configuration O:Object) ;;; (C':Configuration O':Object) }
such that
mte(C:Configuration) > 0 /\
mte(C':Configuration) > 0 /\
```

```
mte(get-inside(O':Object)) > 0 /\
get-press(O:Object) =/= get-press(get-inside(O':Object))
in time <= 100 .)

Timed search [1] in MATCHING-EXEC-TEST
   {(to-msgs(test-input,counter)init-counter);;; to-msgs(bounce-fault(
    test-input,100,nil),counter)init-press-debounce-counter} =>* {(
    C:Configuration O:Object);;; C':Configuration O':Object}
in time <= 100 and with mode maximal time increase with default 1 :


No solution
```

We indeed see that no counterexamples were found, and that the two systems always have the same projected state for every instant in time. Note that the condition of checking `mte` larger than zero is necessary, since we only care about the states when they are stabilized, or just before time advances. We don't want counterexamples where intermediate values of zero-time rewrite rules cause because the two states are transiently different.

3.4 Pattern to Address Phantom Faults

3.4.1 Phantom Faults

Slight disturbances in the environment (e.g. EMI, moving parts, etc.) leads 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 a button input 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 that is, 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 $T_{phantom}$)

```
op phantom-thresh : -> Time .
```

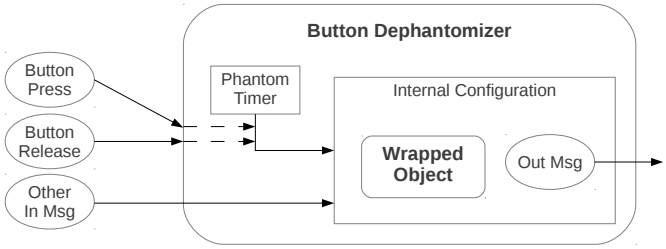

Figure 3.2: The Dephantomizer Pattern

```
op phantom-fault : Input Input -> Bool .
eq phantom-fault(nil, nil) = true .
eq phantom-fault(I press(T), I' press(T))
   = phantom-fault(I,I') .
eq phantom-fault(I press(T) release(T'), I' press(T) release(T'))
   = phantom-fault(I,I') .
ceq phantom-fault(I press(T) release(T'), I' press(R) release(R'))
   = phantom-fault(I,I' press(R) release(R'))
   if T gt R' /\ T' lt (T plus phantom-thresh) .
eq phantom-fault(I,I') = false [owise] .
```

We have the constant `phantom-thresh`, which defines the minimal time that a button must be held down in order to be considered a valid button press. It is assumed that intentional button presses will always last longer than this duration. The rest of the equations match press and release events and ignore ones that have too short duration (time spacing between them).

3.4.2 Dephantom Pattern

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

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 defines the wrapper class and the time advancement semantics. This is exactly the same as 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)
```

3.4.3 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.

We first start by defining the delay operator.

```
(tomod DELAY-MSGS is
   inc PHANTOM-COUNTER .

   op delay-press : Configuration Time -> Configuration .

   var CC CC' : Configuration .
   vars C C' : NEConfiguration .
   vars T T' : Time .
   eq delay-press(C C', T) = delay-press(C, T) delay-press(C', T) .
   eq delay-press(delay(press(counter), t(T)), T')
      = delay(press(counter), t(T plus T')) .
   eq delay-press(press(counter), T') = delay(press(counter), t(T')) .
   eq delay-press(CC, T) = CC [owise] .
```

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.

```
   op remove-small : Configuration -> Configuration .

   ceq
```

```
remove-small(
   delay(press(counter), t(T)) delay(release(counter), t(T'))
   CC)
= remove-small(CC)
if (T' monus T) le phantom-thresh .

ceq
remove-small(CC
   delay(release(counter), t(T))
   < O : Dephantom |
      inside : CC',
      timer : t(T') >
   )
= remove-small(CC
   <  O : Dephantom |
      timer : no-timer >)
if T le T'  .

eq    remove-small(CC) = CC [owise] .
```

The next projection operation is `remove-small` which removes all press and release events which are shorter than the phantom threshold duration. Removing messages in the outer configuration is trivial enough. The subtle part comes when a press has been consumed by the wrapper but the release is still pending. In this case, we need to use the value of the timer in the dephantom wrapper to decide if the message duration was too short.

```
op remove-wrapper : Configuration -> Configuration .
eq remove-wrapper(CC
   < O : Dephantom |
      inside : CC',
      timer : t(T) >
   )
= CC CC' delay(press(counter), t(T)) .
```

```
   eq remove-wrapper(CC
      < O : Dephantom |
         inside : CC' >
      )
   = CC CC' [owise] .
endtom)
```

Finally the last projection relation we need is the `remove-wrapper` operation. This removes the wrapper by merging the external and internal configurations, and furthermore, uses the value of the timer to decide if an additional press message needs to be generated in the projected configuration.

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`.

Again, we use the same definitions as with the button bounce case:

Definition. States of the transition system S_{ideal} are system configurations with a single instance of a *wrapped* (`Dephantom`) 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 $F_{phantom}$.

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 $time(s_f) = time(s_i)$.

Definition. Consider a system with a wrapped object, which in the most general case is a soup of the form $S = \{\ C\ \texttt{<}\ O\ \texttt{:}\ \ \texttt{Wrapper | inside :}\ \ C'\ \texttt{, timer :}\ \ TM\ \texttt{>}\ \}\ \texttt{in time}\ T$.

We define the following functions:

- $nmsgs_{to}(S)$ computes the number of messages in configuration C being sent to oid O.
- $nmsgs_{from}(S)$ computes the number of messages in configuration C' which is being sent to an oid other than O.
- χ_{timer} is 1 when the timer TM is set, and 0 when TM is `no-timer`.

Theorem 3.4.1. *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.*

Proof. First, we show that a well-founded simulation from S_{ideal} to $S_{wrapped}$ exists.

Define $\mu(s_i, s_f) = 0$ and $\mu'(s_i, s_i', s_f) = 2(nmsgs_{to}(s_f) + nmsgs_{from}(s_f)) + \chi_{timer}(s_f)$.

Suppose that $s_i H s_f$. We consider possible cases for the transition $s_i \to s_i'$. For convenience, we denote the object ID of the internal wrapped object to be O_w.

(1) $mte(s_i) = 0$ because a message M needs to be delivered in s_i, we consider the following cases:

(1.a) M is not a message to O_w, and M is in the external configuration of s_f, then $s_f \to s_f'$ takes a corresponding transition by using the same rule, and we have $s_i' H s_f'$.

(1.b) M is a message to O_w, and M is in the internal wrapped configuration of s_f, then again we can just take a corresponding transition $s_f \to s_f'$ in the wrapped configuration using the same rule, and we have $s_i' H s_f'$.

(1.c) M is a message to O_w, and M is in the external configuration of s_f, and M is not a press message, then the `[forward-in]` rule is used to transition $s_f \to s_f'$, and we have $s_i H s_f'$, and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since $nmsgs_{to}$ decreases.

(1.d) M is a message not to O_w but in the internal configuration of s_f. The the `[forward-out]` rule is used to transition $s_f \to s_f'$, we have $s_i H s_f'$ and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since $nmsgs_{from}$ decreases.

(1.e) M is a message to O_w in the external configuration of s_f, and M is a press message. Since $s_i H s_f$, this means that the timer is not set, and the rule `[set-timer]` is used to transition $s_f \to s_f'$. We have $s_i H s_f'$, and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since $2nmsgs_{to} + \chi_{timer}$ decreases.

(1.f) In the case when M is a release message, since we already concluded that the timer is not set, then the `[non-phantom-release]` rule is used, and it will be similar to the `[forward-in]` case.

(2) if $mte(s_i) > 0$, then $s_i \to s_i'$ is a tick rule by one time unit, we have a few subcases to consider for possible transitions $s_f \to s_f'$

(2.a) $mte(s_f) > 0$, we have a corresponding tick rule $s_f \to s_f'$ by one time that preserves the relation H.

(2.b) $mte(s_f) = 0$ because the timer in the wrapper is expired, in this case, the only rule to take is to stop the timer, so $s_f \to s_f'$ only changes the timer from `t(0)` to `no-timer`. We have $s_i H s_f'$ and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since the value of χ_{timer} decreases and all other values stay the same.

(2.c) $mte(s_f) = 0$ because a message needs to be delivered in s_f. Since $s_i H s_f$, and $mte(s_i) > 0$, the message in s_f must have been a phantom press or release message. If we have a phantom press message, then the `[set-timer]` rule is used and $2nmsgs_{to} + \chi_{timer}$ decreases. If we have a phantom release message, then the wrapper timer must be set to some positive time. This means that the rule `[phantom-release]` can be used. Thus, in the transition $s_f \to s_f'$, we still have $s_i H s_f'$, but $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$, since one message to O_w is removed, and $nmsgs_{to}$ decreases.

Now, we show the other direction, i.e., that a well-founded simulation from S_{ideal} to $S_{wrapped}$ exists.

Define $\nu(s_f, s_i) = 2(nmsgs_{to}(s_f) + nmsgs_{from}(s_f)) + \chi_{timer}(s_f)$ and $\nu(s_f, s_f', s_i) = 0$. Let $H' = H^{-1}$,

and suppose $s_f H' s_i$. We consider the following cases:

(1') $mte(s_f) > 0$, and $s_f \to s'_f$ is a tick rule by 1 time unit. By the homomorphic definition of mte, if $s_f H' s_i$, then $mte(s_i) > 0$, and a corresponding 1 time unit tick step can be taken $s_i \to s'_i$ with $s'_f H' s'_i$.

(2') $s_f \to s'_f$ uses the `[forward-in]` rule. In this case, $s'_f H' s_i$, and $\nu(s_f, s_i) > \nu(s'_f, s_i)$, since $nmsgs_{to}$ decreases.

(3') $s_f \to s'_f$ uses the `[forward-out]` rule. Again, we have $s'_f H' s_i$, and ν decreases, since $nmsgs_{from}$ decreases.

(4') $s_f \to s'_f$ uses the `[set-timer]` rule. Since $s_f H' s_i$, we know that the press message forwarded also exists in s_i, and again, $s'_f H' s_i$, and ν decreases, since $2nmsgs_{to} + \chi_{timer}$ decreases.

(5') $s_f \to s'_f$ uses the `[reset-timer]` rule. This actually creates a new press message in the internal configuration, but this is exactly the message that should have been delayed in the ideal configuration, and $s'_f H' s_i$ and ν decreases, since χ_{timer} decreases (not adding a new message to the internal configuration with destination being the internal wrapped object does not increase the value of $nmsgs_{to}$ or $nmsgs_{from}$.

(6') $s_f \to s'_f$ uses the `[phantom-release]` rule. Since $s_f H' s_i$, we know that the button press message that was ignored was not originally in s_i, and $s'_f H' s_i$, and ν decreases, since $nmsgs_{to}$ decreases.

(7') $s_f \to s'_f$ uses the `[nonphantom-release]` rule. Since $s_f H' s_i$, we know that this release is a valid release message, and it is treated similar to the `[forward-out]` rule. We have $s'_f H' s_i$, and ν decreases, since $nmsgs_{to}$ decreases.

(8') $s_f \to s'_f$ has $mte(s_f) = 0$, and it transitions by a rule not in the pattern module. We assume that all zero-time rules not in the pattern module are about consuming messages by objects, and additionally the wrapper object is not on either side of these rules. Since $s_f H' s_i$, we can take a corresponding transition $s_i \to s'_i$ using the same rule with $s'_f H' s'_i$.

This shows that H and H^{-1} are well-founded simulations and, therefore, we have a bisimulation between the two systems as desired.

□

3.5 Pattern to Address Stuck Faults

3.5.1 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 exactly the same as 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).

We can again describe this declaratively in Maude:

```
op stuck-duration : -> Time .
op stuck-fault : Input Input -> Bool .
eq stuck-fault(nil,nil) = true .
eq stuck-fault(I release(T), I') = same-input(I release(T), I') .
eq stuck-fault(I press(T), I' release(T')) = stuck-fault(I press(T), I') .
ceq stuck-fault(I, I') = same-input(I,I') or stuck-fault(I,I'')
   if I'' release(T) press(T') := I' .
eq stuck-fault(I, I') = false [owise] .
```

To simplify things, this describes a model where buttons are permanently stuck if they become stuck. This means that the equations just amounts to checking that the faulty input sequence is a prefix of the normal input up to the stuck press event. The `stuck-duration` is a necessary constant to define the minimal amount of time before a button is considered stuck. It is assumed that normal use of the button will not have the button being held beyond stuck duration.

3.5.2 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 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. 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.

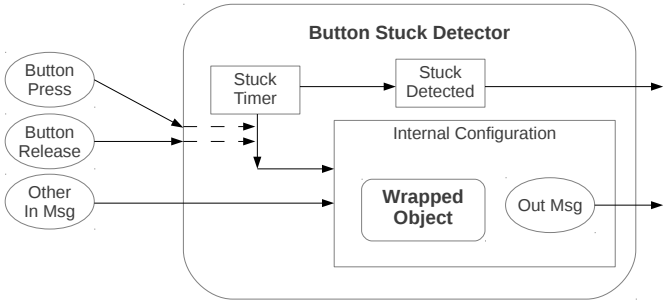

Figure 3.3: The Stuck Detection Pattern

```
(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. 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, which 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 the other patterns 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)
```

3.5.3 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 a bisimulation. 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.

For the case with absence of stuck faults, we again show correspondence with a projection relation from a wrapper system:

```
op remove-wrapper : Configuration -> Configuration .
eq remove-wrapper(CC
   < O : StuckDetector |
      inside : CC' >
   )
= CC CC' .
endtom)
```

The projection π_{stuck} from a wrapped system for stuck detection to an ideal input system with no wrapper is just simply the function `remove-wrapper`.

Again, we use the same definitions as with the button bounce case:

Definition. States of the transition system S_{ideal} are system configurations with a single instance of a *wrapped* (`StuckDetector`) object, and such that the input button press messages are spaced by at least the assumed minimal time spacing, and the input button press durations (time between consecutive press and release events) are always at most T_{stuck} time units apart. 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_{stuck}.

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 $time(s_f) = time(s_i)$.

Definition. Consider a system with a wrapped object, which in the most general case is a soup of the form $S = \{\ C$ `<` O `:  Wrapper | inside :  ` C' `>` $\}$ `in time` T.

We define the following functions:

- $nmsgs_{to}(S)$ computes the number of messages in configuration C being sent to oid O.
- $nmsgs_{from}(S)$ computes the number of messages in configuration C' which is being sent to an oid other than O.

- χ_{timer} is 1 when the timer TM is set, and 0 when TM is just `no-timer`.

Theorem 3.5.1. *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.*

Proof. First, we show that a well-founded simulation from S_{ideal} to $S_{wrapped}$ exists.

Define $\mu(s_i, s_f) = 0$ and $\mu'(s_i, s_i', s_f) = 2(nmsgs_{to}(s_f) + nmsgs_{from}(s_f)) + \chi_{timer}$.

Suppose that $s_i H s_f$. We consider possible cases for the transition $s_i \to s_i'$. For convenience, we denote the object ID of the internal wrapped object to be O_w.

(1) $mte(s_i) = 0$ because a message M needs to be delivered in s_i, we consider the following cases:

(1.a) M is not a message to O_w, and M is in the external configuration of s_f, then $s_f \to s_f'$ takes a corresponding transition by using the same rule, and we have $s_i' H s_f'$.

(1.b) M is a message to O_w, and M is in the internal wrapped configuration of s_f, then, again, we can just take a corresponding transition $s_f \to s_f'$ in the wrapped configuration using the same rule, and we have $s_i' H s_f'$.

(1.c) M is a message to O_w, and M is in the external configuration of s_f, and M is not a press message, then the `[forward-in]` rule is used to transition $s_f \to s_f'$, and we have $s_i H s_f'$, and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since $nmsgs_{to}$ decreases.

(1.d) M is a message not to O_w but in the internal configuration of s_f. Then the `[forward-out]` rule is used to transition $s_f \to s_f'$, we have $s_i H s_f'$ and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since $nmsgs_{from}$ decreases.

(1.e) M is a message to O_w in the external configuration of s_f, and M is a press. The rule `[set-timer]` can be applied to transition $s_f \to s_f'$. In this case, we have $s_i H s_f'$, and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$ since $2nmsgs_{to} + \chi_{timer}$ decreases.

(1.f) In the case that M is a release message, then the `[release-event]` rule is used, and it will be similar to the `[forward-in]` case with a possible added decrease in the value of χ_{timer}.

(2) if $mte(s_i) > 0$, then $s_i \to s_i'$ is a tick rule advancing time by one time unit, and we have a few subcases to consider for possible transitions $s_f \to s_f'$. Note that in this case, by the strict nature of the projection relation, there cannot be any messages waiting to be delivered in s_f.

(2.a) $mte(s_f) > 0$, we have a corresponding tick rule $s_f \to s_f'$ by one time that preserves the relation H.

(2.b) $mte(s_f) = 0$ because the timer in the wrapper is expired, in this case, the only rule to take is to stop the timer, so $s_f \to s_f'$ changes the timer from `t(0)` to `no-timer` and changes the value of `stuck-err`. We have $s_i H s_f'$ and $\mu(s_i, s_i', s_f') < \mu(s_i, s_i', s_f)$, since the value of χ_{timer} decreases and all other values stay the same.

Now, we show the other direction, namely, that a well-founded simulation from S_{ideal} to $S_{wrapped}$ exists.

Define $\nu(s_f, s_i) = 2(nmsgs_{to}(s_f) + nmsgs_{from}(s_f)) + \chi_{timer}(s_f)$ and $\nu(s_f, s'_f, s_i) = 0$. Let $H' = H^{-1}$, and suppose $s_f H' s_i$. We consider the following cases:

(1') $mte(s_f) > 0$, and $s_f \to s'_f$ is a tick rule by 1 time unit. By the homomorphic definition of mte, if $s_f H' s_i$, then $mte(s_i) > 0$, and a corresponding 1 time unit tick step can be taken $s_i \to s'_i$ with $s'_f H' s'_i$.

(2') $s_f \to s'_f$ uses the `[forward-in]` rule. In this case, $s'_f H' s_i$, and $\nu(s_f, s_i) > \nu(s'_f, s_i)$, since $nmsgs_{to}$ decreases.

(3') $s_f \to s'_f$ uses the `[forward-out]` rule. Again, we have $s'_f H' s_i$, and ν decreases, since $nmsgs_{from}$ decreases.

(4') $s_f \to s'_f$ uses the `[set-timer]` rule. Since $s_f H' s_i$, we know that the press message forwarded also exists in s_i, and again, $s'_f H' s_i$, and ν decreases, since $2nmsgs_{to} + \chi_{timer}$ decreases.

(5') $s_f \to s'_f$ uses the `[release-event]` rule. We have $s'_f H' s_i$ and ν decreases, since $2nmsgs_{to} + \chi_{timer}$ decreases.

(6') $s_f \to s'_f$ uses the `[stuck-event]` rule. We have $s'_f H' s_i$ and ν decreases, since χ_{timer} decreases.

(7') $s_f \to s'_f$ has $mte(s_f) = 0$, and it transitions by a rule not in the pattern module. We assume that all zero-time rules not in the pattern module are about consuming messages by objects, and additionally the wrapper object is not on either side of these rules. Since $s_f H' s_i$, we can take a corresponding transition $s_i \to s'_i$ using the same rule with $s'_f H' s'_i$.

This shows that H and H^{-1} are well-founded simulations, and therefore, we have a bisimulation between the two systems as desired.

□

This shows 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 is a bisimulation, and the added wrapper makes no changes to the behavior of the system. However when a button does become stuck, we can no longer give any guarantees about correct behavior, but we can still detect it.

Theorem 3.5.2. *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.*

Proof. First, notice that by the characterization of inputs, the press and release events must alternate. Thus, at t time units, we either have the first `press` event, in which case the timer is initalized to `no-timer`, or we have the `press` event following a previous `release` event which must have been consumed by applying the `[release-event]` rule, in which case the timer attribute is still set to `no-timer`. Thus upon receiving

the press event at time t, the rule `[set-timer]` must be applied. This will set the timer value to be T_{stuck} and at $t + T_{stuck}$ time units, we still have received a release message since $t' > t + T_{stuck}$. Thus, the rule `[stuck-event]` will be applied, and the `stuck-err` attribute is set to true at $t + T_{stuck}$ time units. □

Chapter 4

A Device Level Safety Pattern

4.1 Safety of Life-Critical Medical Devices

In this chapter we describe in detail a safety pattern called the *Command-Shaper Pattern for Medical Devices* that we have found applicable to a range of medical devices. However, before we can talk about a generic safety pattern, we must first discuss a generic notion of safety for a certain class of medical devices receiving commands that may be unsafe in some circumstances.

When studying medical device operation, we have found a recurring pattern of *command restrictions*. Consider the following three examples:

- Infusion pumps for pain medication are normally incorporated into Patient Controlled Analgesia (PCA) systems, where the patient can demand additional bolus doses of drugs with the push of a button. If the patient pushes the button too often, this will clearly lead to depression of the central nervous system and even death. The PCA needs some safety mechanism to make sure that not too many bolus doses are administered.

- Pacemakers normally need to adapt the heart rate to the amount of patient activity. However, pacemaker activity sensors often pick up false positives during bumpy car rides or from noisy vacuum cleaners. Pacing a heart at high rates for a prolonged period of time could lead to patient discomfort or even cardiac arrest. Thus, pacemakers must have safety mechanisms to prevent them from pacing too fast for too long.

- A ventilator machine may need to be turned off temporarily for another piece of equipment to work. Sometimes medical personnel will forget to turn the ventilator machine back on, potentially causing brain damage to the patient due to oxygen deprivation. The ventilator should have time triggers to make sure that it does not turn off too often or for too long.

Intuitively these examples illustrate some common theme with medical devices: human bodies are normally self stabilizing, and our bodies can normally be placed under some stress temporarily, provided they

are given sufficient time to recover.

Thus, for the medical device examples presented above, all of the device states can be partitioned into two types: stressed states and relaxed states. Stressed states are states where the patient cannot stay for too long (heart pacing too fast, holding breath for too long, etc.) because permanent physical harm may result for the patient. Relaxed states are states that allow the patient to recover over time from a previous period of stressed states.

Although the partitioning of device states into stressed and relaxed states is common to many devices, it should be noted that not all devices can be placed into this category. For example, a glucose-insulin pump does not have any static relaxed states. There always exist patient contexts where any potential device state: *infuse insulin*, *infuse glucose*, or *do nothing* could be considered an unsafe action. These devices, which depend on external context and sensor information for their safety, are not addressed by the safety pattern that we present in this chapter.

4.2 A Wrapper Pattern for Safety Monitoring

4.2.1 Patterns as Parameterized Specifications

Parameterized modules are very powerful constructs, since a parameterized module really defines a wide range of modules, one for each possible correct instantiation of its input theories. This means that any stable theorems we prove about a parameterized module should hold no matter what instantiation of the input theories are given. This has a nice correspondence with design patterns (design structures that can be reused within different contexts). If a design pattern can be formalized as a parametrized module and we prove a safety property for it, then whenever we apply the pattern to a system (assuming the context satisfies all the preconditions specified by the input theories), we can be sure that the safety property holds in the instantiated system also.

4.2.2 Overall Idea of the Command-Shaper Pattern

The key idea of the command-shaper pattern is that *commands from external devices should only be taken as suggestions.* Figure 4.1 shows this pattern applied in the form of a wrapper around a pacing module in a cardiac pacemaker. If a command is detected to be deviating or unsafe, the command-shaper can either ignore the command, or more generally, modify the command into something more reasonable. To do this, we must first come up with a general definition of which commands are safe, and also, of how to respond to commands that are unsafe.

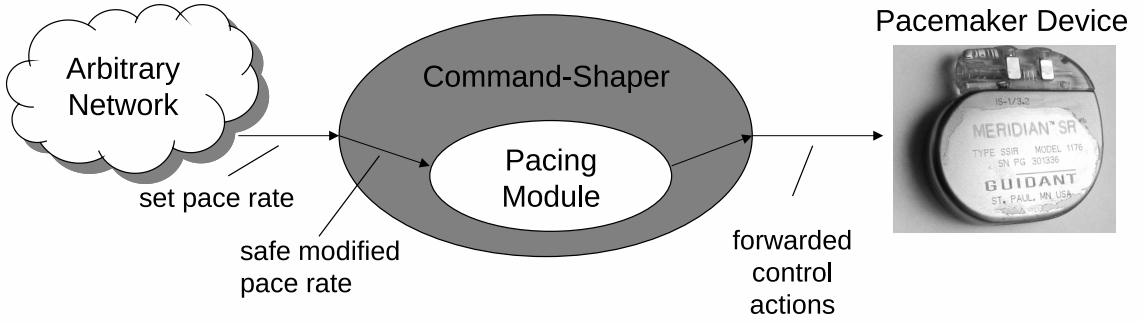

Figure 4.1: Command-Shaper Wrapper Pattern for the Pacemaker

The relavant notion of safety is defined via stressed and relaxed states. Consider the heart rate at which a pacemaker is pacing the heart. We assume that the doctors customized the pacemaker's parameters to adapt to the patient's normal expected heart rythm. There is a base rate that is assumed to be a safe minimal heart rate for an inactive patient. In this list of parameters there should also be a threshold heart rate above which pacing is considered fast. Any heart rate above this threshold will be considered stressed. Also, any heart rate below this threshold will be considered relaxed. Suppose the patient's threshold heart rate is 100 bpm. Every time the heart rate rises above 100 bpm a *stress event* is recorded in the model. Every time the heart rate decreases below 100 bpm a *relax event* is recorded. Assuming all other bodily functions are operating normally, a log of stress and relax events over time can determine whether the patient is safe or not. That is, safety can be defined as a function of the stress and relax events recorded over time. For example if too much time elapses after a stress event and before a relax event, then this is unsafe. This means that the patient has been in a stressed state too long. Similarly, if very little time elapses after a relax event followed by a stress event, then this is also unsafe. This is the same as the intuitive notion that we have too little time to relax before becoming stressed again. Thus, any reasonable definition of safety must satisfy these two properties.

One more point for safety is that, normally, for proper device operation continuous states should change gradually in order for the patient to adapt to the effects. For example, in a pacemaker, if the pacing rate of 100 bpm immediately drops to 60 bpm over 1 second, the patient may start to feel a bit light headed. Patient safety in this case requires constraining how fast the pacing rate can change over time.

Now that the appropriate notion of safety has been analyzed, we can also describe how to detect whether a command will be safe or not. Any command that takes the system into a relaxed state is by default safe,

because the longer we stay in relaxed states the safer the operation. The only commands we need to worry about are those that take us into stressed states. To detect whether these commands are safe or not, we must consider whether complying with a command to enter a stressed state will still allow the system to go back to a relaxed state in the short period of time that is considered safe. If the command violates this condition, then the system should immediately start to bring the system back to a relaxed state if it was not already in a relaxed state to begin with.

4.2.3 Formal Models of Event Streams

In any system, changes in untimed state are caused by significant events that occur in the system. Many times it is useful to keep track of these significant events. These can either be used as auxiliary model elements to define safety properties during model checking, or they can be part of the system implementation itself (e.g. black-box recording). Events of interest differ based on where they are used. Therefore, an event stream is parametrized by the set of events that the system handles. This can be captured by the following parameterized module `EVENT-LOG` (note that the sort `Mt-Log` (empty log) is added to ensure that the empty log `nil` will have a least sort when there are multiple distinct instances of `EVENT-LOG`):

```
(fmod EVENT-LOG{X :: TRIV} is pr RT-COMP .
   sorts NoEventType{X} EventType{X} Event{X} .
   sorts Mt-Log Stopped-Event-Log{X} Event-Log{X} .
   subsorts X$Elt NoEventType{X} < EventType{X} .
   subsort Mt-Log < Stopped-Event-Log{X} < Event-Log{X} < RT-Comp .

   op none : -> NoEventType{X} [ctor] .
   op E : X$Elt Clock -> Event{X} [ctor] .
   op type : Event{X} -> X$Elt .
   op elapsed : Event{X} -> Time .
   op stop : Event{X} -> Event{X} .
   op stopped? : Event{X} -> Bool .

   op tick : Event{X} Time -> Event{X} .
   op mte : Event{X} -> Time .

   var EV : Event{X} . var ET : X$Elt .
```

```
var C : Clock . var L L' : Event-Log{X} .
var SL : Stopped-Event-Log{X} .
var T T' : Time .

eq type(E(ET, C)) = ET .
eq elapsed(E(ET, C)) = value(C) .
eq stop(E(ET, C)) = E(ET, stop(C)) .
eq stopped?(E(ET, C)) = stopped?(C) .

eq tick(E(ET, C), T) = E(ET, tick(C, T)) .
eq mte(E(ET, C)) = mte(C) .

op nil : -> Mt-Log [ctor] .
op __ : Event{X} Stopped-Event-Log{X} -> Event-Log{X} [ctor] .
op stop : Mt-Log -> Mt-Log .
op stop : Event-Log{X} -> Stopped-Event-Log{X} .
op append : Mt-Log Mt-Log -> Mt-Log .
op append : Event-Log{X} Stopped-Event-Log{X} -> Event-Log{X} .

cmb EV SL : Stopped-Event-Log{X} if stopped?(EV) .

op log : Mt-Log NoEventType{X} -> Mt-Log .
op log : Event-Log{X} EventType{X} -> Event-Log{X} .

op tick : Mt-Log Time -> Mt-Log .
op tick : Event-Log{X} Time -> Event-Log{X} .
op mte : Event-Log{X} -> Time .

eq stop(nil) = nil .
eq stop(EV L) = stop(EV) stop(L) .
eq append(nil, L) = L .
eq append(EV L , L') = EV append(L, L') .
```

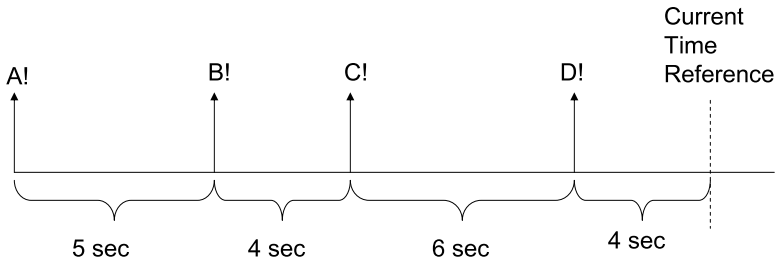

Figure 4.2: Event Example

```
   eq log(L, none) = L .
   eq log(L, ET) = E(ET, clock0) stop(L) [owise] .

   eq tick(nil, T) = nil .
   --- only the latest event needs to be ticked
   --- remaining event clocks are stopped
   eq tick(EV L, T) = tick(EV, T) L .
   eq mte(nil) = INF .
   eq mte(EV L) = mte(EV) .
endfm)
```

To understand event logs it is helpful to provide an example of how one such log is represented. Figure 4.2 shows a stream of events (with event names A!, B!, C!, D!) and the time durations between them. Also, there is a reference point to the current time the system is at. To represent this set of events, we first need to instantiate the set of event names with an appropriate module and view:

```
(fmod ABCD-EVENT is
   sort EventName .
   ops A! B! C! D! : -> EventName .
endfm)

(view ABCD-Event from TRIV to ABCD-EVENT is
   sort Elt to EventName .
endv)

(fmod ABCD-EVENT-LOG is
   pr EVENT-LOG{ABCD-Event} .
```

```
   pr POSRAT-TIME-DOMAIN-WITH-INF .


   op events : -> Event-Log{ABCD-Event} .
   eq events =
      E(D!, c(4, run)) E(C!, c(6, stop)) E(B!, c(4, stop))
         E(A!, c(5, stop)) nil .
endfm)
```

The constant `events` corresponds exactly to the graphical representation of the events in Figure 4.2. Furthermore, ticking `events` increases the time elapsed for the latest event:

```
reduce in ABCD-EVENT-LOG :
  tick(events,10)
result Event-Log{ABCD-Event} :
  E(D!,c(14,run))E(C!,c(6,stop))E(B!,c(4,stop))E(A!,c(5,stop))nil
```

Finally, adding an event to the log locks (stops all the clocks for) the time elapsed of all previous events and starts a clock for the newest event:

```
reduce in ABCD-EVENT-LOG :
  log(tick(events,10),A!)
result Event-Log{ABCD-Event} :
  E(A!,c(0,run))E(D!,c(14,stop))E(C!,c(6,stop))E(B!,c(4,stop))E(A!,c(5,
    stop))nil
```

We would also like to highlight a particularly important type of event log for the medical device wrapper, namely, the *stress-relax log*. An example of a stress-relax log is shown in Figure 4.3. Given a critical threshold for a value that changes over time, a !*stress* event is recorded whenever the value crosses above the threshold, and a !*relax* event is recorded whenever the value crosses below the threshold. Also, we assume that the system starts at a value below the threshold. Notice that a stress-relax log imposes additional structure on top of event logs:

1. The first event logged must be a !*stress* event
2. Events must alternate between !*stress* and !*relax* events over time

These constraints, and the notion of a stress-relax log, are both captured by the following parameterized instantiation (plus additional constraints) on event logs:

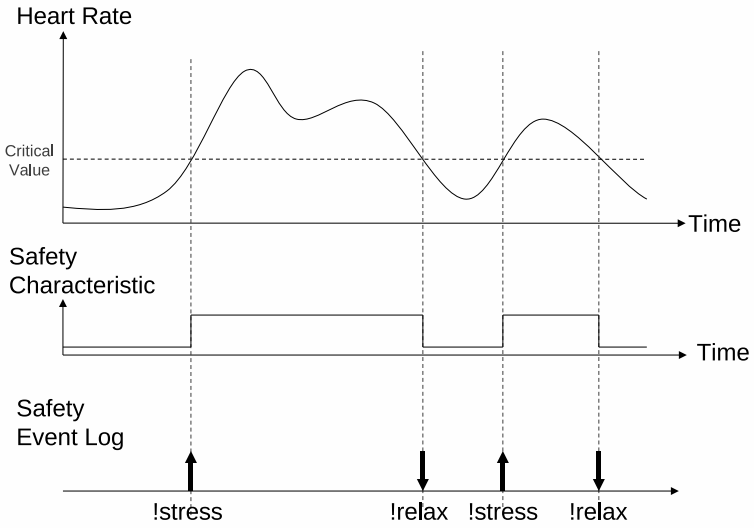

Figure 4.3: Stress Relax Event Log

```
(fmod STRESS-RELAX-EVENT is
   sort SREvent .
   ops !stress !relax : -> SREvent [ctor] .
endfm)

(view SREvent from TRIV to STRESS-RELAX-EVENT is
   sort Elt to SREvent .
endv)

(fmod STRESS-RELAX-LOG is
   inc EVENT-LOG{SREvent} .

   sorts Stopped-Stress-Relax-Log Stress-Relax-Log .
   subsorts Mt-Log
      < Stopped-Stress-Relax-Log
      < Stopped-Event-Log{SREvent} .
```

```
   subsorts Stopped-Stress-Relax-Log
      < Stress-Relax-Log < Event-Log{SREvent} .

   op tick : Stress-Relax-Log Time -> Stress-Relax-Log .

   var C C' : Clock .
   var SL : Stopped-Stress-Relax-Log .
   var L : Stress-Relax-Log .

   cmb E(!stress, C) nil : Stopped-Stress-Relax-Log
      if stopped?(C) .
   cmb E(!stress, C) E(!relax, C') SL : Stopped-Stress-Relax-Log
      if E(!relax, C') SL : Stopped-Stress-Relax-Log
      /\ stopped?(C) .
   cmb E(!relax, C) E(!stress, C') SL : Stopped-Stress-Relax-Log
      if E(!stress, C') SL : Stopped-Stress-Relax-Log
      /\ stopped?(C) .

   mb E(!stress, C) nil : Stress-Relax-Log .
   cmb E(!stress, C) E(!relax, C') SL : Stress-Relax-Log
      if E(!relax, C') SL : Stopped-Stress-Relax-Log .
   cmb E(!relax, C) E(!stress, C') SL : Stress-Relax-Log
      if E(!stress, C') SL : Stopped-Stress-Relax-Log .
endfm)
```

4.2.4 Abstract Safety Definition

It would be absurd to define a *safety* pattern without first defining what it means to be safe. Furthermore, it would be desirable to develop a *generic* form of safety for the class of devices receiving potentially unsafe commands that we are considering so that, once instantiated, it can be used as an input to derive a concrete design. This derivation of a design from safety definitions is exactly what is captured in the notion of a formal safety pattern. This subsection describes the generic theory of safety for the class of medical devices described in Section 4.1.

Devices and their states are modeled as objects. For a device that has statically enforceable safety, we characterize states by a totally ordered set of values with an upper and lower bound, v_{max} and v_{min}, respectively. Intuitively, the order compares the relative safety of two states, with smaller values being safer than larger values. Any state outside the upper and lower bound is assumed to be invalid. From now on, we assume that only valid device states are set. Furthermore, there is a critical value v_{crit} such that any valid value v, is considered *relaxed* when $v \leq v_{crit}$, and is considered *stressed* when $v > v_{crit}$.

As a brief example, a pacing module with the state being the pacing rate may have $v_{min} = 60$bpm, $v_{max} = 120$bpm, and $v_{crit} = 90$bpm. Thus, a rate of 40bpm or 200bpm is invalid, a rate of 70bpm is *relaxed*, and a rate of 100bpm is *stressed*.

Furthermore, we want to have a generic way of characterizing the safe amount of time that devices can remain in relaxed and stressed states. This is captured by a predicate `safe?` defined on the `Stress-Relax-Log` of the system state. Implicitly, it is assumed that a `!stress` event is recorded when the device state rises above v_{crit}, and a `!relax` event is recorded when the state falls to or below v_{crit}. The `safe?` predicate is left generic, so that arbitrarily complex conditions on time bounds can be defined on stress and relax intervals because the `Stress-Relax-Log` maintains all the history. However `safe?` should satisfy some monotonicity assumptions based on our knowledge of the domain. Some of these assumptions include that staying in a stress situation for a longer amount of time cannot make a device safe when it was unsafe before. Also, staying in a relax situation for a longer amount of time should keep the device safe if it was safe before.

The detailed requirements for state safety are captured in the following theory `SAFE-STATE`, whose axioms are discussed subsequently.

```
(fth SAFE-STATE is
   inc TOTAL-ORDER * (
      sort Elt to Val,
      op _<=_ to _<=risk_
      ) .
   pr STRESS-RELAX-LOG .

   ops min-val max-val crit-val safe-val : -> Val .

   eq min-val <=risk safe-val = true .
   eq safe-val <=risk crit-val = true .
   eq crit-val <=risk max-val = true .
```

```
   op period : -> Time . --- period of wrapper dispatch

   op delta : Val Val -> Val .
   op tdelta-min : Val Val -> TimeInf .
   ...
   op safe? : Stress-Relax-Log -> Bool .
   op norm : Stress-Relax-Log -> Stress-Relax-Log .
   ...
endfth)
```

The `SAFE-STATE` theory defines an operator `<=risk`, which we shall write as $\leq_{risk}$, where $A \leq_{risk} B$ means that A is safer or less risky than B. Furthermore, in terms of risk, there are four key constants $v_{min} \leq_{risk} v_{safe} \leq_{risk} v_{crit} \leq_{risk} v_{max}$ as described before. The relations between these values are described by the following equations:

```
   eq min-val <=risk safe-val = true .
   eq safe-val <=risk crit-val = true .
   eq crit-val <=risk max-val = true .
```

Real-time systems normally require a period of operation. Here, `period`, conventionally denoted T, is assumed to have sort *Time* and defines a constant period of execution. For two values V, V', the operator `delta`(V, V'), which we will write as $\Delta(V, V')$, defines the value maximally changed from V towards the direction of V' in one period. The `tdelta-min`(V, V'), which we write as $t_{\Delta min}(V, V')$, defines the minimum amount of time it will take to change from V to V'. Because values are assumed to be modified only during dispatch, which occurs once every period, it follows that $t_{\Delta min}$ must be a multiple of the period. The functions $\Delta(V, V')$ and $t_{\Delta min}(V, V')$ are related by the following equations:

```
   var V V' V'' : Val .
   eq [no-delta] : delta(V, V) = V .
   ceq [delta-bound] :
      ((V <=risk V'') and (V'' <=risk V')) or
      ((V' <=risk V'') and (V'' <=risk V)) = true
      if delta(V, V') = V'' .
   ceq [delta-mono] : delta(V, V') <=risk delta(V, V'') = true if V' <=risk V'' .
```

```
ceq [delta-max] : delta(V,V') = delta(V,V'') = true
   if delta(V,V') =/= V' /\ V' <=risk V'' .

eq [no-tdelta] : tdelta-min(V, V) = zero .
ceq [ind-tdelta] : tdelta-min(V, V') = tdelta-min(delta(V,V'), V') plus period
   if V =/= V' .
ceq [tdelta-mono] : tdelta-min(V', V) le tdelta-min(V'', V) = true
   if V <=risk V' /\ V' <=risk V'' .
```

The first few equations state that Δ must change V towards V' without going over. That is, if $V \leq_{risk} V'$, then $V \leq_{risk} \Delta(V, V') \leq_{risk} V''$. If $V' \leq_{risk} V$, then $V' \leq_{risk} \Delta(V, V') \leq_{risk} V$. Also, we have a `delta-max` constraint, which essentially states that if a value is changing to become a safe value, it should change at a *maximal* rate, unless the target value can be reached in one step.

The next set of equations describe how $t_{\Delta min}$ relates to Δ. Essentially, to change from V to V', the time $t_{\Delta min}(V, V')$ is inductively characterized (equation `ind-tdel`) by the number periods required for to reach V' using Δ. The final `tdel-mono` constraint makes sure that the minimum time taken to reach a safer value cannot be larger than closer values.

It is important to note that if repeated applications of Δ from V towards V' never reach V', then it should be the case that $t_{\Delta min}(V, V') = \texttt{INF}$. The equations still hold in this case because of the equality $\texttt{INF} + T = \texttt{INF}$. Normally, we do not have to worry about `INF`, when all values are eventually reachable from every other value.

The most important function in `SAFE-STATE` is the predicate `safe?` which characterizes whether a `Stress-Relax-Log` is safe (intuitively describing the time bounds on stressed states for safety). Furthermore, we define a `norm` operator, which will shorten the length of a `Stress-Relax-Log` without influencing the value of the `safe?` predicate over time. The definition of `safe?` must satisfy the following conditions:

```
var T T' : Time .
var CA : ClockAttr .
var SRE : SREvent .
var L L' : Stress-Relax-Log .
eq [initially-safe] : safe?(nil) = true .
--- instantaneous safety equivalent to safety with stopped time
eq [stopped-safe] : safe?(E(SRE, c(T, CA)) L) =
   safe?(E(SRE, c(T, stop)) L) .
```

```
--- safety implies safety of all sub-logs
eq [sub-safe] : safe?(E(SRE, c(T, CA)) L)
   implies safe?(L) = true .
--- decreasing stress durations preserves safety
ceq [stress-safe] : safe?(E(!stress, c(T', CA)) L) implies
   safe?(E(!stress, c(T, CA)) L) = true
   if T' ge T .
--- safety is preserved while in a relaxed state
eq [relax-safe] : safe?(E(!relax, c(T, CA)) L)   =
   safe?(L) .
```

A `Stress-Relax-Log` is assumed to be initially `nil`, so the equation `initially-safe` states that the system is assumed to satisfy `safe?` initially. The equation `stopped-safe` states the simple fact that stopping the duration of the last event does not affect the value of `safe?`. The next three equations define the monotonic property of `safe?`. The equation `sub-safe` states that any sub-log of a log satisfying `safe?` must also satisfy `safe?`. The equation `stress-safe` states that if a stress duration is made shorter, it cannot make the system unsafe if it was safe before. The equation `relax-safe` states that adding any relaxed duration will be safe if the log was safe before.

The last operator `norm` is really a modeling construct used for efficiency of representation. Instead of maintaining an unbounded list of events that grows over time, it is often possible to contract or normalize the history of events to a list of bounded size. This is also important to ensure that term sizes and the untimed state space are not unbounded (in the latter case, the equations defining the `norm` operator can be thought of as an equational abstraction).

```
--- normalization preserves log structure
--- norm-tick-sort
cmb tick(norm(L), T) : Stress-Relax-Log
    if tick(L, T) : Stress-Relax-Log .
--- norm-app-sort
cmb append(L', stop(norm(L))) : Stress-Relax-Log
   if append(L', stop(L)) : Stress-Relax-Log .
--- normalization preserves safety
eq [norm-tick-safe] : safe?(tick(norm(L), T)) =
   safe?(tick(L, T)) .
```

```
ceq [norm-log-safe] : safe?(append(L', stop(norm(L)))) =
   safe?(append(L', stop(L)))
   if append(L', stop(L)) : Stress-Relax-Log .
--- normalization is idempotent
eq [norm-tick] : norm(tick(norm(L), T)) = norm(tick(L, T)) .
eq [norm-app] : norm(append(L', stop(tick(norm(L), T))))
   = norm(append(L', stop(tick(L, T)))) .
```

The first two membership statements are requirements on the sort-preserving (i.e. data-structure-preserving) property of `norm`. These state that `norm` preserves the `Stress-Relax-Log` structure of the list. The next few equations state that `norm` will not affect the value of `safe?`. The last equations *norm-tick* and *norm-app* state that `norm` is idempotent, in that applying `norm` is not affected by also applying `norm` to any sub-logs. It can be shown, based on the equational requirements of `norm`, that applications of `norm` cannot alter the value of `safe?` on a list, regardless of how it is interleaved with applications of the `tick` operator.

4.3 Command Shaper Pattern as a Parametrized Specification

Now that we have a formal definition of what we are considering for safety. We describe the pattern that addresses all of these safety requirements in a medical device. Mostly intercepting commands to medical devices so that:

- device operation stay within minimum and maximum thresholds,
- device operation does not exceed maximum rate of change requirements,
- device operation satisfies timing requirements defined by stress-relax requirements.

4.3.1 Wrapped Object Theory

Before we describe a wrapper, it is necessary to describe exactly what it is that we are wrapping. As already mentioned, the wrapped object is assumed to have a state and to receive external messages to change this state. Furthermore, it may also send output messages to interact with its environment. We define the `WRAPPED-OBJECT` theory as follows:

```
(oth WRAPPED-OBJECT is
```

```
   inc SAFE-STATE .
   inc TICK-MTE-SEM .

   class Wrapped | set-val : Val .

   sort InMsg .
   sort OutMsg .
   subsorts InMsg OutMsg < Msg .

   msg set-val : Oid Val -> InMsg .

   var O : Oid .
   vars V V' : Val .
   eq mte(set-val(O, V)) = zero .

   rl [wrapped-recv] : set-val(O, V)
      < O : Wrapped | set-val : V' >
      => < O : Wrapped | set-val : V > .

   --- assumes that all messages sent to external objects are of sort OutMsg
   --- assumes that set-val is the only message of sort InMsg sent from external objects
endoth)
```

The wrapped object is of class `Wrapped`, and has a single attribute `set-val` for its state (recall that this is the totally ordered set of values with the order operator $\leq_{risk}$ from the theory `SAFE-STATE`). We distinguish messages into input and output messages. This is so that the wrapper knows which messages to forward to the external configuration. A message `set-val` is assumed to be the only message of sort `InMsg`. Also, the behavior when receiving the message is fixed, and it is assumed that the wrapped object will immediately set the state to the new state specified by the message. The fact that the message is immediately delivered is specified by the fact that `mte` is 0, so that time will not be allowed to advance until the message is received.

4.3.2 Parameterized Wrapper Object

The wrapper class structure is parametrized based on an instance of the WRAPPED-OBJ theory. The definition of the wrapper is split up into three modules for ease of readability. The first part of the wrapper defines the wrapper object itself with the appropriate accessor and modifier operations for each attribute. This is shown in the parametrized EPR-WRAPPER module:

```
(tomod EPR-WRAPPER{X :: WRAPPED-OBJECT} is
   class EPR-Wrapper{X} | inside : NEConfiguration,
      next-val : X$Val, val : X$Val, disp : Timer,
      stress-intervals : Stress-Relax-Log .

   op _get-next-val : Object ~> X$Val [frozen] .
   op _get-val : Object ~> X$Val [frozen] .

   op _set-next-val_ : Object X$Val ~> Object [frozen] .
   op _set-val_ : Object X$Val ~> Object [frozen] .
   op _log-stress_ : Object EventType{SREvent} ~> Object [frozen] .
   op _norm-stress : Object ~> Object [frozen] .
   op _deliver : Object ~> Object [frozen] .

   var O : Oid .
   var V V' : X$Val .
   var L : Stress-Relax-Log .
   var E : EventType{SREvent} .
   var C : NEConfiguration .

   eq < O : EPR-Wrapper{X} | next-val : V > get-next-val = V .
   eq < O : EPR-Wrapper{X} | val : V > get-val = V .

   eq < O : EPR-Wrapper{X} | next-val : V > set-next-val V'
      = < O : EPR-Wrapper{X} | next-val : V' > .
   eq < O : EPR-Wrapper{X} | val : V > set-val V'
      = < O : EPR-Wrapper{X} | val : V' > .
```

```
   eq < O : EPR-Wrapper{X} | stress-intervals : L > log-stress E
      = < O : EPR-Wrapper{X} | stress-intervals : log(L, E) > .
   eq < O : EPR-Wrapper{X} | stress-intervals : L > norm-stress
      = < O : EPR-Wrapper{X} | stress-intervals : norm(L) > .
   eq < O : EPR-Wrapper{X} | inside : C, val : V > deliver
      = < O : EPR-Wrapper{X} | inside : (set-val(O, V) C) > .
endtom)
```

The `EPR-Wrapper` class has four attributes. The `inside` attribute defines the internal wrapped configuration. This is assumed to contain an instance of an object of the instantiated class `Wrapped` and possibly various messages of type `InMsg` and `OutMsg`. The `next-val` and `val` attributes describe the next requested (target) state and the current state respectively. The last attribute `stress-intervals` is the log of stress and relaxed events used to evaluate safety. All the operators perform the intuitive operations of accessing and modifying objects. Notice how they are all frozen in order to prevent any rewrites while we are trying to obtain the attributes for the wrapper object.

4.3.3 Safety Envelope Calculations for the Wrapper

It is useful to have a parametrized module to describe various auxiliary operations based on the operations defined in the theory `SAFE-STATE`:

```
(fmod WRAPPER-AUX{X :: WRAPPED-OBJECT} is inc EPR-WRAPPER{X} .

   op cap : X$Val -> X$Val .
   op stress? : X$Val -> Bool .
   ops toStress? toRelax? : X$Val X$Val -> Bool .
   op inEnv? : X$Val Stress-Relax-Log -> Bool .

   var V V' : X$Val .
   ceq cap(V) = min-val if V <=risk min-val .
   ceq cap(V) = max-val if max-val <=risk V .
   eq cap(V) = V [owise] .

   eq stress?(V) = not (V <=risk crit-val) .
```

```
   eq toStress?(V, V') = not stress?(V) and stress?(V') .
   eq toRelax?(V, V') = stress?(V) and not stress?(V') .

   var L : Stress-Relax-Log .
   ceq [inenv-unreachable] : inEnv?(V, L) = false
      if tdelta-min(V, crit-val) == INF /\ stress?(V) .
   ceq [inenv-stress] : inEnv?(V, L) = safe?(L) and
      safe?(log(tick(L, tdelta-min(V, crit-val) plus period), !relax))
      if stress?(V) /\ tdelta-min(V, crit-val) :: Time .
   ceq [inenv-relax] : inEnv?(V, L) = safe?(L) if not stress?(V) .
endfm)
```

The operator `cap` changes the value to be within the min and max risk range, if it was originally outside of that range. The predicates `toStress?` and `toRelax?` describe when a value has crossed the `crit-val` threshold in the more risky direction or less risky direction, respectively.

The last predicate `inEnv?`, which is an abbreviation for *inside the envelope*, is one of the most important predicates for the pattern. It is used to detect whether a configuration can persistently satisfy the `safe?` predicate in the future by performing a look ahead to see the shortest time it will take to reach a relaxed state. It is worthwhile to discuss the properties of the `inEv?` predicate in more detail.

It is easy to see that `inEnv?` is a stronger predicate than `safe?`. All the equations for `inEnv?` can be written in the form $\texttt{inEv?}(V, L) = \texttt{safe?}(L) \wedge t$ (where t is an abitrary boolean term). Essentially, `inEnv?` strengthens `safe?` so that it becomes inductive for each step of system execution. That is, if $inEv?(V, L) = true$, then there always exists a controllable path of operation for the system to remain safe. Figure 4.4 provides an intuition for this as a device state can only be in the envelope when it has sufficient time to transition back to a safe state. The top part of the figure shows the shaded regions satisfying `safe?`, and the bottom part of the figure shows the shaded regions satisfying `inEv?`.

4.3.4 Wrapper Execution

Finally, with all the auxiliary functions and the wrapper object fully defined, we can now describe how a wrapper object should execute. Intuitively, the wrapper should filter and correct improper settings for the state, so that the system always remains safe.

The wrapper executes with periodicity given by the `period` defined in the input theory `SAFE-STATE`. During each period of execution, the wrapper must update the set state so that safety is maintained. The

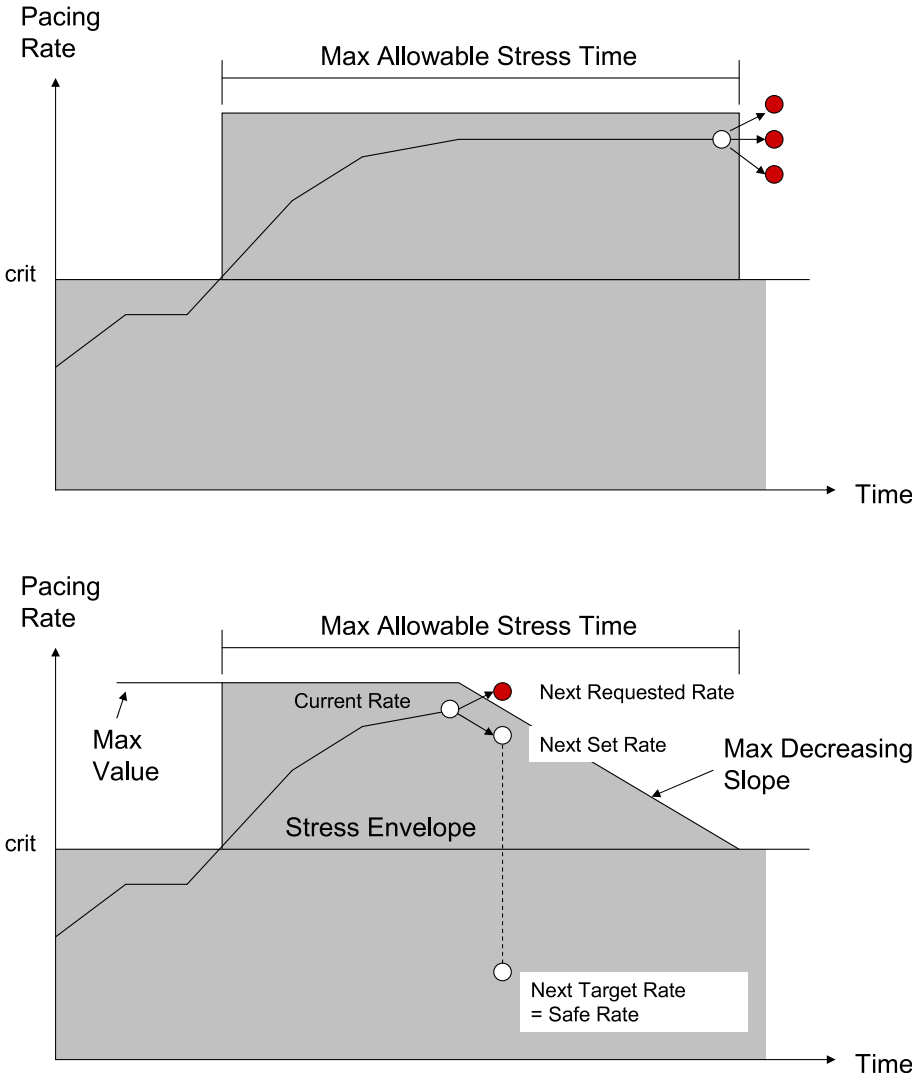

Figure 4.4: Characterization of the Stress Envelope

wrapper also logs events `!stress` or `!relax` whenever the value of the predicate `stress?` changes for the state of the system. Furthermore, any messages the wrapper receives will be buffered immediately in the attribute `next-val`, but this attribute will not be used until the next dispatch time corresponding to the period of execution. This means that if multiple messages are received, only the last buffered message is used in the computation of the next state. The specification of the wrapper execution is describe in the module `EPR-WRAPPER-EXEC`:

```
(tomod EPR-WRAPPER-EXEC{X :: WRAPPED-OBJECT} is
   inc WRAPPER-AUX{X} .

   op log-entry : X$Val X$Val ~> EventType{SREvent} .
   op next-val : Object ~> X$Val [frozen] .

   var O : Oid .
   var L : Stress-Relax-Log .
   var V V' V'' : X$Val .
   var T T' : Time .

   ceq log-entry(V, V') = !stress if toStress?(V, V') .
   ceq log-entry(V, V') = !relax if toRelax?(V, V') .
   eq log-entry(V, V') = none [owise] .

   ceq next-val(< O : EPR-Wrapper{X} | val : V, next-val : V', stress-intervals : L >
      = delta(V, safe-val)
      if not inEnv?(delta(V, V'), log(L, log-entry(V, delta(V, V')))) .

   ceq next-val(< O : EPR-Wrapper{X} | val : V, next-val : V', stress-intervals : L >)
      = delta(V, V') if inEnv?(delta(V, V'), log(L, log-entry(V, delta(V, V')))) .

   var C : NEConfiguration .
   var M : X$OutMsg .

   crl [dispatch] :
```

```
      < O : EPR-Wrapper{X} | disp : t(T), val : V, next-val : V' >
      => (< O : EPR-Wrapper{X} | disp : t(period), next-val : V'', val : V'' >
         log-stress log-entry(V, V'')) norm-stress deliver
      if V'' := next-val(< O : EPR-Wrapper{X} | >) /\ T == zero .
   rl [recv] : set-val(O, V) < O : EPR-Wrapper{X} | >
      => < O : EPR-Wrapper{X} | next-val : cap(V) > .
   rl [forward] : < O : EPR-Wrapper{X} | inside : C M >
      => < O : EPR-Wrapper{X} | inside : C > M .

   eq mte(< O : EPR-Wrapper{X} | inside : C, disp : t(T) >)
      = minimum(T, mte(C)) .
   eq tick(< O : EPR-Wrapper{X} | stress-intervals : L,
               disp : t(T), inside : C >, T')
      = < O : EPR-Wrapper{X} | stress-intervals : tick(L, T'),
               disp : tick(t(T), T'), inside : tick(C, T') > .
endtom)
```

The first few equations describe how to log the values into the `stress-intervals` attribute of a wrapper, and also how to advance to the next state taking into account the suggestions of the buffered input value. The rule `dispatch` describes how to update all the attributes inside a wrapper. The rule `recv` will buffer any messages to set the next state immediately (without processing it). Finally, the rule `forward` will forward messages in the internal configuration of sort `OutMsg` as defined in the theory `WRAPPED-OBJ`. The last few equations for `mte` and `tick` propagate these functions to the internal configuration. This is in accordance with the requirements in Theorem 4.5.15 for ensuring time robustness.

An important minor detail to notice is that the buffered suggestion for the next state is overwritten with the next state set upon dispatch. This means that, to maintain a target trajectory for the state, a controller for the state must continue to send messages to the wrapper with target values. This is important for effective operation, because if a controller disconnects, then the next suggested state is soon overwritten and no obsolete commands are stored.

4.3.5 Some Pattern Instantiations

At this point, we have fully defined the formal *Command Shaper Pattern*. Furthermore, any instantiation of the pattern has provably nice properties for safety as will be shown in Section 4.5.6. In this subsection, we

show some instantions of the pattern to a cardiac pacemaker and to an infusion pump. Once instantiated, the specifications also become executable, so we are able to use model checking to validate with the Real-Time Maude tool that, for given initial states, our specifications are indeed safe.

Pacemaker Instantiation

The pacemaker system represents a quite general application of the *Command Shaper Pattern*. It preserves the structure of the pattern without introducing any collapsing of terms or degeneracies, so it is a good test case to completely cover most constructs of the pattern.

We assume that the instantiation is customized to a specific patient, where the specific patient safety properties are as follows (since pacing periods are more naturally modelled than a pacing rate, we set the constraints on pacing periods):

1. Only pacing periods in the range between 500ms (120 bpm) and 1000ms (60 bpm) are considered valid.
2. Any pacing period below 660ms (above 90bpm) is considered stressful.
3. The pacemaker should not pace continuously at stressful rates for more than one minute.
4. Once the pacemaker's pacing rate drops down from stressful rates, the pacing rate should remain relaxed for a duration proportional to twice the previous stress interval.
5. The pacing period can be updated at most once every second.
6. An updated pacing period can increase the period by at most 30ms from the previous pacing period, or it can decrease the period by at most 20ms from the previous pacing period.

All of these pacing requirements are captured in the module `SAFE-PACEMAKER-DURATION` with each time unit representing 10ms.

```
(fmod SAFE-PACEMAKER-DURATION is
   pr STRESS-RELAX-LOG .
   --- time unit: 10ms

   op _<=risk_ : Time Time -> Bool .
   ops max-risk-dur min-risk-dur crit-dur safe-dur : -> Time .

   var D D' : Time .
   eq D <=risk D' = D ge D' . --- longer duration are less risky
```

Requirements 1 and 2 constraining the pacing periods are specified as follows:

```
 eq min-risk-dur = 100 . --- x 10ms = 60 bpm
eq safe-dur = 75 . --- x 10ms = 80 bpm
eq crit-dur = 66 . --- x 10ms = 90 bpm
eq max-risk-dur = 50 . --- x 10ms = 120 bpm
```

Requirements 5 and 6 constraining the rate of change can be specified as follows:

```
op period : -> Time .
op risk-dec-max : Time -> Time .
op risk-inc-max : Time -> Time .

eq period = 100 . --- x 10ms = 1s
eq risk-dec-max(D) = 2 . --- x 10ms = 20ms
eq risk-inc-max(D) = 3 . --- x 10ms = 30ms

op del : Time Time -> Time .
op del-inc-risk : Time Time -> Time .
op del-dec-risk : Time Time -> Time .

ceq del(D, D') = del-inc-risk(D, D') if (D <=risk D') .
eq del(D, D') = del-dec-risk(D, D') [owise] .

ceq del-inc-risk(D, D') = D' if (D - D' <= risk-inc-max(D)) .
eq del-inc-risk(D, D') = D - risk-inc-max(D) [owise] .
ceq del-dec-risk(D, D') = D' if (D' - D <= risk-dec-max(D)) .
eq del-dec-risk(D, D') = D + risk-dec-max(D) [owise] .

op ndel-min : Time Time -> Nat .
op tdel-min : Time Time -> TimeInf .

--- assumes that ndel-min recurrence terminates
--- which is the case with the current definition
```

```
--- subtle issue : could appear to converge but not terminate
eq ndel-min(D, D) = 0 .
ceq ndel-min(D, D') = s ndel-min(del-inc-risk(D, D'), D') if
   (D <=risk D') and (D <=risk del-inc-risk(D, D')) [owise] .
ceq ndel-min(D, D') = s ndel-min(del-dec-risk(D, D'), D') if
   (D' <=risk D) and (del-dec-risk(D, D') <=risk D) [owise] .

ceq tdel-min(D, D') = ndel-min(D, D') * period if
   ndel-min(D, D') :: Nat .
eq tdel-min(D, D') = INF [owise] .
```

Finally, requirements 3 and 4, which are a bit more verbose to specify due to the generality of the input theory, are shown below:

```
op max-stress-interval : -> Time .
op min-relax-interval : Time -> Time .
op safe? : Stress-Relax-Log -> Bool .
op norm : Stress-Relax-Log -> Stress-Relax-Log .

var T : Time .
eq max-stress-interval = 6000 . --- x 10ms = 1 min
eq min-relax-interval(T) = 2 * T .

vars C C' C'' : Clock .
var L : Stress-Relax-Log .

eq safe?(nil) = true .
eq safe?(E(!stress, C) L) = safe?(L) and
   value(C) <= max-stress-interval .
eq safe?(E(!relax, C) E(!stress, C') L) =
   safe?(E(!stress, C') L) .
eq safe?(E(!stress, C'') E(!relax, C) E(!stress, C') L) =
   safe?(E(!stress, C') L) and
   value(C'') <= max-stress-interval and
```

```
      value(C) >= min-relax-interval(value(C')) .

   eq norm(nil) = nil .
   ceq norm(E(!stress, C) L) = E(!stress, C) nil if
      safe?(E(!stress, C) L) .
   eq norm(L) = L [owise] .
endfm)
```

Notice that, in addition to requirements 3 and 4, we also included a definition of `norm` which will throw away all history aside from the current stress duration if everything is already safe. This makes sense, since for the defined pacemaker safety properties, there is no need to keep track of previous stress durations that the patient has already had sufficient time to recovered from.

After safety has been defined for the pacemaker, we can specify how the internal (wrapped) pacing module behaves. Notice that a module called `EXTERNAL-CONFIGURATION` is included. This is used for creating emulations, introducing sorts InExtMsg and OutExtMsg for external input and output messages respectively. These can be treated as just normal messages for the time being.

```
(omod WRAPPED-PACING-MODULE is
   pr SAFE-PACEMAKER-DURATION .
   inc EXTERNAL-CONFIGURATION .

   sort SetMsg PropMsg .
   subsort SetMsg PropMsg < InExtMsg .

   class Pacing-Module | nextPace : Timer, period : Time .

   msg set-period : Oid Time -> SetMsg .
   op pace : -> OutExtMsg .

   var O : Oid .
   var TM : Timer .
   var T T' : Time .

   eq tick(< O : Pacing-Module | nextPace : TM >, T)
```

```
      = < O : Pacing-Module | nextPace : tick(TM, T) > .
   eq mte(< O : Pacing-Module | nextPace : TM >)
      = mte(TM) .

   rl [set-period] :
      set-period(O, T) < O : Pacing-Module | period : T' >
         => < O : Pacing-Module | period : T > .
   crl [reset-next-pace] :
      < O : Pacing-Module | nextPace : TM, period : T >
         => < O : Pacing-Module | nextPace : t(T) > pace
         if TM == timer0 .
endom)
```

Notice that many of the defined parameters have names which are different from the original default names in the pattern definition. This renaming is conveniently taken care of in the *view* from our module to the input theory for the pattern.

```
(view Safe-Pacer from WRAPPED-OBJECT to WRAPPED-PACING-MODULE is
   sort Val to Time .
   sort SetMsg to SetMsg .
   sort PropMsg to PropMsg .
    sort InMsg to InExtMsg .
   sort OutMsg to OutExtMsg .

   op min-val to min-risk-dur .
   op max-val to max-risk-dur .
   op crit-val to crit-dur .
   op safe-val to safe-dur .

   class Wrapped to Pacing-Module .
   attr Wrapped . set-val to period .
   msg set-val to set-period .
endv)
```

Now that the view to the pattern's input theory is fully specified, we can finally instantiate the pattern to an executable system specification:

```
(tomod PARAM-PACEMAKER is
   pr EPR-WRAPPER-EXEC{Safe-Pacer} .
   pr PM-LEAD-SAFETY-PROP .
   pr PM-PACING-MODULE .
   pr DELAY-MSG .

   op init : -> Configuration .
   op wrapper-init : -> Object .
   op extern-init : -> Object .
   op msgs-init : -> Configuration .
   ops wrapper pacing-module lead : -> Oid .
   op shock : -> OutExtMsg .
   eq pace = shock .

   eq init = msgs-init wrapper-init extern-init .

   eq msgs-init =
      delay(set-period(pacing-module, 50), t(99))
      ... --- more delayed messages delivered at different times
      delay(set-period(pacing-module, 50), t(899)) .
   eq wrapper-init =
      < pacing-module : EPR-Wrapper{Safe-Pacer} |
         inside :
            < pacing-module : Pacing-Module | nextPace : t(0), period : safe-dur >,
         val : safe-dur, next-val : safe-dur, disp : t(period),
         stress-intervals : (nil).Event-Log{Stress-Relax} > .
   eq extern-init =
      < lead : Lead |
         shocks : (nil).Event-Log{Shock} > .
endtom)
```

An important thing to notice is that the object ID of the wrapper module is exactly the same as the internal module being wrapped. This is needed for modularity. Essentially, this allows the wrapper object to be used wherever the original (wrapped) object can be used, receiving the exact same set of input messages. The model also includes in its initial state the term `msgs-init` which is a set of initial delayed messages to simulate external input and an external environment model `extern-init` which includes the pacemaker lead delivering the shock.

Infusion Pump Instantiation

The instantiation of the infusion pump is similar to that of the pacemaker. However, continuous infusion rates can normally be abstracted into three states: *no infusion*, *base infusion*, and *bolus infusion*. For example, in patient controlled analgesia, base infusion specifies a minimal infusion rate that is safe and able to keep a resting patient reasonably sedated, and bolus infusion specifies a temporary high infusion rate to relieve immediate pain. With these state abstractions, what happens is that we obtain a degenerate view of the pattern, where different terms in the input theory actually get collapsed to a same term through the view. Of course, this is perfectly legitimate instantiation, and it shows how a generic pattern can be instantiated to a more constrained system.

We assume that the base infusion rates and bolus infusion rates are customized for each individual patient (depending on age, medical conditions, and other factors). An example safety property instantiation of an infusion pump for an analgesic (e.g. morphine sulfide) is as follows (since the timing constraints are relative, all the timing intervals have been shortened to the order of seconds to allow for faster testing during emulation):

1. Bolus doses do not last longer than 2 seconds
2. Two consecutive bolus doses must be separated by an interval of at least 8 seconds
3. There is a maximum of 3 bolus doses for any time window of one minute

The safety definition of the pump is similar to the pacemaker definition. We allow the pump state to be updated every minute and allow the pump to change states without delay between updates.

```
(fmod SAFE-PUMP is
   pr STRESS-RELAX-LOG .
   --- time unit: 1s
```

```
sort PumpState .
ops stop base bolus : -> PumpState .
op _<=risk_ : PumpState PumpState -> Bool .

var S S' : PumpState .
eq S <=risk bolus = true .
eq stop <=risk S = true .
eq base <=risk base = true .
eq S <=risk S' = false [owise] .

op period : -> Time .
eq period = 1 . --- 1s

op del : PumpState PumpState -> PumpState .
eq del(S, S') = S' .

op tdel-min : PumpState PumpState -> TimeInf .
eq tdel-min(S, S) = 0 .
ceq tdel-min(S, S') = period if S =/= S' .

op max-stress-interval : -> Time .
op min-relax-interval : -> Time .
op window-size : -> Time .
op max-in-window : -> Nat .
op safe-duration? : Stress-Relax-Log -> Bool .
op safe-freq? : Stress-Relax-Log -> Bool .
op stress-interval-window : Stress-Relax-Log Nat -> TimeInf .
op safe? : Stress-Relax-Log -> Bool .
```

The infusion pump safety properties are then defined as follows. Again, the specification is somewhat verbose due to the generality of the `safe?` operator. In this case, we must define auxiliary operators to capture the semantics of at most 3 bolus doses per minute. Furthermore, the `norm` operator is also more complicated to define, as we can discard all events that happened more than one minute ago or all stress

events aside from the three most recent.

```
op keep-latest : Stress-Relax-Log Time -> Stress-Relax-Log .
op norm : Stress-Relax-Log -> Stress-Relax-Log .

var T : Time .
eq max-stress-interval = 2 . --- 2 seconds
eq min-relax-interval = 8 . ---  seconds
eq window-size = 60 . --- 1 minute
eq max-in-window = 3 . --- 3 times per minute

vars C C' C'' : Clock .
var L : Stress-Relax-Log .

eq safe?(L) = safe-duration?(L) and safe-freq?(L) .

eq safe-duration?(nil) = true .
eq safe-duration?(E(!stress, C) nil) =
   value(C) <= max-stress-interval .
eq safe-duration?(E(!stress, C) E(!relax, C') L) =
   safe-duration?(L) and
   value(C) <= max-stress-interval and
   value(C') >= min-relax-interval .
eq safe-duration?(E(!relax, C) E(!stress, C') L) =
   safe-duration?(E(!stress, C') L) .

eq safe-freq?(nil) = true .
eq safe-freq?(E(!relax, C) L) =
   safe-freq?(L) .
eq safe-freq?(E(!stress, C') L) =
   safe-freq?(L) and
   stress-interval-window(E(!stress, C') L, max-in-window)
      >= window-size .
```

```
   var N : Nat .
   eq stress-interval-window(nil, N) = INF .
   eq stress-interval-window(E(!stress, C) L, 0) = 0 .
   eq stress-interval-window(E(!stress, C) L, s N)
      = stress-interval-window(L, N)
      + value(C) .
   eq stress-interval-window(E(!relax, C) L, N)
      = stress-interval-window(L, N)
      + value(C) .

   eq keep-latest(nil, T) = nil .
   eq keep-latest(E(!stress, C) L, 0) = nil .
   ceq keep-latest(E(!stress, C) L, T)
      = E(!stress, C) keep-latest(L, T monus value(C))
      if T > 0 .
   ceq keep-latest(E(!relax, C) L, T) = nil
      if value(C) >= T .
   ceq keep-latest(E(!relax, C) L, T)
      = E(!relax, C) keep-latest(L, T monus value(C))
      if value(C) < T .

   ceq norm(L) = keep-latest(L, window-size) if safe?(L) .
   eq norm(L) = L [owise] .
endfm)
```

The internal object definition and view definition are quite similar to the pacemaker example, so we omit them for brevity. We show the final executable system model. To simulate the external environment, we have provided a simple patient model which keeps track of the amount of chemicals present in the patient's body given the infusion rates and the patient's metabolic rates.

```
(tomod PARAM-PUMP is
   pr EPR-WRAPPER-EXEC{Safe-Pump} .
   pr DELAY-MSG .
```

```
   pr PUMP-EMU .

   op init : -> Configuration .
   op wrapper-init : -> Object .
   op extern-init : -> Object .
   op msgs-init : -> Configuration .
   ops pump-module patient : -> Oid .

   eq init = msgs-init wrapper-init extern-init .

   eq msgs-init =
      delay(set-mode(pump-module, bolus), t(9))
      ... --- more messages delivered at different times
      delay(set-mode(pump-module, stop), t(61)) .
   eq wrapper-init =
      < pump-module : EPR-Wrapper{Safe-Pump} |
         inside :
            < pump-module : Pump-Module | mode : base > base,
         val : base, next-val : base, disp : t(period),
         stress-intervals : (nil).Stress-Relax-Log > .
   eq extern-init = < patient : Patient-Sim |
      amount : 0, infusion-rate : 0, metabolic-rate : 2,
      base-rate : 1, bolus-rate : 7 > .
endtom)
```

4.4 Model Checking Completeness and Verification

4.4.1 Completeness of Compositional Nested Systems

In general, Real-Time Maude provides sound but incomplete model checking for system specifications [42]. That is, all counterexamples found will be real, but some counterexamples may not be found. However, if time advancement strategies and propositions satisfy the properties of time robustness and tick invariance (i.e., no important system states are missed due to the time advancement strategy), then the timed model

checking results are sound and complete [42].

We take a short detour to discuss the issue of completeness of model checking of timed systems in more detail. In [42], Theorem 14 provided a simple criterion for verifying that *flat* object-oriented specifications are time-robust. However, for many practical application we have *nested* or wrapped objects, in which case Theorem 14 does not apply. We provide a proof sketch of a refined Theorem in Section 4.5.7 which gives sufficient criteria for ensuring completeness of model checking in configurations with nested objects.

4.4.2 Model Checking of Instantiations

As we shall show in Section 7, our *Command Shaper Pattern* described in Section 4.2 is provably safe, so naturally, all instantiations should satisfy the necessary safety properties. However, since we already have executable instantiations available for certain medical devices, we can also use model checking as an extra level of validation for the correctness of our pattern given certain initial states. Furthermore, the safety properties described in the pattern are sometimes not the ultimate safety properties for the patient (for example, there may be delays between when the device state changes and when the change actually affects the patient), so it is also desirable to model check that these patient level safety properties are also satisfied as a consequence of the pattern for certain initial states.

Model Checking the Pacemaker

Given the module `PARAM-PACEMAKER` described in Section 4.3.5, we can immediately model check that the safety log of events in the wrapper always satisfies the defined safety properties of the pacemaker by searching for a violation of such properties.

```
Maude> (tsearch [1] in PARAM-PACEMAKER : {init} =>*
   {C:Configuration
   < pacing-module : EPR-Wrapper{Safe-Pacer} |
     A:AttributeSet, stress-intervals : L::Stress-Relax-Log >}
   such that not safe?(L::Stress-Relax-Log) in time <= 10000 .)

rewrites: 1077615 in 1379ms cpu (1419ms real) (780999 rewrites/second)

Timed search [1] in PARAM-PACEMAKER
   {init} =>* {C:Configuration
< pacing-module : EPR-Wrapper{Safe-Pacer}| A:AttributeSet, stress-intervals :
```

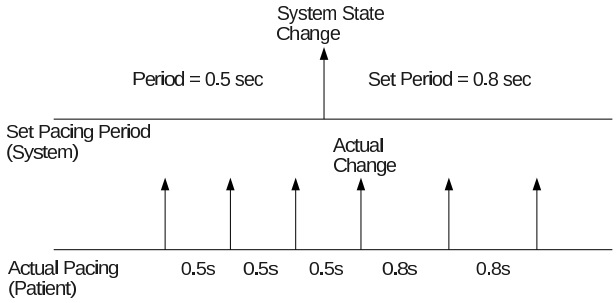

Figure 4.5: A state change in the system may not be immediately reflected in the patient. There may be some delay.

```
    L::Stress-Relax-Log >}
in time <= 10000 and with mode maximal time increase :


No solution
```

This at least tells us that the pattern instantiation does indeed perform what it is meant to do with the given initial state up to a given time bound. However, safety is defined for the patient and not for the device; so is patient safety the same as device safety? For pacemaker operation, patient safety is closely related to device safety but with some time delays as illustrated in Figure 4.5. Thus, to capture the actual events affecting the patient, we create a model of a pacemaker lead (the bioelectrical element that actually stimulates heart contractions by creating an activation potential on muscle tissue). In our case, the model of the lead is quite simple: whenever it receives a `!shock` event (from the pacing module), it will log the event. Thus, the model of the lead effectively keeps track of all the heart beats stimulated by the pacemaker.

```
(omod PM-LEAD is
   pr EVENT-LOG{Shock} .

   class Lead | shocks : Event-Log{Shock} .

   op shock : -> Msg .

   var O : Oid .
   var L : Event-Log{Shock} .
   var T : Time .
```

```
   eq tick(< O : Lead | shocks : L >, T)
      = < O : Lead | shocks : tick(L, T) > .
   eq mte(< O : Lead | >) = INF .

   rl [recv-shock] :
      shock < O : Lead | shocks : L >
      => < O : Lead | shocks : log(L, !shock) > .
endom)
```

Now, we have a lead model with a log of heart beats and the intervals between them, but it is now necessary to define patient safety in terms of heart beat intervals. These corresponding safety properties are somewhat more tedious to specify, since we are at a much lower level of abstraction, but their specification is essentially straightforward.

```
(omod PM-LEAD-SAFETY-PROP is pr PM-LEAD .
   ops min-period max-period crit-period : -> Time .
   eq max-period = 100 . --- x 10ms = 60 bpm
   eq min-period = 50 . --- x 10ms = 120 bpm
   eq crit-period = 66 . --- x 10ms = 90 bpm

   var T T' : Time .
   op stressed? : Time -> Bool .
   eq stressed?(T) = T < crit-period .

   ops dec-max inc-max : -> Time .
   eq dec-max = 3 . --- x 10ms = 20ms
   eq inc-max = 2 . --- x 10ms = 30ms

   op max-stress-dur : -> Time .
   op min-relax-dur : Time -> Time .
   eq max-stress-dur = 6000 . --- x 10ms = 1 min
   eq min-relax-dur(T) = 2 * T .

   op pm-safe? : Object ~> Bool .
```

```
op log-range-safe? : Event-Log{Shock} -> Bool .
op range-safe? : Time -> Bool .
op log-change-safe? : Event-Log{Shock} -> Bool .
op change-safe? : Time Time -> Bool .
op log-stress-safe? : Event-Log{Shock} Time Time -> Bool .
op stress-safe? : Time Time -> Bool .

var O : Oid .
var E E' : Event{Shock} .
var L : Event-Log{Shock} .

eq pm-safe?(< O : Lead | shocks : L >) =
   log-range-safe?(L) and log-change-safe?(L)
   and log-stress-safe?(L, min-relax-dur(max-stress-dur), 0) .

--- periods must be within range
eq log-range-safe?(nil) = true .
eq log-range-safe?(E L) =
   (not stopped?(E) or range-safe?(elapsed(E)))
   and log-range-safe?(L) .
eq range-safe?(T) = T >= min-period and T <= max-period .

--- periods cannot change too fast
eq log-change-safe?(nil) = true .
eq log-change-safe?(E nil) = true .
eq log-change-safe?(E E' L) =
   (not stopped?(E) or change-safe?(elapsed(E), elapsed(E')))
   and log-change-safe?(E' L) .
ceq change-safe?(T, T') = T - T' <= inc-max if T >= T' .
ceq change-safe?(T, T') = T' - T <= dec-max if T <= T' .
```

```
   --- periods cannot remain stressed too often
   eq log-stress-safe?(nil, T, T') = true .
   ceq log-stress-safe?(E L, T, T') =
      log-stress-safe?(L, T, T' + elapsed(E))
      and stress-safe?(T, T' + elapsed(E))
      if stressed?(elapsed(E)) and stopped?(E) .
   ceq log-stress-safe?(E L, T, T') =
      log-stress-safe?(L, elapsed(E), 0)
      if not (stressed?(elapsed(E)) and stopped?(E)) and T' > 0 .
   ceq log-stress-safe?(E L, T, T') =
      log-stress-safe?(L, T + elapsed(E), 0)
      if not (stressed?(elapsed(E)) and stopped?(E)) and T' == 0 .
   eq stress-safe?(T, T') =
      T >= min-relax-dur(T')
      and T' <= max-stress-dur .
endom)
```

Now, we can perform model checking using the patient safety requirements to verify that the system indeed still satisfies the true safety requirements in spite of the delays.

```
Maude> (tsearch [1] in PARAM-PACEMAKER : {init} =>*
   {C:Configuration < lead : Lead | A:AttributeSet >}
   such that not pm-safe?(< lead : Lead | A:AttributeSet >) in time <= 10000 .

rewrites: 6179398 in 4763ms cpu (5202ms real) (1297300 rewrites/second)

Timed search [1] in PARAM-PACEMAKER
   {init} =>* {C:Configuration
< lead : Lead | A:AttributeSet >}
in time <= 10000 and with mode maximal time increase :

No solution
```

The reason that safety still holds despite delays is because the wrapper pattern implicitly assumed that a delay of at most one period may be required for actuation (notice the `plus period`).

```
ceq [inev-stress] : inEv?(V, L) = safe?(L) and
   safe?(log(tick(L, tdel-min(V, crit-val) plus period), !relax))
   if stress?(V) /\ tdel-min(V, crit-val) :: Time .
```

In the pacemaker the period was 1 second, and each heart beat was at most 1 second apart (60 bpm), so the delay in pacing could not exceed 1 second.

Model Checking the Infusion Pump

As with the pacemaker example, we can immediately check that the device satisfies the predefined safety properties for a given initial state up to a certain time bound.

```
Maude> (tsearch [1] in PARAM-PUMP : {init} =>*
   {C:Configuration
   < pump-module : EPR-Wrapper{Safe-Pump} |
     A:AttributeSet, stress-intervals : L::Stress-Relax-Log >}
   such that not safe?(L::Stress-Relax-Log) in time <= 10000 .)

rewrites: 4906764 in 7239ms cpu (7285ms real) (677739 rewrites/second)

Timed search [1] in PARAM-PUMP
   {init} =>* {C:Configuration
< pump-module : EPR-Wrapper{Safe-Pump}| A:AttributeSet, stress-intervals :
    L::Stress-Relax-Log >}
in time <= 10000 and with mode maximal time increase with default 1 :

No solution
```

Again, this form of safety verification may not be convincing enough as the end safety is about the patient. Thus, we create another patient model in this case. For infusion pumps administering an analgesic, it is required that concentrations of the analgesic do not exceed certain quantities in the patient's body. This is done through a delicate balance of infusion rate and patient metabolism. The model of this is captured

by a `Patient-Sim` object which specifies the patient's metabolic rate along with various set infusion rates and keeps track of the amount of analgesic in the body.

```
(tomod PUMP-EMU is
   pr WRAPPED-PUMP-MODULE .

   class Patient-Sim |
      amount : Rat,
      infusion-rate : Rat,
      metabolic-rate : Rat,
      base-rate : Rat,
      bolus-rate : Rat .

   var O : Oid .
   vars R R' R'' : Rat .
   var T : Time .

   rl [stop-rate] :
      stop
      < O : Patient-Sim |
         infusion-rate : R >
      => < O : Patient-Sim |
         infusion-rate : 0 > .
   rl [base-rate] :
      base
      < O : Patient-Sim |
         infusion-rate : R, base-rate : R' >
      => < O : Patient-Sim |
         infusion-rate : R' > .
   rl [bolus-rate] :
      bolus
      < O : Patient-Sim |
         infusion-rate : R, bolus-rate : R' >
```

```
      => < O : Patient-Sim |
         infusion-rate : R' > .

   eq tick(< O : Patient-Sim |
      amount : R, infusion-rate : R', metabolic-rate : R'' >, T)
      = < O : Patient-Sim |
         amount : (R + R' * T) monus R'' * T > .
   eq mte(< O : Patient-Sim | >)
      = INF .
endtom)
```

The infusion pump specification in Section 4.3.5 already defines a `Patient-Sim` object with the various attributes set. In this case, we model check if the amount of analgesic in the patient can exceed 5 mg.

```
Maude> (tsearch [1] in PARAM-PUMP : {init} =>*
   {C:Configuration
   < patient : Patient-Sim |
     A:AttributeSet, amount : R:Rat >}
   such that R:Rat > 5 in time <= 10000 .)

rewrites: 4653017 in 7261ms cpu (7493ms real) (640744 rewrites/second)

Timed search [1] in PARAM-PUMP
   {init} =>* {C:Configuration
< patient : Patient-Sim | A:AttributeSet, amount : R:Rat >}
in time <= 10000 and with mode maximal time increase with default 1 :

No solution
```

However if the threshold were to be 4 mg, then model checking shows that it is possible for a sequence of bolus doses to overdose the patient.

```
Maude> (tsearch [1] in PARAM-PUMP : {init} =>*
   {C:Configuration
   < patient : Patient-Sim |
```

```
     A:AttributeSet, amount : R:Rat >}
   such that R:Rat > 4 in time <= 10000 .)

rewrites: 117481 in 292ms cpu (296ms real) (401020 rewrites/second)

Timed search [1] in PARAM-PUMP
   {init} =>* {C:Configuration
< patient : Patient-Sim | A:AttributeSet, amount : R:Rat >}
in time <= 10000 and with mode maximal time increase with default 1 :

Solution 1
A:AttributeSet --> base-rate : 1, bolus-rate : 7, infusion-rate : 7,
    metabolic-rate : 2 ; C:Configuration --> delay(set-mode(pump-module,bolus),
    t(1))
delay(set-mode(pump-module,bolus),t(2))
delay(set-mode(pump-module,bolus),t(3))
delay(set-mode(pump-module,bolus),t(10))
delay(set-mode(pump-module,bolus),t(20))
delay(set-mode(pump-module,bolus),t(40))
delay(set-mode(pump-module,bolus),t(50))
delay(set-mode(pump-module,stop),t(51))
< pump-module : EPR-Wrapper{Safe-Pump}| disp : t(0), inside : < pump-module :
    Pump-Module | mode : bolus >, next-val : bolus, stress-intervals : E(
    !stress,c(1,run))nil, val : bolus > ; CLASS_OF_patient:Patient-Sim -->
    Patient-Sim ; R:Rat --> 5 ; TIME_ELAPSED:Time --> 10
```

This just goes to show that even though the *Command Shaper Pattern* will satisfy the defined safety properties for a device, the device's safety properties do not immediately ensure safety of the patient if the appropriate patient context is not taken into account. In this case, the patient is overly sensitive to the analgesic, so either bolus doses should be disallowed, or lower infusion rates should be set for bolus doses.

4.5 Correctness of the Command Shaper Pattern

In this section, we proceed to prove the correctness of the command shaper. We start by showing that the our equational definition of `safe?` is sensible as a safety property, and we also take a slight detour to prove that normalization operations on event logs will not change the evaluation of safety. After we have proved these properties about our definition of safety, we then show that system execution defined by the command shaper does indeed satisfy the preserve the safety invariant using induction on possible execution steps (i.e. rewrite rules).

4.5.1 Stability of the Safety Property

Lemma 4.5.1. *Consider any term L of sort* `Stress-Relax-Log`*, any term E_{sr} of sort* `SREvent`*, and any term T of sort* `Time`*.*

If $\texttt{safe?}(L) = \texttt{false}$*, then it must be the case that:*

1. $\texttt{safe?}(\texttt{log}(L, E_{sr})) = \texttt{false}$*, and*
2. $\texttt{safe?}(\texttt{tick}(L, T)) = \texttt{false}$*.*

Proof. For ease of readability, we use "`;`" to denote the concatenation of events in event logs.

$\texttt{safe?}(\texttt{nil}) = \texttt{true}$ by the equational axiom `initially-safe` in the theory `SAFE-STATE`, so we only need to consider a nonempty list L.

For the first condition, if $\texttt{safe?}(\texttt{log}(L, E_{sr})) = \texttt{true}$, then

$\texttt{safe?}(E(E_{sr}, c(0)); \texttt{stop}(L)) = \texttt{true}$ (by using the equational definition of `log`)

$\texttt{safe?}(\texttt{stop}(L)) = \texttt{true}$ (by the equational axiom `sub-safe` in `SAFE-STATE`)

$\texttt{safe?}(L) = \texttt{true}$ (by the axiom `stopped-safe` in `SAFE-STATE`), a contradiction.

For the second condition, there are two possibilities for nonempty L. For some T' of sort `Time`, and some L' or sort `Stress-Relax-Log`, either $L = \texttt{E}(!\texttt{stress}, T'); L'$ or $L = \texttt{E}(!\texttt{relax}, T'); L'$. If $L = \texttt{E}(!\texttt{stress}, T'); L' = \texttt{false}$, then $T' + T \geq T'$ (using a linear time model), so

$\texttt{safe?}(\texttt{tick}(L, T)) = \texttt{safe?}(\texttt{E}(!\texttt{stress}, T' + T); L') = \texttt{false}$ (by equation `stress-safe`)

On the other hand, if $L = \texttt{E}(!\texttt{relax}, T'); L'$, then

$\texttt{safe?}(\texttt{tick}(L, T)) = \texttt{safe?}(\texttt{E}(!\texttt{relax}, T' + T); L')$ (by definition of `tick`)

$= \texttt{safe?}(L') = \texttt{safe?}(\texttt{E}(!\texttt{relax}, T'); L')$ (by two applications of the equation `relax-safe`).

Showing that $\texttt{safe?}(\texttt{tick}(L, T)) = \texttt{safe?}(L) = \texttt{false}$ in this case. □

Lemma 4.5.1 shows that during execution where system transitions only modify a `Stress-Relax-Log` by applications of the operators `log` and `tick`, it is sufficient to check that the `safe?` predicate holds over a countable number of time instances S_t such that for any time t there is a time $t' \in S_t$ such that $t' > t$. This also shows the tick stabilizing property for the predicate `safe?`. Also, by Theorem 4.5.15, assuming that we have defined our system to satisfy all the preconditions, our system should also be time robust. This means that all untimed model checking on the predicate `safe?` of a system is both sound and complete.

4.5.2 Safety Preserving Property of the norm Operator

Next we consider the `norm` operation. We show that applying `norm` cannot affect the satisfaction of `safe`.

Notation. In this section, we rely heavily on reasoning with subterms and term substitution, so it is appropriate for us to introduce some notation to make discussions more compact.

Given terms t and u and a position p, we write t_p for the subterm at position p of t, and we write $t[u]_p$ for the term t where the subterm at position p is replaced by u. For position p, we use the conventional representation where p is a string of natural numbers corresponding to branch traversal of the tree representation of a term.

For example, if $t = h(f(x, g(x, y)))$ and $u = z$ and $p = 1.2$, then $t_p = g(x, y)$, and $t[u]_p = h(f(x, z))$.

Definition. An **unreduced log** is a term t of sort `Stress-Relax-Log` where for all positions p in t, the subterm $t' = t_p$ either has the form

1. $t' =$ `nil` or
2. $t' =$ `log`(L, E_{sr}) or
3. $t' =$ `tick`(L, T)

where subterm L in (2) and (3) is of sort `Stress-Relax-Log`, E_{sr} of sort `SREvent`, and T of sort `Time`. Intuitively, t is constructed from only using the `nil`, `log`, and `tick` operators.

Furthermore, we define an **unreduced normalized log** to be a term t of sort `Stress-Relax-Log`, the subterm $t' = t_p$ can have the form $t' = norm(L)$ in addition to all the subterm forms for an unreduced log.

Lemma 4.5.2. *If L is a term of* `Stress-Relax-Log`*, then* $tick(tick(L, T), T') = tick(L, T + T')$.

Proof. Straightforward using structural induction on L. □

Lemma 4.5.3. *If L, L', L'' are terms of sort* `Event-Log{X}` *with $L \neq$* `nil`*, and* $append(L, L')$ *is of sort* `Event-Log{X}`*, then*

1. $\texttt{tick}(\texttt{append}(L, L')) = \texttt{append}(\texttt{tick}(L), L')$

2. $\texttt{stop}(\texttt{append}(L, L')) = \texttt{append}(\texttt{stop}(L), L')$

3. $\texttt{append}(\texttt{append}(L, L'), L'') = \texttt{append}(L, \texttt{append}(L', L''))$

Proof. Straightforward using structural induction on L and applying equational definitions of the operators.

□

Lemma 4.5.4. *Let t be an unreduced log. For any position p in t, with the subterm $t' = t_p$, we have one of the following equalities:*

1. $t = \texttt{tick}(t', T)$ *or*

2. $t = \texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T)))$.

for some nonempty log L of sort $\texttt{Log}\{\texttt{SREvent}\}$ *and some T of sort* `Time`.

Proof. We proceed by induction on the lenght of the position p as a string.

When $p = nil$, the topmost postion, $t' = t$, and we trivially have $t = \texttt{tick}(t, 0)$ (the first form of the equivalence).

Now, assuming the statement holds for positions of length n. For p of length $n + 1$, it must be of the form $p = q.r$. We need to consider two cases:

If $t = \texttt{tick}(t'', T)$, then $t' = t''_r$. To use the inductive hypothesis, we must consider two subcases:

If $t'' = \texttt{tick}(t', T')$, then $t = \texttt{tick}(t'', T) = \texttt{tick}(\texttt{tick}(t'), T), T') = \texttt{tick}(t', T + T')$ (Lemma 4.5.2).

If $t'' = \texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T')))$, then $t = \texttt{tick}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T'))), T)$
$= \texttt{append}(\texttt{tick}(L, T), \texttt{stop}(\texttt{tick}(t', T'))) = \texttt{append}(L', \texttt{stop}(\texttt{tick}(t', T')))$ (using Lemma 4.5.3 and making the substitution $L' = \texttt{tick}(L, T)$).

Now consider the case when $t = \texttt{log}(t'', E_{sr})$. Again we have $t' = t''_r$, and we must consider two subcases for t'':

If $t'' = \texttt{tick}(t', T')$, then $t = \texttt{log}(\texttt{tick}(t', T'), E_{sr}) = \texttt{E}(E_{sr}, \texttt{c}(0)); \texttt{stop}(\texttt{tick}(t', T'))$
$= \texttt{append}(\texttt{E}(E_{sr}, \texttt{c}(0)); \texttt{nil}, \texttt{stop}(\texttt{tick}(t', T')) = \texttt{append}(L', \texttt{stop}(\texttt{tick}(t', T')))$.

If $t'' = \texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T')))$, then $t = \texttt{log}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T'))), E_{sr})$
$= E; \texttt{stop}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T')))) = E; \texttt{append}(\texttt{stop}(L), \texttt{stop}(\texttt{tick}(t', T')))$
$= \texttt{append}(E; \texttt{stop}(L), \texttt{stop}(\texttt{tick}(t', T'))) = \texttt{append}(L', \texttt{stop}(\texttt{tick}(t', T')))$
(applying Lemma 4.5.3 multiple times).

Thus, we have proved the inductive step considering all cases.

□

Definition. If t is an unreduced log, then a **subnormalized log** from t is a term t_p^{norm} s.t. the subterm $t' = t_p$ is replaced by $\texttt{norm}(t')$. I.e., $t_p^{norm} = t[\texttt{norm}(t_p)]_p$.

Lemma 4.5.5. *Let t be an unreduced log, and t_p^{norm} its subnormalized log. We have the following equivalence:*

$\texttt{norm}(t_p^{norm}) = \texttt{norm}(t)$

Proof. We have shown in Lemma 4.5.4, for t with a subterm $t' = t_p$, t has one of two possible forms.

If $t = \texttt{tick}(t', T)$, then $t_p^{norm} = \texttt{tick}(\texttt{norm}(t'), T)$. By the equation `norm-tick`, we immediately have, $\texttt{norm}(t_p^{norm}) = \texttt{norm}(\texttt{tick}(\texttt{norm}(t'), T)) = \texttt{norm}(\texttt{tick}(\texttt{norm}(t'), T)) = \texttt{norm}(\texttt{tick}(t', T)) = \texttt{norm}(t)$.

If $t = \texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T)))$, then

$t_p^{norm} = \texttt{append}(L, \texttt{stop}(\texttt{tick}(\texttt{norm}(t'), T)))$.

By the equation `norm-app`, we immediately have,

$\texttt{norm}(t_p^{norm}) = \texttt{norm}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(\texttt{norm}(t'), T))))$

$= \texttt{norm}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(\texttt{norm}(t'), T)))) = \texttt{norm}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(t', T)))) = \texttt{norm}(t)$ □

Definition. For an unreduced normalized log, define the function $strip_{norm}$ recursively as follows.

1. $strip_{norm}(\texttt{nil}) = \texttt{nil}$
2. $strip_{norm}(\texttt{log}(L, E_{sr})) = \texttt{log}(strip_{norm}(L), E_{sr})$
3. $strip_{norm}(\texttt{tick}(L, T) = \texttt{tick}(strip_{norm}(L), T)$
4. $strip_{norm}(\texttt{norm}(L)) = strip_{norm}(L)$

Thus, for an unreduced normalized log t, $strip_{norm}(t)$ is an unreduced log that removes all the applications of `norm` from t.

Lemma 4.5.6. *Let t be an unreduced normalized log with the topmost application of* `norm` *at position p, then $strip_{norm}(t) = t[strip_{norm}(t_p)]_p$.*

Proof. Straight forward by induction on the depth of p and considering all possible forms of the top-most term. □

Lemma 4.5.7. *Let t be an unreduced normalized log, then* $\texttt{norm}(strip_{norm}(t)) = \texttt{norm}(t)$.

Proof. We proceed by induction on the number of `norm` operators present in t.

If t has no `norm` operators, then of course, $strip_{norm}(t) = t$.

Assume that the statement holds when for any term with n `norm` operators in subterms. If t has $n+1$ `norm` operators, then consider the topmost `norm` operator position p. So $t' = t_p = \texttt{norm}(t'')$, and by the inductive hypothesis, t'' has n `norm` operators, so $t' = \texttt{norm}(\mathit{strip}_{norm}(t'')) = \texttt{norm}(t'')$.

Now using Lemma 4.5.6, we can define $u = t[t'']_p = \mathit{strip}_{norm}(t)$.

We have that $u_p^{norm} = t[\texttt{norm}(\mathit{strip}_{norm}(t''))]_p$ is a subnormalized log, and we have that $\texttt{norm}(u_p^{norm}) = \texttt{norm}(u)$ by Lemma 4.5.5.

Thus, summarizing everything, we have

$\texttt{norm}(\mathit{strip}_{norm}(t)) = \texttt{norm}(u) = \texttt{norm}(u_p^{norm})$

$= \texttt{norm}(t[\texttt{norm}(\mathit{strip}_{norm}(t''))]_p) = \texttt{norm}(t[\texttt{norm}(t'')]_p) = \texttt{norm}(t)$. □

Theorem 4.5.8. *Let t be an unreduced normalized log, then* $\texttt{safe?}(t) = \texttt{safe?}(\mathit{strip}_{norm}(t))$.

Proof. Consider the position p of the topmost `norm` operator in t, and let $t' = t_p = \texttt{norm}(t'')$. Now define, $u = t[\mathit{strip}_{norm}(t'')]_p = \mathit{strip}_{norm}(t)$, and using Lemma 4.5.7, we have $u_p^{norm} = t[\texttt{norm}(\mathit{strip}_{norm}(t''))]_p = t[\texttt{norm}]_(t'')p = t$.

For $u' = u_p = \mathit{strip}_{norm}(t'')$, we know from Lemma 4.5.4, that u has one of two forms.

For the first case, if $u = \texttt{tick}(u', T)$, then by application of equation `norm-tick-safe` in the `SAFE-STATE` theory

$\texttt{safe?}(\mathit{strip}_{norm}(t)) = \texttt{safe?}(u) = \texttt{safe?}(\texttt{tick}(\mathit{strip}_{norm}(t''), T)) = \texttt{safe?}(\texttt{tick}(\mathit{strip}_{norm}(t''), T))$

$= \texttt{safe?}(\texttt{tick}(\texttt{norm}(\mathit{strip}_{norm}(t'')), T)) = \texttt{safe?}(u_p^{norm}) = \texttt{safe?}(t)$.

For the second case, if

$u = \texttt{append}(L, \texttt{stop}(\texttt{tick}(u', T)))$, then by application of the equation `norm-app-safe` in the `SAFE-STATE` theory

$\texttt{safe?}(\mathit{strip}_{norm}(t)) = \texttt{safe?}(u) = \texttt{safe?}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(\mathit{strip}_{norm}(t''), T))))$

$= \texttt{safe?}(\texttt{append}(L, \texttt{stop}(\texttt{tick}(\texttt{norm}(\mathit{strip}_{norm}(t'')), T)))) = \texttt{safe?}(u_p^{norm}) = \texttt{safe?}(t)$.

□

We have shown that contracting or normalizing a log based on the requirements of the `norm` operation cannot affect the result of the `safe?` predicate. This means that for the remainder of the section, we can essentially "ignore" applications of `norm`, and assume that an unbounded length list is maintained even though only a bounded length list is maintained for efficient implementation.

4.5.3 Persistence of Safety using Safety Envelopes

Definition. Consider a term L of sort `Stress-Relax-Log` and a term V of sort `Val`. L is said to **faithfully follow** V iff

1. when $V \leq v_{crit}$, we have $L = \texttt{E}(\texttt{!relax}, T); L'$, and
2. when $V > v_{crit}$, we have $L = \texttt{E}(\texttt{!stress}, T); L'$.

Here, L' is a term or soft `Stress-Relax-Log`, and T is a term of sort `Time`.

An pair (V, L) that satisfies this definition is called a **sensible pair**.

Lemma 4.5.9. *If (V, L) is a sensible pair with $V > v_{crit}$, and V' is of sort* `Val` *with $V' < v_{crit}$, then* $\texttt{inEnv?}(\Delta(V, v_{crit}), \texttt{tick}(L, \texttt{period})) = \texttt{true} \Leftrightarrow \texttt{inEnv?}(\Delta(V, V'), \texttt{tick}(L, \texttt{period})) = \texttt{true}$.

Proof. Notice that by the axioms `del-mono`, we have $\Delta(V, V') \leq \Delta(V, v_{crit})$.

If $\Delta(V, v_{crit}) \neq v_{crit}$, then by `del-max` we have $\Delta(V, v_{crit} = \Delta(V, V')$, from which the implication trivially follows because the left hand sides of the two equations are then equal. This means that we only need to consider the case where $\Delta(V, v_{crit}) = v_{crit}$. In this case, we have $\Delta(V, V') \leq \Delta(V, v_{crit}) = v_{crit}$, so the only conditional equational axiom that can be applied is `inev-relax`, which again trivials guarantees the equivalence by:

$$\texttt{inEnv?}(\Delta(V, V'), \texttt{tick}(L, \texttt{period})) = \texttt{safe?}(\texttt{tick}(L, \texttt{period})) = \texttt{inEnv?}(\Delta(V, v_{crit}), \texttt{tick}(L, \texttt{period}))$$

□

Lemma 4.5.10. *Let (V, L) be a sensible pair with $V > v_{crit}$ and* $\texttt{inEnv?}(V, L) = \texttt{true}$. *If $V' \leq v_{crit}$, then* $\texttt{inEnv?}(\Delta(V, V'), \texttt{tick}(L, \texttt{period})) = \texttt{true}$.

Proof. Since $\texttt{inEnv?}(V, L) = \texttt{true}$ and V is a stress state, it must be the case that equation `inev-stress` was applied. This of course means that

$\texttt{safe?}(L) = \texttt{true}$ and $\texttt{safe?}(\texttt{log}(\texttt{tick}(L, t_{\Delta min}(V, v_{crit}) + \texttt{period}), \texttt{!relax})) = \texttt{true}$, and $t_{\Delta min}$ is finite.

By Lemma 4.5.9, it is sufficient to just consider the case where $V' = v_{crit}$.

We first show the simple property $\texttt{safe?}(\texttt{tick}(L, \texttt{period})) = \texttt{true}$. From Lemma 4.5.1, we directly have:

$$\begin{aligned}
&\texttt{safe?}(\texttt{log}(\texttt{tick}(L, t_{\Delta min}(V, v_{crit}) + \texttt{period}), \texttt{!relax})) = \texttt{true} \\
&\qquad \Rightarrow \texttt{safe?}(\texttt{tick}(L, t_{\Delta min}(V, v_{crit}) + \texttt{period})) \\
&= \texttt{safe?}(\texttt{tick}(\texttt{tick}(L, \texttt{period}), t_{\Delta min}(V, v_{crit}))) = \texttt{true} \\
&\qquad \Rightarrow \texttt{safe?}(\texttt{tick}(L, \texttt{period})) = \texttt{true}
\end{aligned}$$

By the inductive definition of `tdel-min`, $t_{\Delta min}$, by equation `ind-tdel`, we have $t_{\Delta min}(\Delta(V, v_{crit}), v_{crit}) +$ `period` $= t_{\Delta min}(V, v_{crit})$ (recall that $V > v_{crit}$ by definition of a stressed state). Since $t_{\Delta min}(V, v_{crit})$ is finite, $t_{\Delta min}(\Delta(V, v_{crit}), v_{crit})$ must also be finite. Furthermore, we have:

$$\begin{aligned}
&\texttt{safe?}(\texttt{log}(\texttt{tick}(\texttt{tick}(L, \texttt{period}), t_{\Delta min}(\Delta(V, v_{crit}), v_{crit}) + \texttt{period}), \texttt{!relax})) \\
&\quad = \texttt{safe?}(\texttt{log}(\texttt{tick}(\texttt{tick}(L, \texttt{period}), t_{\Delta min}(V, v_{crit})), \texttt{!relax})) \\
&\quad = \texttt{safe?}(\texttt{log}(\texttt{tick}(L, t_{\Delta min}(V, v_{crit})) + \texttt{period}, \texttt{!relax})) = \texttt{true}.
\end{aligned}$$

We have shown that all conditions for $\texttt{inEnv?}(\Delta(V, v_{crit}), \texttt{tick}(L, \mathit{period}))$ evalutate to `true`.

□

4.5.4 System Execution Assumptions

Correspondence Between Tick Rules and System Time Elapse

We model all time-evolving behavior in a system by terms of sort `Timer` and `Clock`. It is necessary to show that these entities indeed behave as their intuitive names suggest. This means that when time elapse occurs in Real-Time Maude, the corresponding time should decrease in a timer or increase in a non-stopped clock. We only have one timed rewrite rule in our system:

```
crl [advance] : {C} => {tick(C, T)} in time T if T le mte(C) [nonexec] .
```

This means that (deterministic) time evolution of the system is fully captured by the `tick` operator. It is assumed that all timed values in the system are affected by rewrites corresponding to the advancement of the system time in Real-Time Maude.

4.5.5 Wrapper Dispatch and Well-Behaved Rule Applications

After knowing that timers are faithfully represented, we show that certain system behaviors may only occur periodically with fixed period durations.

Definition. A **sensible configuration** is a term C of sort `Configuration`, where any top-level object O in C has a unique oid (object identifier).

For a sensible configuration, we define $O_{id}(C)$ to denote the unique subterm, if it exists, of C representing the object having id as its oid. If a top-level object of id does not exist, then we say that $O_{id}(C)$ is undefined.

Furthermore, if a is an attribute of an object with oid id, and given a configuration C, then we write $O_{id}.a(C)$ to denote the value of attribute a in the object instance $O_{id}(C)$ when it is defined.

For an object with oid w of class `EPR-Wrapper`, we use the following conventions when referencing attributes of O_w:

- $O_w.disp$ represents the timer in attribute `dispatch`
- $O_w.next$ represents the value of attribute `next-val`
- $O_w.val$ represents the current value of attribute `val`
- $O_w.log$ represents the stress-relax log of attribute `stress-intervals`
- $O_w.C_{in}$ represents the internal configuration of attribute `inside`

We will also write $O_w.Static$ to refer to the set of attributes $\{O_w.disp, O_w.next, O_w.val, O_w.log\}$ (not including $O_w.C_{in}$).

Notice that, since we sometimes have nested objects, some object attributes may be configurations, so if *pacemaker* is the oid of an object in $O_w.C_{in}$, then we can write $O_{pacemaker}.rate(O_w.C_{in}(C))$ to unambiguously reference the set rate of a pacemaker in system $\{C\}$. For the internal wrapped class with oid v, we use $O_v.val$ to denote the value of attribute of `set-val`.

Furthermore, for a one-step rewrite $C \to C'$, when $O_{id}.a(C) = O_{id}.a(C')$, we say that $O_{id}.a$ *does not change* under $C \to C'$. Also, for a rewrite rule $l : t \to t'$, we say that $O_{id}.a$ *does not change* under l if for any configuration C, such that l has a rewrite $C \to C'$, then $O_{id}.a(C) = O_{id}.a(C')$.

Definition. Fix an oid w for an object of sort `EPR-WRAPPER`.

An w-**execution state** (or just execution state) is a term $\{C\}$ of sort `System` such that C is a sensible configuration and $O_w(C)$ is defined. An **initial state** is an execution state $\{C\}$ s.t. $v_{min} \leq O_w.val(C) \leq_{risk} v_{crit}$ and $O_w.log(C) = \texttt{nil}$ and $v_{min} \leq_{risk} O_w.next(C) \leq_{risk} v_{max}$.

We define a corresponding proposition **init** (implicitly parameterized on some fixed w):

$\{C\} \models init \iff (v_{min} \leq O_w.val(C) \leq_{risk} v_{crit}) \wedge (O_w.log(C) = \texttt{nil}) \wedge (valid(O_w.next(C)))$

Definition. Fix an oid w for an object of sort `EPR-WRAPPER`.

A w-**sensible transition system** (or just sensible transition system) is a transition system on w-execution states such that if a one-step transition δ exists from execution states $\{C\} \to \{C'\}$, then either:

1. $O_w.static(C) = O_w.static(C')$, or
2. $C' = tick(C, T)$ where $T \leq mte(C)$, or
3. $O_w(C) \to O_w(C')$ is a one-step rewrite applying the rule `dispatch`, or

4. $C \to C'$ is a one-step rewrite applying the rule `recv`.

In case the transition δ has $C' = tick(C, T)$ with $T > 0$, then we say that δ is a timed transition with time elapse $te(\delta) = T$.

A rewriting logic system module is also said to be a w-**sensible system module** if it's induced transition system on the sort `System` is an w-sensible transition system.

4.5.6 Correctness of the Command Shaper Pattern

From our earlier discussions, safety of medical devices boils down to three important properties that we must check:

1. The state must remain in some valid risk range between v_{min} and v_{max};
2. The state must satisfy the rate of change requirements. That is, for any state V to change to a state V', we can change by at most $\Delta(V, V')$ in one operation period;
3. The states cannot remain stressed for too long. That is for a log of stress and relax events a predicate `safe?` satisfying monotonicity properties must hold for the entire execution of the system (assuming some initial safety properties).

We start with the first property. For notational compactness, we will use $\leq$ in place of $\leq_{risk}$.

Definition. A state v is called *valid* if $v_{min} \leq v \leq v_{max}$.

Theorem 4.5.11. *For an w-sensible system module, any execution state $\{C\}$ reachable from an initial state has both $O_w.val(C)$ and $O_w.next(C)$ valid.*

Proof. We prove this by induction on the number of transitions (i.e., rule applications in our case).

$O_w.val(C)$ and $O_w.next(C)$ are clearly both valid for $\{C\}$ being an initial state.

For the inductive step, suppose that $O_w.val(C')$ and $O_w.next(C')$ are both valid for an execution state $\{C'\}$, and $\{C'\} \to \{C\}$ via a valid transition. Of the three possible cases for transitions in sensible transition systems:

The two easy cases, if $C = tick(C')$ or $O_w(C) = O_w(C')$, then $O_w.static(C) = O_w.static(C')$, and the statement holds.

If the transition occurs as a result of applying the rule `dispatch`, then by the form of the rhs of the rule, we either have $O_w.val(C) = O_w.next(C) = \Delta(O_w.val(C'), O_w.next(C'))$ or $O_w.val(C) = O_w.next(C) =$

$\Delta(O_w.val(C'), v_{safe})$. It is clear from applications of the equation `del-bound` in theory `SAFE-STATE` that both $\Delta(O_w.val(C'), O_w.next(C'))$ and $\Delta(O_w.val(C'), v_{safe})$ provide valid values.

If the transition occurs as a result of applying the rule `recv`, then of course $O_w.val(C) = O_w.val(C')$. Also, by the form of the rhs of the rule, $O_w.next(C) = cap(v)$ for some term v of sort `Val`. $cap(v)$ will always be a valid value.

□

Next we tackle the property that the values satisfy the rate of change requirements. Whatever the Δ function defines, we must make sure a wrapped system will not change its state faster than this per period of operation.

Definition. In an w-sensible transition system, a state $\{C'\}$ is reachable from $\{C\}$ in τ time units denoted $\{C\} \xrightarrow{\tau} \{C'\}$, if there is path in the underlying transition $\delta_1 \ldots \delta_n$ such that the timed transitions $\delta_{i_1} \ldots \delta_{i_m}$ in this path satisfy the equality $\tau = te(\delta_{i_1}) + \ldots + te(\delta_{i_m})$.

Theorem 4.5.12. *For an w-sensible system module, then if* $\{C\} \xrightarrow{\tau} \{C'\}$ *with* $\tau <$ `period`*, it must be the case that* $O_w.val(C') = \Delta(O_w.val(C), v')$ *for some value* v'*.*

Proof. If $O_w.val$ was unchanged, then of course we have $O_w.val(C') = \Delta(O_w.val(C), O_w.val(C))$.

The value of $O_w.val$ only changes by the *dispatch* rule. This is only applied when the attribute $O_w.disp$ is $t(0)$, and this same rule resets the timer to be $t(period)$. Since no other rules can modify $O_w.disp$ aside from the time advancement rules which decrements the timer by the amount of real-time elapsed, after each change of $O_w.val$, the time advancement rules must advance time by *period* time units before changing again. This shows that the intervals between updates of the attribute $O_w.disp$ is at exactly *period* time units.

Furthermore, an application of the *dispatch* rule will reset $O_w.val$ to either $\Delta(O_w.val, O_w.nextval)$ or $\Delta(O_w.val, v_{safe})$ each of these expressions satisfy the Δ requirement by definition.

□

Now, we get to the most important and non-trivial theorem that says that the system will never remain in a stressed state for too long to be considered unsafe (based on the definition of the `safe?` predicate). We first define the following auxiliary predicates on a system.

Definition. For an w-sensible system module, define the proposition $safe_{hist}$ (implicitly parameterized on w) as follows:

$\{C\} \models safe_{hist} \iff$ `safe?`$(O_w.log(C))$

Lemma 4.5.13. *For an w-sensible system module, consider a state* $\{C\}$*. Let* $V = O_w.val(C)$ *and* $L = O_w.log(C)$*, and let* (V, L) *be a sensible pair. Furthermore, let* $\texttt{inEnv?}(V, L) = \texttt{true}$*, and* $\texttt{mte}(O_w.disp(C)) = \texttt{period}$*. If* $\{C\}$ *reaches* $\{C''\}$ *in* `period` *time with any application of the* `dispatch` *rule, and if* $\{C''\} \to \{C'\}$ *with an application of the* `dispatch` *rule, then with* $V' = O_w.val(C')$ *and* $L' = O_w.log(C')$*, we have that* (V', L') *is a sensible pair, and* $\texttt{inEnv?}(V', L') = \texttt{true}$*.*

Proof. We show $\texttt{inEnv?}(V', L') = \texttt{true}$ by case analysis.

Recall that $\texttt{stress?}(v) = \texttt{true} \Leftrightarrow v >_{risk} v_{crit}$, and $\texttt{relax?}(v) \Leftrightarrow not(\texttt{stress?}(v))$.

Case 1: $\texttt{relax?}(V) = \texttt{true}$ and $\texttt{relax?}(V') = \texttt{true}$.

Because V is a relaxed state, we must have either $L = \texttt{nil}$ or $L = \texttt{E}(!\texttt{relax}, c(T)); L''$ for some time T and log L''. Since V' is also a relaxed state, no new event is logged during the application of the `dispatch` rule, and $L' = \texttt{tick}(L, \texttt{period})$ due to the accumulation of timed rules with tick applications. Thus, (V', L') is still a sensible pair. Notice that, since V is a relaxed state, we must have applied the equation *inev-relax* to evaluate to `true`, or $\texttt{safe?}(L) = \texttt{true}$. Now we have:

$$inEnv?(V', L') = \texttt{safe?}(L') = \texttt{safe?}(\texttt{tick}(L, \texttt{period})) = \texttt{safe?}(\texttt{tick}(\texttt{E}(!\texttt{relax}, c(T)); L'', \texttt{period}))$$
$$= \texttt{safe?}(\texttt{E}(!\texttt{relax}, c(T + \texttt{period})); L'') = \texttt{safe?}(L'') = \texttt{safe?}(E(!relax, c(T); L)) = \texttt{safe?}(L) = \texttt{true}.$$

Here we used the equations *relax-safe* to show the equivalence to $\texttt{safe?}(L'')$.

Case 2: $\texttt{stress?}(V) = \texttt{true}$ and $\texttt{stress?}(V') = \texttt{true}$.

The initial and final states are both stressed, so again the log does not add any new events. (V', L') is again a sensible pair. In this case, $L' = \texttt{tick}(L, \texttt{period})$. Now, we have $V' = \Delta(V, O_w.next)$, if $O_w.next \leq v_{crit}$, then we directly use Lemma 4.5.10 to conclude that $inEnv?(V', L') = \texttt{true}$. On the other hand, if we have $O_w.nextval > v_{crit}$, then it must be the case that $\texttt{inEnv?}(V', L') = \texttt{true}$; otherwise, the set value would be $v_{safe} \leq v_{crit}$.

Case 3: $\texttt{stress?}(V) = \texttt{true}$ and $\texttt{relax?}(V') = \texttt{true}$.

The final state becomes relaxed. This is almost a directl consequence of Lemma 4.5.10, but we have to allow for the extra event being added to the log. In this case, $L' = \texttt{E}(!\texttt{relax}, \texttt{c}(0); \texttt{stop}(\texttt{tick}(L, period))$. Again (V', L') is a sensible pair. We know from Lemma 4.5.10 that $inEnv?(V', tick(L, period)) = \texttt{true}$. This transforms into our desired result by the following equalities (using applications of equations *relax-safe*

and *stopped-safe*):

$$\texttt{inEnv?}(V', L') = \texttt{inEnv?}(V', \texttt{E}(!\texttt{relax}, \texttt{c}(0); \texttt{stop}(\texttt{tick}(L, \texttt{period}))))$$

$$\texttt{inEnv?}(V', \texttt{stop}(\texttt{tick}(L, \texttt{period}))) = \texttt{inEnv?}(V', \texttt{tick}(L, \texttt{period})) = \texttt{true}.$$

Case 4: $\texttt{relax?}(V) = \texttt{true}$ and $\texttt{stress?}(V') = \texttt{true}$.

In this last case, we must reason about the conditional fragments on equations that lead to the stressed state in the first place. If the initial state was relaxed and the final state is stressed, it must be the case that $\texttt{inEnv?}(V', \texttt{log}(\texttt{tick}(L, \texttt{period}), !\texttt{stress}))$ evaluated to $\texttt{true}$ before an application of $\texttt{dispatch}$, otherwise we would end up in the final state $V' = \Delta(V, v_{safe})$, which would not be stressed. However, after an application of $\texttt{dispatch}$; we would have set $L' = \texttt{log}(\texttt{tick}(L, \texttt{period}), !\texttt{stress})$. This means that we conditionally checked that transitioning to a stressed state would satisfy `inEnv?` before taking the transition, and thus, trivially by just looking ahead, $\texttt{inEnv?}(V', L') = \texttt{true}$. Of course, (V', L') is still a sensible pair in this last case. □

Theorem 4.5.14. *In a w-sensible system module, if $\{C\}$ initially satisfies predicates init and $safe_{hist}$, then the future states will always satisfy $safe_{hist}$. More precisely:*

$\{C\} \models (init \wedge safe_{hist}) \rightarrow \Box safe_{hist}$.

Proof. As always, we proceed by induction on the number of application of rewrite rules applied to C to prove $\Box safe_{hist}$. The base case is trivial as it is part of the hypothesis.

For one transition $C \rightarrow C'$. The only rules that can make $O_w.log(C')$ different from $O_w.log(C)$ are $\texttt{dispatch}$ and the time advancement rule $\texttt{advance}$. The dispatch timer $O_w.disp(C)$ must be $\texttt{t}(\texttt{0})$ when $\texttt{dispatch}$ is applied. Furthermore, when $O_w.disp(C)) = \texttt{t}(\texttt{0})$, the $\texttt{advance}$ rule cannot advance the system (except by idempotent 0 time ticks).

We first show that the starting point of the system reduces to considering a system with $O_w.disp(C) = \texttt{t}(0)$ by advancing time until the timer expires. Suppose that $O_w.disp(C) = \texttt{t}(T)$ with $T > 0$, then the *advance* rule must be applied before we can apply the *dispatch* rule. Of course, $\texttt{tick}(\texttt{nil}, T') = \texttt{nil}$, so $O_w.log$ will remain unchanged. Furthermore, $O_w.val$ does not change with time advancement. Finally, by Theorem 4.5.11, $O_w.next$ will remain valid. Thus, after many applications of rules other than *dispatch*, the configuration C will still satisfy the *init* proposition. Thus, we only have to consider the case of the system starting with $O_w.disp(C) = \texttt{t}(\texttt{0})$.

We have reduced the problem to where C starts with an expired timer. Let t_r denote the real time of the system C, and define the initial time (when $O_w.disp$ first expires) to be $t_r = 0$. Since $O_w.disp$ ticks

with real-time, and only the `dispatch` rule modifies $O_w.disp$ from `t(0)` to `t(period)`, it must be the case that `dispatch` is applied exactly once every `period`. This means that the `dispatch` rule is applied at times $t_r = n \times \texttt{period}$, where n is a natural number. Let $C(t_r)$ be the system configuration at time t_r. We now show that $\texttt{inEnv?}(O_w.val(C(t_r)), O_w.log(C(t_r))) = \texttt{true}$ for all $t_r \in \{n \times \texttt{period} | n \in \mathbb{N}\}$.

For $n = 0$, it is clear that $\texttt{inEnv?}(O_w.val(C(0)), O_w.log(C(0))) = \texttt{true}$ just by definition and the conditions of the *init* predicate. For the inductive step, using Lemma 4.5.13 we can check that the inductive hypothesis is indeed satisfied in each step, so each application of the `dispatch` rule will preserve the truth value of `inEnv?`.

Now, we have $\texttt{inEnv?}(O_w.val(C(t_r)), O_w.log(C(t_r))) = \texttt{true}$ from all times $t_r = n \times \texttt{period}$. It then follows that $\texttt{safe?}(O_w.log(C(t_r))) = \texttt{true}$. Since `safe?` is stably invalidated as shown in Lemma 4.5.1, for every time t_r, there is an n such that $t_r < n \times \texttt{period}$, we have that $\texttt{safe?}(\texttt{O}_\texttt{w}.\texttt{log}(\texttt{C}(\texttt{t}_\texttt{r})))$ holds for all times $t_r \geq 0$. Showing that $C(t_r) \models safe_{hist}$ for all reacheable times t_r. □

4.5.7 Model Checking Completeness of Nested Object Configurations

This section proves that, as already stated in Section 4.4, the model checking analysis performed in Real-Time Maude for the nested object configurations corresponding to instantiations of the Command-Shaper Pattern are *complete*, that is, if it does not find a counterexample within a given time bound, no such counterexample exists, for such a bound. The result we give here is *more general*, and therefore applies to the modeling checking of other systems involving nested object configurations, such as those associated to the patterns in Chapters 3 and 5.

Theorem 4.5.15. *Consider a rewrite theory $\mathcal{R}$ in Real-Time Maude.*

Let S_t be a set of timed sorts such that for all $s \in S_t$ the operators `tick` $: s$ `Time` $\rightarrow s$ *and* `mte` $: s \rightarrow$ `TimeInf` *exist. Let $S_{t0} \subseteq S_t$ be the set of timed sorts such that for all terms t of any sort $s \in S_{t0}$, t does not contain a proper subterm of any sort in S_t. Also, let all instantaneous rewrite rules have left hand sides with sort $s \in S_t$.*

Furthermore, assume that for any function symbol f that is not the system encapsulation function $\{\_\}$, if `f : s1 ... sn -> s` *has $s_i \in S_t$ for any i, then $s \in S_t - S_{t0}$. Assume that the following equations are satisfied by $\mathcal{R}$ for any f with a domain sort in S_t:*

$\texttt{tick}(f(t_1, \ldots, t_n), T) = f(t'_1, \ldots, t'_n)$, *where $t'_i = tick(t_i, T)$ if t_i is a term of sort $s \in S_t$ and $t'_i = t_i$ otherwise,*

and $\texttt{mte}(f(t_1, \ldots, t_n)) = min\{\texttt{mte}(t_i) | t_i \text{ is of sort } s \in S_t\}$.

Finally, let there be only one timed rewrite rule in $\mathcal{R}$:

```
var C : Configuration .  var T : Time .
crl [advance] :  {C} => {tick(C, T)} in time T if T le mte(C) [nonexec] .
```

Then, $\mathcal{R}$ is time-robust (see [44]) if the following conditions are satisfied for all appropriate ground terms t of a sort $s \in S_t$ and r , r' of sort `Time`*:*

1. $\texttt{mte}(\texttt{tick}(t, r)) = \texttt{mte}(t) \dot{-} r$ *for all* $r \leq \texttt{mte}(t)$
2. $\texttt{tick}(t, 0) = t$
3. $\texttt{tick}(\texttt{tick}(t, r), r') = \texttt{tick}(t, r + r')$, *for* $r + r' \leq \texttt{mte}(T)$
4. $\texttt{mte}(\sigma(l)) = 0$ *for each ground instance* $\sigma(l)$ *of a left-hand side of an instantaneous rewrite rule.*

Furthermore, it is sufficient to consider conditions 1-3 for t of sort $s \in S_{t0}$.

Proof Sketch. Most of the proof from Theorem 19 in [44] extends almost directly to the definitions here. The only subtle part is the proof of time robustness property *TR4*: if $t \xrightarrow{r} t'$ is a tick step with $r > 0$, and $t' \xrightarrow{inst} t''$ is an instantaneous one-step rewrite, then $t \xrightarrow{r} t'$ is a maximal tick step.

Let $\{t_{top}\}$ be the top-level encapsulated term of sort `System`. If an instantaneous rule applies to $\{t_{top}\}$, then by the assumptions on the format of instantaneous rewrite rules some subterm t' of sort $s \in S_t$ must match a substitution of the left hand side of a rewrite rule. By Condition 4, this means that $\texttt{mte}(t') = 0$. If $\texttt{mte}(t_{top}) > 0$, by an inductive argument, `mte` is propagated from the minimum among all timed subterms to the top-most term. Thus, it must be the case that $\texttt{mte}(t') \geq \texttt{mte}(t_{top}) > 0$, a contradiction. □

Chapter 5

A Pattern for Network Safety

Medicine is all about interactions. The patient's health is improved by interactions with treatment. Of course, treatments often interact with each other and present the possibility of adverse interactions. Drug contraindications and conflicts are commonplace in the area of medical drugs. A similar phenomenon happens in medical devices. In this chapter, we consider safety of device interactions, and in particular mutual exclusion constraints. We give a practical example involving laser airway surgery. We look at network faults that can make designing protocols difficult and can lead to unsafe situations. We discuss a formal pattern that will allow the system to operate safetly under message loss failures, and we describe how this pattern can be used to deal with mutual exclusion constraints.

5.1 Airway Laser Fires - A Case Study

Consider a typical airway laser surgery scenario. A surgical laser is used to perform surgery on a patient's airway. However, the flow of oxygen is not stopped, and the heat from the surgical point can start a fire. Many times these fires can be dealt with, without any permanent harm to the patient, but sometimes these airway fires can lead to unfortunate fatalities [50].

Airway laser fires occur frequent enough, that methods have been developed to deal with these problems with human protocols [46]. However, the safety of preventing fires can further be improved by using automated controls and interlocks between oxygen supply devices and active surgical equipment.

5.2 The Issue of Open Loop Safety

It is easy enough to design a protocol to prevent the airway laser surgery scenario. Always make sure that the laser is turned off before the oxygen is turned on. Figure 5.1 shows the simple first step implementation of a potential laser oxygen interlock protocol. The laser requests for the oxygen to turn off, and it will wait for the oxygen to return an acknowledgement. Once the oxygen acknowledges, then the laser will turn on safely after some delay. When the oxygen timer runs out, then it will request the laser to turn off again,

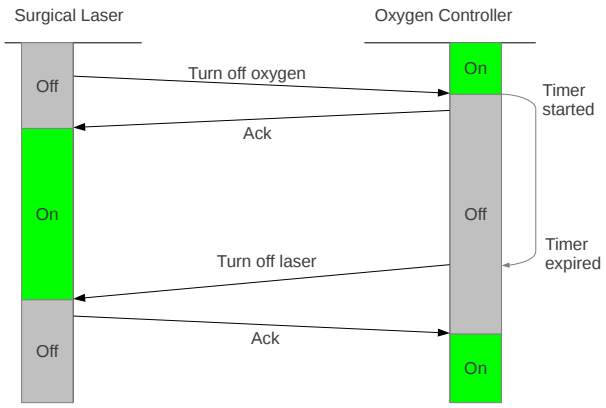

Figure 5.1: Simple Laser Protocol

and it will turn back on the oxygen before further harm comes to the patient.

On a purely logical viewpoint, there is nothing wrong with the laser oxygen interlock. However, from a fault tolerance standpoint, it can be a very dangerous protocol. Once we consider message loss in the system, then all bets are off. Once the oxygen turns off, the network can fail and prevent any further messages from being sent. The oxygen will continue to try to send messages requesting the laser to turn off, but no messages will delivered. After a certain time, we are left with a lose-lose situation. If the oxygen stays off, then permanent brain damage can result due to the lack of oxygen, but if the oxygen turns on, then a potential case of airway fire can occur. This dilemma is illustrated in Figure 5.2.

So how can a protocol be designed so that we never get into these unsafe situations? This is the idea of designing for open loop safety.

5.3 The Heartbeat Pattern

5.3.1 Modeling Network Faults

We move from discussing a single device interface and interface faults to networks of devices. In order to start thinking about the problems that occur in networked configurations, it suffices to start considering two communicating entities. Essentially, a network can be thought of as many pairs of communicating entities.

We first focus exclusively on the fault of message loss during communication. Although this seems like a very specific fault, many of the known existing network faults can be abstracted into this type. Of

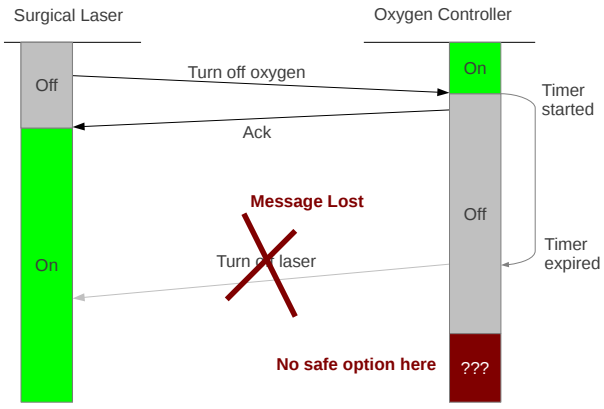

Figure 5.2: Network Loss and Failures

course, not receiving or dropping a message is clearly a message loss, but what about other faults such as message corruption and delayed messages? For message corruption, in order for the system to deal with it correctly, either error detection or error recovery must be implemented. If a message error is detected and not correctable, then we just have a useless message and this is equivalent to a message loss. If the message error is recoverable, then we have the original message as if no errors occurred. In terms of delayed messages, since we are dealing with real-time systems (medical devices that must operate in a timely manner) then not receiving the message in time can also be considered a message loss. For many other types of network faults the same reasoning can be used to reasonably abstract them to a message loss fault.

We can model a message loss fault in the Maude object-oriented actor model with a single rule:

```
crl M => none if lossy?(M) = true .
```

Here a message is just removed form the configuration if it satisfies the predicate `lossy?`.

5.3.2 Transient Commands and Fail-Safe Modes

Consider the case study of turning off a ventilator for a short period of time in order to perform some other medical procedure. This is a transient command as it should only be performed temporarily, and a turn on message must be sent after this short window of time to restart the ventilator. However, if the network drops this turn on message, then the ventilator may never turn on, and we enter an unsafe situation. In order to deal with this error, we cannot rely on the network for delivering critical messages on time, and there must be some intrinsic behavior in the ventilator for falling back to the fail-safe mode itself.

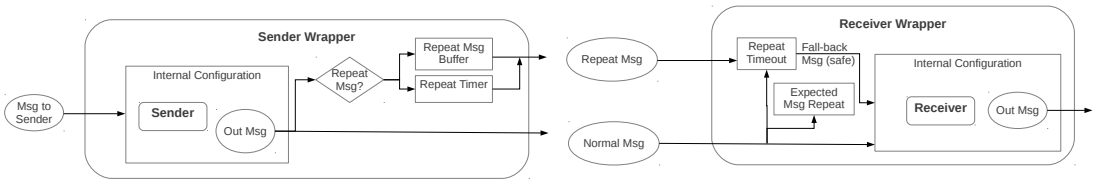

Figure 5.3: The Heartbeat Protocol

There are many ways to acheive this, and actually, we have already solved this exact problem as an instance of the Command-Shaper Pattern before. The Command-Shaper focused on the level of safety that can be guarateed from a single device receiving commands from unknown sources, and in a sense, the wrapped device acts somewhat autonomously. This makes analyzing the individual device easier since it no longer depends completely on the network, but it makes analyzing networked behaviors difficult since they may or may not behave according to the commands issued. However, if we also control the behavior of the sender of these commands, then it is possible to design something that is still safe, but much easier to analyze from a network perspective.

5.3.3 The Heartbeat Protocol

We present a well known pattern for handling unreliable network communication, while still providing safe fall-back modes. The overall structure of our heartbeat protocol is shown in Figure 5.3.

The heartbeat protocol essentially creates a pair of wrappers around a sender-receiver pair. It classifies the messages into repeated and non-repeated messages. The messages that are repeated are exactly the ones that are for keeping devices in transient states as discussed above. The receiver will only stay in a transient state if it continues to receive the repeated messages within a certain time. The sender will periodically send the repeated messages. The idea is that if too many messages get dropped, then the receiver wrapper will time out on receiving the repeated message and will automatically send a fall back message to the internal configuration.

To capture the required information for the heartbeat protocol in Maude, we specify the following theory:

```
(oth COMM-PAIR is
   inc LTIME-INF .
   class Sender .
```

```
   class Receiver .

   op dest : Msg -> Oid .
   op sid : -> Oid .
   op rid : -> Oid .
   op repeat-msg? : Msg -> Bool .
   msg safe-msg : Oid -> Msg .

   op repeat-time : -> Time .
   op timeout : -> Time .

   eq repeat-time lt timeout = true .

   op init-sender : -> Configuration .
   op init-receiver : -> Configuration .
endoth)
```

Of course, this theory assumes that the system is modeled based on an object-oriented actor model with linear time semantics. First, the `Sender` and `Receiver` classes are identified as the classes for the objects to be wrapped. The `dest` operator is used to extract the destination object id from a message (since message format is unknown). The `sid` and `rid` identify the specific object ids for the sender and receiver to be wrapped. The `repeat-msg?` predicate determines which messages should be repeated (i.e. which messages are only setting transient states). The `safe-msg` operator defines the message to the fall-back mode. The `repeat-time` sets the period in which the sender will resend repeated messages, and the `timeout` defines how long the receiver will receive a message before it performs the fail-safe actions. Of course, the assumption is that the sender repeat time is smaller than the receiver timeout otherwise, the receiver will just timeout before the sender even gets a chance to send. The remaining parameters are for setting the initial states of the system.

Using the above input theory `LTIME-INF`, we can define the wrapper objects of the Heartbeat protocol in a parameterized module:

```
(tomod REPEATER{X :: COMM-PAIR} is
   pr DELAY-MSG .
   class SenderWrap | internal : Configuration, time : Timer,
```

```
          repeat-msg : Configuration .ing
class ReceiverWrap | internal : Configuration, time : Timer,
          repeat-msg : Configuration .
```

The `internal` attribute stores the wrapped configuration. Timers are used to both know when to send a repeated message and when to time out for the receiver. The `repeat-msg` attribute is used to store what message is currently being repeated. But the crucial aspect of the protocol is its real-time behavior as fromally specified by rewrite rules. To illustrated the subtleties involved, we first present a flawed first attempt, and then give the correct rules.

5.3.4 An Intuitive Yet Flawed First Attempt

The heartbeat protocol is intuitive enough. If we assume that messages received are idempotent (as is the case for our case study: two turn off messages is the same as one), we just need to repeat messages until the next one is sent. The important heartbeat rules specified in our first attempt to specify the protocol as a parameterized module in Maude are as follows.

If the sender needs to send a repeated message, then the message is forwarded, and the repeat timer is set, and the repeated message is buffered to be retransmitted.

```
crl < sid : SenderWrap | internal : M C, time : TM >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
repeat-msg : M >  M
if dest(M) =/= sid /\ repeat-msg?(M) = true .
```

If the repeat timer times out, then retransmit the buffered message.

```
crl < sid : SenderWrap | internal : C, time : TM, repeat-msg : M >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
repeat-msg : M >  M
if TM = timer0 .
```

If the receiver receives a repeated message then reset the timeout timer and deliver the message.

```
crl M < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : M C, time : t(timeout) >
if dest(M) = rid /\ repeat-msg?(M) = true .
```

If the receiver timer timesout, then go to a fall-back mode.

```
crl < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : safe-msg(rid) C, time : no-timer >
if TM = timer0 .
```

This description follows immediately form the intuitive description of the pattern. For model checking, we let the sender turn the receiver off from time 0 to 50, then turn the receiver back on after time 50. We are not even considering message loss yet, this is merely a check of the pattern's correctness in behavior under normal conditions. In the model checking, we look for points where the receiver is still turned off after time 50. We checked up to time 100, and we get many counter examples, the last one being

```
Solution 61
C':Configuration --> < receiver : Receiver | state : on >
   < sender : Sender | receiver : receiver,state : on >; C:Configuration --> <
    receiver : ReceiverWrap | internal : < receiver : Receiver | state : off >
    ,time : no-timer >
    < sender : SenderWrap | internal : < sender :
    Sender | receiver : receiver,state : on >,repeat-msg : none,time : no-timer >
    ; TIME_ELAPSED:Time --> 100

No more solutions
```

This means that it is possible that at time 100 the receiver was still off, and indeed looking into the details of the counter example, the receiver will be off indefinitely. The reason for this is that messages can be reordered, and the repeated message can become out of order with a normal command message overriding that command. This shows the importance of encoding patterns formally and not just relying on intuitive descriptions for safety critical designs. Designs should be provably safe.

5.3.5 Ironing Out The Kinks

The disastrous counterexample that we have just seen has two fundamental causes. One is that the messages can be reordered; the other is that repeated messages are indistinguishable from original commands. Addressing each of these points can solve the problem.

To get rid of the problem of message reordering, we can add time stamps, for example the first sender rule now becomes

```
crl < sid : SenderWrap | internal : M C, time : TM,
        time-stamp : N >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
   repeat-msg : M, time-stamp : up(N) >
   (if dest(M) == rid then stamp(rid, M , N) else M fi)
if dest(M) =/= sid /\ repeat-msg?(M) = true .
```

We of course needed to change the sender and receiver attributes to add in the time stamp. We also need to introduce a new message format with time stamps. And we must also ensure that forwarding to internal wrapped objects also preserves the desired order. After all these changes to the protocol, model checking this system does remove all the counterexamples. However, the time stamp based approach does have it's drawbacks. Of course the system becomes more complicated and, furthermore, the timestamps cause the untimed state space to become infinite, which makes complete model checking more difficult without good abstraction techniques. Furthermore, this increase in state space and context complicates the proof of correctness, and it makes many more assumptions and constaints on the design.

Thus, it is better to proceed based on the second approach of distinguishing repeated messages from original commands. Using the repeated messages only to reset timeouts but not to be delivered to the internal configuration. The complete specification of this design is as follows:

```
op repeated : Oid Msg -> Msg .
op repeated? : Msg -> Bool .

vars M M' : Msg .
var C : Configuration .
var TM : Timer .
var T : Time .

var O : Oid .

eq dest(repeated(O,M)) = O .
eq repeat-msg?(repeated(O,M)) = false .
eq repeated?(repeated(O,M)) = true .
eq repeated?(M) = false [owise] .
```

```
eq mte(< sid : SenderWrap | internal : C, time : TM >)
   = minimum(mte(C), mte(TM)) .
eq mte(< rid : ReceiverWrap | internal : C, time : TM >)
   = minimum(mte(C), mte(TM)) .

eq tick(< sid : SenderWrap | internal : C, time : TM >, T)
= < sid : SenderWrap | internal : tick(C,T), time : tick(TM, T) > .
eq tick(< rid : ReceiverWrap | internal : C, time : TM >, T)
= < rid : ReceiverWrap | internal : tick(C, T), time : tick(TM, T) > .

op init-sender-wrap : -> Configuration .
eq init-sender-wrap =
< sid : SenderWrap | internal : init-sender, time : no-timer,
   repeat-msg : none > .

op init-receiver-wrap : -> Configuration .
eq init-receiver-wrap =
< rid : ReceiverWrap | internal : init-receiver, time : no-timer,
   repeat-msg : none > .

--- send a nonrepeated message
crl [send-nonrepeat] : < sid : SenderWrap | internal : M C, time : TM >
=> < sid : SenderWrap | internal : C , time : no-timer,
repeat-msg : none > M
if dest(M) =/= sid /\ repeat-msg?(M) = false .

--- send a repeated message
crl [send-repeat] : < sid : SenderWrap | internal : M C, time : TM >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
repeat-msg : M >  M
if dest(M) =/= sid /\ repeat-msg?(M) = true .
--- need dest(M) = rid ???
```

```
--- repeat a message
crl [repeat] : < sid : SenderWrap | internal : C, time : TM, repeat-msg : M >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
repeat-msg : M >  repeated(dest(M), M)
if TM = timer0 .

--- forward msg to sender
crl [fwd-to-sender] : M < sid : SenderWrap | internal : C >
=> < sid : SenderWrap | internal : C M >
if dest(M) = sid .

--- forward a message that will be repeated
crl [fwd-with-repeat] : M < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : M C, time : t(timeout),
repeat-msg : M >
if dest(M) = rid /\ repeat-msg?(M) = true .

--- forward a message that is not repeated
crl [fwd-no-repeat] : M < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : M C, time : no-timer >
if dest(M) = rid /\ repeat-msg?(M) = false /\ repeated?(M) = false .

--- receiving a message repeat that was seen
rl [repeated-seen] : repeated(rid,M) < rid : ReceiverWrap | internal : C, time : TM,
repeat-msg : M >
=> < rid : ReceiverWrap | time : t(timeout) > .

--- receiving a message repeat that was not seen
crl [repeat-not-seen] : repeated(rid,M) < rid : ReceiverWrap | repeat-msg : M' >
=> < rid : ReceiverWrap | >
if M =/= M' .
```

```
--- repeat message timeout
crl [repeat-timeout] : < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : safe-msg(rid) C, time : no-timer,
repeat-msg : none >
if TM = timer0 .

--- forward out from receiver
crl [fwd-from-recv] : < rid : ReceiverWrap | internal : M C >
=> < rid : ReceiverWrap | internal : C > M
if dest(M) =/= rid .

endtom)
```

The modelchecking for this new protocol also generates no counterexamples, and has the advantage of being much simpler than the design using time stamps.

5.3.6 Proof of Correctness

We show that under the condition of no message loss our pattern gives the exact behavior as an ideal communicating pair. Again, this requires us to prove a stuttering bisimulation, and the first step in to establish the bisimulation relation H.

The only difference between our system and the ideal system is that the sender and receiver are wrapped and there are more types of messages. Thus we define the projection relation as follows:

```
vars C C' : Configuration .
vars NC NC' : NEConfiguration .
op proj-wrapped : Configuration -> Configuration .
eq proj-wrapped(NC NC') = proj-wrapped(NC) proj-wrapped(NC') .
eq proj-wrapped(< receiver : ReceiverWrap | internal : C >) = C .
eq proj-wrapped(< sender : SenderWrap | internal : C >) = C .
eq proj-wrapped(repeated(O:Oid, M:Msg)) = none .
eq proj-wrapped(NC) = NC [owise] .
```

Thus giving two systems, an ideal one $s = \{C\}$ and a wrapped one $s' = \{C'\}$. We have that sHs' iff

$C =$`proj-wrapped`(C').

In order to show this, we must first show that under normal operation (no dropped messages and no out of order delivery), the wrapped receiver will never take the fall back mode. That is the rule `repeat-timeout` should never be applied starting in a proper initial state and when no messages are lost.

Definition. Let s be a wrapped system of the form

C `<` O `:` `SenderWrapper | internal :` C'`, time :` TM`, repeat-msg :` M `>`
`<` O' `:` `ReceiverWrapper | internal :` C''`, time :` TM'`, repeat-msg :` M' `>`

s is a *proper initial state* iff

1. C contains no messages of the form `repeated`(Oid, Msg) and no other objects of type `SenderWrapper` and `ReceiverWrapper`
2. $TM = TM' =$ `no-timer`
3. $M = M' =$ `none`

Also, define a relation $<$ for timers such that $t(T) < t(T')$ when $T < T'$ and also $t(T) <$ `no-timer`.

With the proper definition of an initial state, we have the following Lemma.

Lemma 5.3.1. *Let s_0 be a proper initial state of a wrapped sender-receiver pair. Let s be a any state reacheable from s_0 by the rewrite rules in the module* `REPEATER`*. s will have the form*

C `<` O' `:` `ReceiverWrapper | time :` TM'`, ...>`*, and* $TM' > t(0)$

Proof. We proceed by induction on the number of rewrite rules applied from the module `REPEATER`. We use the stronger induction hypothesis that any reacheable state has the form

$s = C$ `<` O `:` `SenderWrapper | time :` TM`, repeat-msg :` M`, ...>`
`<` O' `:` `ReceiverWrapper | time :` TM'`, repeat-msg :` M'`, ...>`

and one of the following must hold

1. $TM' =$ `no-timer`
2. $C = C'$ `repeated(`O'`,` M'`)` and $M = M'$ and $TM' > t(0)$
3. $C = C'M''$ where `repeat-msg?(`M''`) = false` and M `= none`
4. $TM' > TM$.

The initial state clearly satisfies the induction hypothesis.

Now, we reason on each of the rules. Notice that since we assume in order delivery of messages the send and receive rules are usually taken in pairs. Consider the step $s' \rightarrow s$.

Notice that the timer for the sender, when set, is always set to $t(\texttt{repeat-time})$, and the timer for the receiver, when set, is always set to $t(\texttt{timeout})$. Also, since values of timers can never increase, we always have $TM' \leq t(\texttt{timeout})$, and $TM \leq t(\texttt{repeat-time})$. We also have $t(\texttt{repeat-time}) < t(\texttt{timeout})$ by the requirements of the input theory `COMM-PAIR`.

If the last rewrite rule taken was `send-nonrepeat`, then by the form of the right hand side of the rule, s satisfies case 3 of the induction invariant.

If the last rewrite rule taken was `send-repeat` or `repeat`, then by the form of rewrite rule, s should satisy case 2 of the induction invariant.

If the last rewrite rule taken was `fwd-with-repeat`, and assuming s' only had one instance of a repeated message, then the sender must have sent a repeated message by the rules `send-repeat` or `repeat`. Since time cannot elapse while a message is in the configuration, the sender timer must be $TM = t(\texttt{repeat-time})$, and the receiver timer is set to $TM' = t(\texttt{timeout})$, and we have $TM' > TM$.

If the last rewrite rule taken was `fwd-no-repeat`, then the receiver timer is set to TM' is `no-timer`, and s satisfies case 1.

If the last rewrite rule was `repeated-seen`, then the sender has just sent a repeated message, and the sender timer must be set. The receiver timer is set to $t(\texttt{timeout})$ (largest possible time to set for any sender or receiver timer), and we have $TM' > TM$.

Any other rewrite rule, including `fwd-to-sender` or `fwd-from-recv`, should not modify the attributes of the `SenderWrapper` and `ReceiverWrapper` objects, and also should not add any repeated messages into the external configuration. The rule `repeat-not-seen` should never be taken if the messages are delivered in order, and the rule `repeat-timeout` will of course never be taken as the inductive hypothesis always has $TM' \neq t(0)$.

Thus considering all possible rewrite rules, we see that the `time` attribute for the `ReceiverWrapper` object never reaches $t(0)$. □

The important implication of this Lemma is that we no longer have to consider the case where the `ReceiverWrapper` timesout by not receiving a repeated message. Thus, in reasoning about the case where no messages are lost in communication, it means that we do not have to consider transitions taken by the rule `repeat-timeout`.

With this knowledge, we are ready to prove the bisimulation result between an ideal communicating pair,

and a wrapped communicating pair.

We again use the notion of a well-founded simulation to show a bisimulation. First, we define some important values associated with the states.

Definition. Let a wrapped system s have the form:

{ C < O : SenderWrapper | internal : C', time : TM, repeat-msg : M >

< O' : ReceiverWrapper | internal : C'', time : TM', repeat-msg : M' > }

Let $nmsg_{ts}(s)$ be the number of messages in C to the sender object O.

Let $nmsg_{tr}(s)$ be the number of messages in C to the receiver object O'.

Let $nmsg_{fs}(s)$ be the number of messages in C' where the destination is not the sender object O.

Let $nmsg_{fr}(s)$ be the number of messages in C'' where the destination is not the receiver object O'.

Let χ_{repeat} be an indicator value that is set to 1 when $TM = t(0)$ and 0 otherwise.

Also, in order for the wrapper to function correctly, we must satisfy the condition of no messages bypassing the sender wrapper. The state of s actually has enough information to define the condition of no bypass without considering time. Let $nobypass(s)$ be true iff

1. C contains no messages to the receiver O'
2. C contains a message MR to the receiver, and $M = MR$

Define the relation $H \subseteq S_{ideal} \times S_{wrapped}$ such that $\{C\}H\{C'\}$ iff $C =$proj-wrapped$(C') \wedge nobypass(C)$.

We have already defined a proper inital state for wrapped systems earlier. For an ideal system, define a proper inital state to be a configuration with a sender and reciever (to be wrapped) and no messages in transit.

Theorem 5.3.2. *The relation H is a well-founded bisimulation (when restricted to all the reachable states from proper inital states).*

Proof. We first show the well-founded simulation form S_{ideal} to $S_{wrapped}$.

Define $\mu(s_i, s_f) = 0$ and $\mu(s_i, s_i', s_f) = 3nmsg_{fr}(s_f) + 2\chi_{repeat} + 2nmsg_{fs}(s_f) + nmsg_{ts}(s_f) + nmsg_{tr}(s_f)$.

Suppose that $s_i H s_f$, we consider possible transitions $s_i \rightarrow s_i'$.

Let s_f have the following form

{ C < O_s : SenderWrapper | internal : C_s, time : TM, repeat-msg : M_r >

< O_r : ReceiverWrapper | internal : C_r, time : TM', repeat-msg : M_r' > }

Let $nmsg_{ts}(s)$ be the number of messages in C to the sender

(1) $mte(s_i) = 0$ and a message M is delivered in s_i', we consider the following cases:

(1.a) M is not a message to or from O_s or O_r, and thus M is also in the external configuration of s_f, then $s_f \to s'_f$ takes a corresponding transition by using the same rule, and we have $s'_i H s'_f$.

(1.b) M is a message to O_s, and M is in C_s, then, again, $s_f \to s'_f$ takes a corresponding transition using the same rule, and we have $s'_i H s'_f$.

(1.c) M is a message to O_s, and M is in C, then we can apply the `fwd-to-sender` rule, and $nmsg_{ts}$ decreases, and thus μ' decreases as required.

(1.d) M is a message to O_r, and M is in C_r, this is similar to case (1.b)

(1.e) M is a message to O_r, and M is in C, and `repeat-msg?`(M) =`true`, the `fwd-with-repeat` rule applies, and $s_i H s'_f$ and $nmsg_{tr}$ decreases.

(1.d) M is a message to O_r, and M is in C, and `repeat-msg?`(M) =`false`, the `fwd-no-repeat` rule applies, and $s_i H s'_f$ and $nmsg_{tr}$ decreases.

(1.f) M is a message from O_s, and M is in C_s, M must be forwarded by either rules `send-nonrepeat` or `send-repeat`. In any case, $s_i H s'_f$ and $nmsg_{fs}$ decreases by 1, and $nmsg_{tr}$ or $nmsg_{ts}$ (not both) may increase by 1, but this will still decrease μ'.

(1.g) M is a message from O_r, and M is in C_r, M must be forwarded by the rule `fwd-from-recv`. $s_i H s'_f$ and $nmsg_{fs}$ decreases by 1, and $nmsg_{ts}$ may increase by 1, but we have a net decrease in μ'.

(1.h) M is a message from O_s or O_r, but M is in C. This will be subsumed under the previous cases unless the destination of M is not O_s or O_r, in which case s_f will take a corresponding rewrite rule $s_i H s'_f$.

(2) $mte(s_i) > 0$, and $s_i \to s_f$ is a tick rule advancing time by one time unit. We consider a few subcases for the transition of s_f.

(2.a) $mte(s_f) > 0$, then we also apply a one time unit tick rule to have $s_i H s_f$.

(2.b) $mte(s_f) = 0$ because the sender repeat timer TM has expired. In this case, the rule `repeat` is used, and we have $s_i H s'_f$ and χ_{repeat} decreases, and $nmsg_{tr}$ increases; a net decrease in μ'.

(2.e) $mte(s_f) = 0$ because a message M needs to be delivered. The only new messages that can be in s_f and not s_i are repeated messages, and so the rule `repeated-seen` can be applied, and we have a decrease in $nmsg_{tr}$.

This covers all the cases for showing a well-founded simulation from S_{ideal} to $S_{wrapped}$. Now for the other direction.

We define $\nu(s_f, s_i) = 3nmsg_{fr}(s_f) + 2\xi_{repeat} + 2nmsg_{fs}(s_f) + nmsg_{ts}(s_f) + nmsg_{tr}(s_f)$

and $\nu'(s_f, s'_f, s_i) = 0$.

Let $H' = H^{-1}$, and suppose $s_f H' s_i$.

We consider the following cases:

(1') $mte(s_f) > 0$, and $s_f \rightarrow s'_f$ is a tick rule advancing time by one time unit. By the homomorphic definition of mte, if $s_f H' s_i$, then $mte(s_i) > 0$, and a corresponding one time unit tick step can be taken $s_i \rightarrow s'_i$ with $s'_f H' s'_i$.

In the next set of cases, we consider transitions that occur by the rules in module `REPEATER`. Note that none of the rules modify the states of internal objects, and they only generate new message of the form `repeated`. Thus, $s'_f H' s_i$ holds for all these cases by the nature of the projection relation. Thus, we only have to show that ν decreases.

(2') s'_f is reached by rule `send-nonrepeat`. $nmsg_{fs}$ decreases by 1 while $nmsg_{tr}$ or $nmsg_{ts}$ (but not both) may increase by 1 resulting in a net decrease of ν.

(3') s'_f is reached by rule `send-repeat`. Even though internal object parameters are set differently, externally, this looks the same as case (2').

(4') s'_f is reached by rule `repeat`. χ_{repeat} decreases by 1 and $nmsg_{tr}$ increases by 1 resulting in a net decrease of ν.

(5') s'_f is reached by rule `fwd-to-sender`. $nmsg_{ts}$ decreases.

(6') s'_f is reached by rule `fwd-with-repeat` or `fwd-no-repeat` or `repeated-seen`. $nmsg_{tr}$ decreases.

(7') s'_f is reached by rule `fwd-from-recv`. $nmsg_{fr}$ decreases by 1 while $nmsg_{tr}$ or $nmsg_{ts}$ (but not both) may increase by 1 resulting in a net decreases of ν.

The last case to consider is when $mte(s_f) = 0$ and a rule outside the `REPEATER` module is applied. We assume that none of these rules have terms containing `SenderWrapper`, `ReceiverWrapper`, or `repeated`. In this case, s_i should also be able to match the left hand side of the same rule, and we have $s'_f H' S'_i$.

Note that for all these cases, we were able to ignore the rules `repeat-timeout` and `repeat-not-seen` in the case by case analysis by the results of Lemma 5.3.1, and since we only consider proper initial states.

We have shown that H is a well-founded bisimulation, and thus, we have a bisimulation between ideal communicating pairs and wrapped sender and receiver configurations.

□

5.3.7 Proof of Robustness

What we have just proved is the correctness of the Heartbeat pattern when the network does not fail. This is good in showing that under normal operating conditions our pattern does not deviate from the ideal behavior. However, our pattern should do much more than that under failure situations. Note that in cases where faults occur, there is really no ideal behavior to compare to since the fault has already deviated from ideal conditions. Furthermore, message loss over the network, and real-time constraints means that the

system must make automated safe transitions even when they are not required.

Intuitively, a receiver is safe if it reacts to all the critical commands. In our pattern, the critical command is captured by `safe-msg`. A receiver reacting to a critical message would then mean that when a sender sends a `safe-msg`, then the receiver will receive the `safe-msg` at most some T_{max} time units later. This brings us to the following theorem.

Theorem 5.3.3. *Consider a sender and receiver configuration with view form* `COMM-PAIR`*. Now, consider a wrapped configuration*

{ C < O_s : `SenderWrapper | internal :` C_s, `time :` TM, `repeat-msg :` M_r >

< O_r : `ReceiverWrapper | internal :` C_r, `time :` TM', `repeat-msg :` M_r' > }

If the receiver object last received a repeated message M with `repeat-msg?(M) = true`*, and if the sender object with oid* `sid` *in C_s sends a message of type* `safe-msg` *at time t_1, and the sender object sends no other messages in time betwenn t_1 and t_1+*`timeout`*, then the receiver object with oid* `rid` *in C_r will receive a message of type* `safe-msg` *no later than t_1+*`timeout`*.*

This occurs even if we have the message loss rule at the top most level of the configuration:

`rl [drop-msg] :` { `M:Msg C:Configuration` } `=>` { `C:Configuration` } .

Proof. Suppose that the sender sends a `safe-msg` at time t_1, and since there are no other messages being sent by the sender at time t_1, then the configuration must be of the form:

{ C

< O_s : `SenderWrapper | internal :` C_s `safe-msg(`O_r`), time :` TM, `repeat-msg :` M_r >

< O_r : `ReceiverWrapper | internal :` C_r, `time :` TM', `repeat-msg :` M_r' > }

where C_s, C, and C_r contains no messages from O_s to O_r.

For simplicity we assume that the `safe-msg` is not repeated, that is, `repeated-msg?(safe-msg) = false`. This means that the rule `send-nonrepeat` will eventually be applied (assume the configuration we described is the last state before the rule was applied, and we have a configuration of the form:

{ C < O_s : `SenderWrapper | internal :` C_s, `time : no-timer, repeat-msg : none >`

`safe-msg`

< O_r : `ReceiverWrapper | internal :` C_r, `time :` TM', `repeat-msg :` M_r' > }

Now, there are two cases, one is to apply the rule `fwd-no-repeat`, in which case `safe-msg` ends up in the internal configuration of the receiver, eventually getting consumed by the receiver object. In this case, the statement of the theorem is trivially satisfied. The second case is that the rule `drop-msg` is applied, and we are left with the configuration:

{ C < O_s : `SenderWrapper | internal :` C_s, `time : no-timer, repeat-msg : none >`

safe-msg

< O_r : ReceiverWrapper | internal : C_r, time : TM', repeat-msg : M'_r > }

Since C contains no messages, and since the last message received by the receiver was repeated, it must be that timer TM' is set. Furthermore, the maximum value of TM' is `t(timeout)`, and no rules can be applied to reset TM' as all rules to reset the value requires a message from O_s to O_r in configuration C. This means that eventually TM' will expire in at most `timeout` time units, and the rule `repeat-timeout` will be used. This rule generates a `safe-msg` in the receiver configuration, and again the statement of the theorem is satisfied in this case.

□

5.4 A Safe Laser Surgery Protocol - An Application of the Heartbeat Pattern

We have described in detail a heartbeat pattern that can provide timely message delivery guarantees for critical messages even when the network may be faulty and drop messages. We have described in the beginning a laser surgery example that needs specific safety considerations when messages can be dropped. Clearly, creating a safe design for such a system is nontrivial and, furthermore, even if such a safe design was acheived, it is unclear how future changes in the code and protocols can affect the safety of the system. This is a perfect opportunity to apply the Heartbeat Pattern that we just described. It will provide safety guarantees while still maintaining flexibility of the internal logic.

The general idea of using the heartbeat pattern as a solution for faulty networks is to use the property guaranteed by Theorem 5.3.3. Clearly, the critical message that can be missed by the laser is the message to turn off. Thus, if we assign as the `safe-cmd` the turn off command for the laser, then we will be able to design a safe system taking into account timing delays. Recall that from Theorem 5.3.3, the maximum time from sending a `safe-cmd` to receiving it is at most `timeout` time units. Thus, we know that the laser must turn off within `timeout` time units if we use the pattern, and we are able to use this to guarantee the safety property that the oxygen and the laser will never be on at the same time.

A simple description of the laser oxygen system is shown below:

```
(tomod LASER is
   pr DELAY-MSG .
   pr POSRAT-TIME-DOMAIN-WITH-INF .
```

```
sort State .
ops on off : -> State .

class Oxygen | state : State, receiver : Oid, turn-on-timer : Timer,
   buffer-timer : Timer .
class Laser | state : State .

ops oxygen laser : -> Oid .

op off-time : -> Time .
eq off-time = 100 .
op buffer-time : -> Time .
eq buffer-time = 10 .

op set : Oid State -> Msg .

vars S S' : State .
var O O' : Oid .

rl set(O, off)
   < O : Oxygen | state : S', receiver : O' >
   => < O : Oxygen | state : off, turn-on-timer : t(off-time) >
   set(O', on) .

rl set(O, on)
   < O : Oxygen | state : S', receiver : O' >
   => < O : Oxygen | state : S', turn-on-timer : no-timer,
   buffer-timer : t(buffer-time) >
   set(O', off) .

rl < O : Oxygen | state : off, turn-on-timer : t(0), receiver : O' >
   => < O : Oxygen | state : off, turn-on-timer : no-timer,
```

```
      buffer-timer : t(buffer-time) >
      set(O', off) .

   rl < O : Oxygen | state : off, buffer-timer : t(0) >
      => < O : Oxygen | state : on, buffer-timer : no-timer > .

   rl set(O, S)
      < O : Laser | state : S' >
      => < O : Laser | state : S > .

   var T : Time .
   vars TM TM' : Timer .
   eq mte(< oxygen : Oxygen | buffer-timer : TM, turn-on-timer : TM' >)
      = minimum(mte(TM), mte(TM')) .
   eq tick(< oxygen : Oxygen | buffer-timer : TM, turn-on-timer : TM' >, T)
      = < oxygen : Oxygen | buffer-timer : tick(TM, T), turn-on-timer : tick(TM', T) > .
   eq mte(< laser : Laser | >) = INF .
   eq tick(< laser : Laser | >, T) = < laser : Laser | > .
endtom)
```

The laser and oxygen systems are both modelled as having two states on and off. The oxygen system has safety requirements to not turn off for too long. This is captured in the `turn-on-time` constant. A timer in the oxygen system will ensure that the oxygen turns back on in a desired time window. Furthermore, in order to compensate for command delays to the laser, the oxygen sysetm also has a `buffer-time` that it must also time before making state transitions. The rules are effectively describing changing states based on messages received as well as setting these timers appropriately. When the oxygen system wants to turn back on, it will send a message to the laser, as well as wait for the required `buffer-time`. Of course, this system by itself is clearly not safe, since the laser has no automated means to turn off once the network fails. Indeed, model checking a simple configuration gives many counter examples:

```
(tsearch { init-sender init-receiver init-msg }
   =>* { C:Configuration } s.t.
   get-state(oxygen, C) == on /\ get-state(laser, C) == on
in time <= 100 .)
```

```
Timed search in TEST-TEST
   {init-sender init-receiver init-msg} =>* {C:Configuration}
in time <= 100 and with mode default time increase 1 :

Solution 1
C:Configuration --> < laser : Laser | state : on > < oxygen : Oxygen |
    buffer-timer : no-timer,receiver : laser,state : on,turn-on-timer :
    no-timer > ; TIME_ELAPSED:Time --> 60

...
```

Now, we apply the heartbeat pattern, and automatically transition the receiver (the laser) to safe states when needed. First, we need to define all the parameters for the view.

```
(tomod LASER-AUX is
   inc LASER .
   op repeat-msg? : Msg -> Bool .

   var O : Oid .
   eq repeat-msg?(set(O, on)) = true .
   eq repeat-msg?(set(O, off)) = false .

   op dest : Msg -> Oid .
   eq dest(set(O, S:State)) = O .

   op safe-msg : Oid -> Msg .
   eq safe-msg(O) = set(O, off) .

   op timeout : -> Time .
   eq timeout = 8 .
   op repeat-time : -> Time .
   eq repeat-time = 5 .
```

```
   op init-sender : -> Configuration .
   op init-receiver : -> Configuration .

   eq init-sender =
      < oxygen : Oxygen |
         state : on, receiver : laser, turn-on-timer : no-timer,
         buffer-timer : no-timer > .
   eq init-receiver = < laser : Laser | state : off > .
endtom)

(view SimpleCommPair
   from COMM-PAIR
   to LASER-AUX is

   class Sender to Oxygen .
   class Receiver to Laser .

   op sid to oxygen .
   op rid to laser .
   op repeat-msg? to repeat-msg? .
   msg safe-msg to safe-msg .

   op repeat-time to repeat-time .
   op timeout to timeout .

   op init-sender to init-sender .
   op init-receiver to init-receiver .
endv)
```

Essentially, we are just providing the key information required before using the pattern correctly. The sending and receiving object ids and the appropriate timeouts to use. We also identify the `set(O, off)` message to be the `safe-msg`, and, naturally, since we only have two types of messages, the `set(O, on)` message is the repeated message. Note that the `timeout` is set to 8 time units, while the buffered state

transition delay for the oxygen is 10 time units, and by Theorem 5.3.3 this is sufficient waiting time.

Instantiating the pattern is pretty much completely done by the parameterized `REPEATER` specification. We just need to define the initial states for execution.

```
(tomod TEST-TEST is
   inc REPEATER{SimpleCommPair} .

   op init : -> Configuration .
   eq init = init-sender-wrap init-receiver-wrap .

   op init-msg : -> Configuration .
   eq init-msg = delay(set(oxygen, off), t(0)) delay(set(oxygen, on), t(50)) .
```

Furthermore, aside from instantiating the model, we model the message loss fault by a rule that will remove messages in the outermost configuration. Note that we only remove messages in the outer-most configuration, the inner configurations must still reliably deliver messages in order for the behavior to make sense. It is assumed that the inner configurations receive messages locally and not across the network.

```
   rl { M:Msg C:Configuration } => { C:Configuration } .
```

For model checking purposes, we also define an operator `get-state` to probe the internal states of the objects.

```
   op get-state : Oid Configuration -> State .
   op err : -> State .

   var O : Oid .
   vars C C' : Configuration .
   var S : State .
   eq get-state(O, < O : ReceiverWrap | internal : C > C') = get-state(O, C) .
   eq get-state(O, < O : Laser | state : S > C) = S .
   eq get-state(O, < O : SenderWrap | internal : C > C') = get-state(O, C) .
   eq get-state(O, < O : Oxygen | state : S > C) = S .
```

```
   eq get-state(O, C) = err [owise] .
endtom)
```

Thus, we have finally instantiated our laser-oxygen example, added the rule for the faults, given the information for an initial state, and provided the necessary operators to define the safety predicate. We can use a timed search to perform the model checking of the safety invariant that the laser and oxygen are never on at the same time.

```
(tsearch { init init-msg }
   =>* { C:Configuration } s.t.
   get-state(oxygen, C) == on /\ get-state(laser, C) == on
in time <= 100 .)
```

We find that, indeed, our system with the initial conditions defined has no counter examples:

```
Timed search in TEST-TEST
   {init init-msg} =>* {C:Configuration}
in time <= 100 and with mode default time increase 1 :


No solution
```

Chapter 6

Formal Model Based Device Emulation

Currently, all of our models execute in simulation time. In general for modeling and verification, this is a desirable feature, as simulation time can advance much faster than real time, and we may be able to simulate 1 hour of behavior in a few seconds and analyze it. However, Maude inherently provides us with the ability to execute our specifications, and the sockets API also allow the model to communicate with the outside world. This brings up the question of whether we can use the model not only as a simulation but an emulation of the software itself. That is, if we model a control algorithm for a medical device, what is to stop us from using this model as a real-time software controller? It turns out that the Maude framework has already laid out all the foundations for this to happen, and we just need to put them all together.

6.1 Distributed Emulation of Safe Medical Devices

We now discuss the transformation of the model to execute in *real world* time with physical devices in a medical device emulation environment. For this purpose we take the model of the wrapped pacemaker and wrap it again in an external execution wrapper (Figure 6.1). The execution wrapper is responsible for conveying to the model the notion of real world time as well as providing a communication interface to the external world. A dedicated timer thread is responsible for "ticking" the model by sending a minimal number of messages to advance the model's logical time. The timer thread also intercepts all asynchronous (interrupt) messages and relays them to the model. Another aspect of the execution wrapper is the ability to map external I/O messages to communicate with the external devices. For example, in the pacemaker specification an internal message called *paceVentricle* may be mapped into an entire client configuration to send a message for setting the final voltage on a pacing lead.

The external execution wrapper is an object that encapsulates the original model. It is primarily responsible for interfacing constructs between the physical world (the real interfaces to devices) and the logical world (the world as seen by the model). In particular, the execution wrapper is responsible for conveying the measurement of real time elapsed to the model and also mapping logical communication messages to

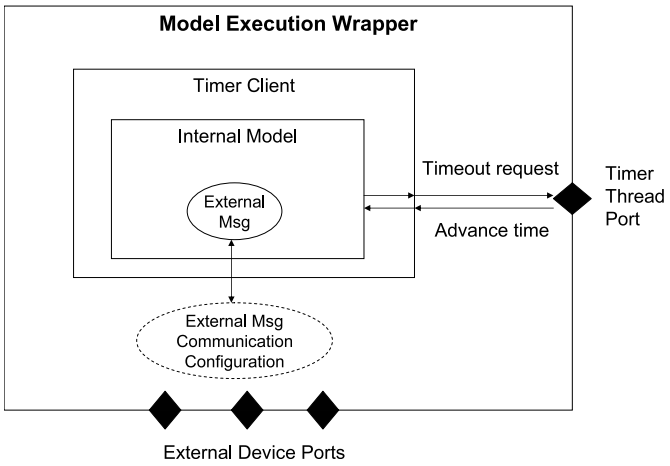

Figure 6.1: Real-Time Model Execution Wrapper

communication configurations that can deliver the message. The most important feature of the external execution wrapper is its modularity. Aside from adding the minimal information about how to map external I/O to messages in the model, no further specifications are required to execute the logical model with an external environment.

6.2 Mapping Internal Messages to External I/O

Validating the design of a device in an execution environment requires handling its outputs. After all, the end validation of a system's behavior is based on its outputs. Thus, it seems reasonable to talk about how internal messages in the model can be converted into messages for communicating with the external world. This section also serves as an explanation for unfamiliar readers of how Maude sockets are used.

In order to talk about external communication, we must first define in the model what is external. The model will have an internal distributed actor configuration with internal messages as well as messages to be output to the external world. Thus, the first definition is `EXTERNAL-CONFIGURATION`, which defines a sort of external messages `ExtMsg` as a subsort of `Msg`. Furthermore, external messages are classified in terms of incoming external messages `InExtMsg` and outgoing external messages `OutExtMsg`. A configuration is called *open* if there are external messages present in the configuration: either an incoming external message has not been delivered, or an outgoing external message has not been sent. The predicate `open?` is defined accordingly.

```
subsorts InExtMsg OutExtMsg < ExtMsg < Msg .
op open? : Configuration -> Bool .
 eq open?(C C') = open?(C) or open?(C') .
```

```
 eq open?(O) = false .
 eq open?(M) = M :: ExtMsg .
```

Actually sending an external message may be more complex than just forwarding the message through the gateway object. External messages may not be represented in the same way in the internal configuration. For example, a simple output message in the internal configuration may need to be mapped to a client object that initiates the communication to deliver the message. Operators `in-adapter` and `out-adapter` are defined to perform these mappings from external message client configurations to internal messages.

An example of an output adapter for a pacemaker message to beat the heart may be:

```
eq out-adapter(shock)
    = createSendReceiveClient(pacer-client, "localhost", 4451, "SetLeadVoltage 5V")
```

In this example, the message `shock` is transformed into a client object which sends a message on port 4451 with the string `"SetLeadVoltage 5V"` indicating that the proxy server will then proceed to set a 5V voltage on the pacemaker lead.

6.2.1 One-Round Communication Clients

Once the external message is mapped into a client configuration, we define the rewrite rules to specify how the communication protocol works with the external device. Here we describe a simple `SEND-RECEIVE-CLIENT` which is responsible for establishing communication, sending a message, receiving a reply, and then closing the communication. Although simple, this type of protocol is sufficient for most of the communication for medical devices we have used in our case studies.

```
(mod SEND-RECEIVE-CLIENT is ...
  op createSendReceiveClient : Oid String Nat String -> Configuration .
   eq createSendReceiveClient(CLIENT, ADDRESS, PORT, SEND-CONTENTS)
      = < CLIENT : SendReceiveClient | ... >
      createClientTcpSocket(socketManager, CLIENT, ADDRESS, PORT) .
  op msg-received : Oid String -> InExtMsg .
...endm)
```

After creating the client and establishing communication, the client goes into one round of sending and receiving before the socket is closed. Once the socket is closed, the entire client object is converted into one reply message to be delivered to the internal configuration using the operator `msg-received`.

```
--- send contents
rl createdSocket(CLIENT, socketManager, SOCKET-DST)
   < CLIENT : SendReceiveClient | ... send-contents : SEND-CONTENTS >
   => < CLIENT : SendReceiveClient | ... > send(SOCKET-DST, CLIENT, SEND-CONTENTS) .
--- receive contents
rl sent(CLIENT, SOCKET-DST) < CLIENT : SendReceiveClient | ... >
   => < CLIENT : SendReceiveClient | ... > receive(SOCKET-DST, CLIENT) .
--- close socket
rl received(CLIENT, SOCKET-DST, RECEIVE-CONTENTS) < CLIENT : SendReceiveClient | ... >
   => < CLIENT : SendReceiveClient | ... recv-contents : RECEIVE-CONTENTS >
      closeSocket(SOCKET-DST, CLIENT) .
--- done
rl closedSocket(CLIENT, SOCKET-DST, "")
   < CLIENT : SendReceiveClient | ... recv-contents : RECEIVE-CONTENTS >
   => msg-received(CLIENT, RECEIVE-CONTENTS) .
```

6.3 Mapping Logical Time to Physical Time

As mentioned earlier, time advancement of the system is achieved by defining the *tick* and *mte* operators. Ideally the system continuously evolves over time (possibly nondeterministically). Of course, we cannot capture the notion of continuous time without abstractions in the model, so to advance time discretely, an *mte* (maximum time elapsable) operator is introduced. A correctly defined *mte* operator ensures that if a system is in state S, then for any time $T < mte(S)$, no 0-time rewrite rules (state transitions) can apply to $tick(S, T)$. That is, if a system is in state S, and $T \leq mte(S)$, then $tick(S, T)$ will be equivalent to the state S advancing in continuous time for T time units. This ideal semantics of time is shown on the left side of Figure 6.2. The figure shows that 0-time rewrite rules are assumed to take zero time, and ideally, the system continuously evolves over time between the 0-time rewrite rules.

Of course, in a real execution of the model, the ideal notion of time with 0-time rewrite rules and time-advancing rules is only an idealized abstraction. Performing rewrites cannot take zero time, and we cannot continuously rewrite states of the system over time. We could of course create a model in discrete time with very fine time granularity and drive it by a high frequency clock like in hardware. However, this would introduce a lot of unnecessary overhead in terms of communication of timing messages and performing

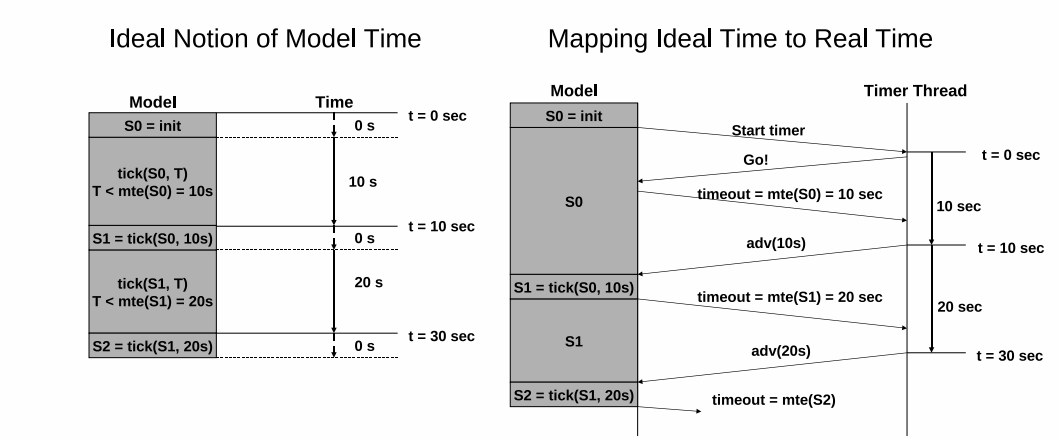

Figure 6.2: From Ideal Time Advancement Semantics to Physical Time Advancement

rewrite rules to change the model for every clock tick. We resolve this problem by observing that the actual internal state of the model is not important at most instants in time, unless it is communicating with the external world. The model states only generate output messages with 0-time rewrite rules, so we can essentially let the model in state S remain unaffected by the passage of time until the next time instant in which a 0-time rewrite rule can be applied; this is exactly $mte(S)$ time units later. This method of driving execution is shown in the right part of Figure 6.2. We have created a dedicated timer server thread (in Java) that has access to the system time. When the execution of the wrapped model starts, it will send a start request which includes the time units of the model or the minimum granularity of time for model execution in milliseconds. Once the timer thread processes all the initial information, it will send a *Go!* message to signal the model to start executing. The model then calculates the maximum time elapsable (which is 10 seconds in the example) and sends this information to the timer thread. The model then proceeds to sleep until the timer thread wakes it in time for the next 0-time rewrite rule. The process then continues. There are two key points to notice about this example. The input and output messages from the model may be delayed by an amount of time equal to the communication jitter plus the time to complete rewriting. Normally this delay is on the order of 10 ms, but this is still suitable for medical devices, which normally receive commands on the order of seconds or more. Also, the timer thread sets the timeout from the last time it sent a time advancement message to the model and not from the time it receives the *mte* message from the model. This ensures that clock skew and jitter are bounded over time.

6.3.1 Synchronous Timed Execution

The *communication wrapper* (`commwrap`) is represented as an object with the attributes for the communication state, the socket information for communication, and the internal wrapped (Real-Time Maude) system

model being executed. The top level system is of sort `CommWrapConfiguration` for any communicating model.

```
op commwrap : Configuration -> CommWrapConfiguration .
op wrap-client : Configuration -> Configuration .
 eq wrap-client(C) = < client : TickClient |
    state : start, internal : [ {C} in time 0 ], socket-name : no-oid > .
op init-client : -> CommWrapConfiguration .
 eq init-client = commwrap( <> wrap-client(internal)
    createClientTcpSocket(socketManager, client, addr, port) ) .
```

The communication wrapper initializes a wrapped communication client that receives messages from the tick server (a Java thread executing in real-time that sends it messages for time advancement). After creating the TCP socket, the first message sent from the client to the tick server is the time-granularity (`time-grain`), which is a rational number specifying the number of milliseconds in one time unit. Then, the actual execution starts when the communication wrapper receives a `GO` message from the tick server. The time when the tick server sends the `GO` message is the starting point from which time elapses are being measured. Upon receiving the `GO` message, the formal model will immediately start to execute (`state : run`).

```
rl [send-init] :
 commwrap( <> createdSocket(...) < client : TickClient | ... > )
  => commwrap( <> < client : TickClient | ... > send(..., string(time-grain)) ) .
rl [wait-for-go] :
 commwrap( <> sent(...) < client : TickClient | ... > )
  => commwrap( <> < client : TickClient | ... > receive(...) ) .
rl [start-running] :
 commwrap( <> received(..., "GO\r\n") < client : TickClient | ... > )
  => commwrap( <> < client : TickClient | state : run, ... > .
```

The formal model executes until `mte` becomes non-zero (no other 0-time rewrite rules can be applied), and the model sends a message to request the next time advancement message after the maximum time elapse and blocks. After sending this waiting duration, the tick server will sleep for this time duration and then send a time advancement message when the time has expired. The model will then advance time (tick)

the model for the time duration expired and perform 0-time rewrite rules. The model now blocks again for the next `mte`, and the cycle repeats.

```
crl [request-wait-timer] :
 commwrap( <>
   < client : TickClient |
      state : run,
      internal : [ {C} in time T ], ... > )
 => commwrap( <>
   < client : TickClient |
      state : request, ... >
   send(..., string(mte(C, T))) )
 if mte(C,T) :: TimeInf /\ mte(C,T) > 0 /\ not open?(C) . ...

rl [block] :
 commwrap( <> sent(...)
   < client : TickClient |
      state : request, ... > )
 => commwrap( <>
   < client : TickClient |
     state : wait, ... >
     receive(SOCKET-NAME, client) ) .

rl [wake-up] :
 commwrap( <> received(..., ADV-STR)
   < client : TickClient |
      state : wait, ... > )
 => commwrap( <>
   < client : TickClient |
      state : run,
      internal : [ {tick(C, rat(ADV-STR))} in time rat(ADV-STR) in time T ], ... > ) .
```

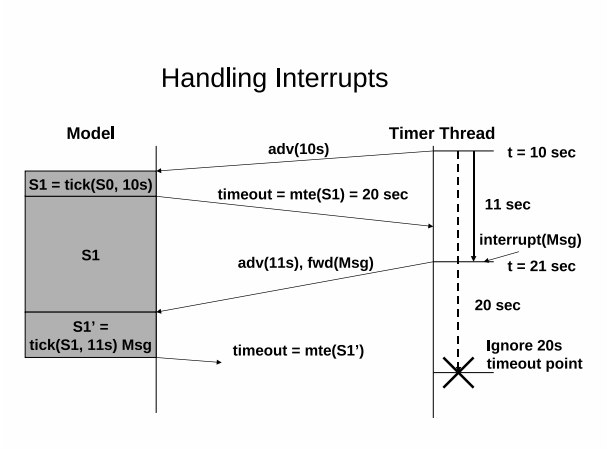

Figure 6.3: Handling Interrupts and Asynchronous Communication Semantics

6.3.2 Handling Asynchronous External Events

So far, the model can only handle synchronous events (polling and blocking communication). However, in general a useful design must be able to react to external events from the environment. For example, an EKG sensor detects a QRS waveform, and sends this information to the pacemaker. This points to the fact that our model needs to be able to handle external events asynchronously.

An external message would trigger a 0-time rewrite rule to receive the message by some object and process it. More precisely, if we have C_M and C_{Ext} as the model configuration and the external (environment) configuration respectively, the maximum time elapse for the system $C_M C_{Ext}$ should be $min(mte(C_M), mte(C_{Ext}))$, where $mte(C_{Ext})$ denotes time duration before the next interrupt message. This semantics is captured by having interrupt messages forwarded by the timer thread, as shown in Figure 6.3. The timer thread will only check for interrupts when it is waiting for the next timeout, so when the interrupt message arrives, it will wake up and immediately forward the interrupt message to the model with the amount of time that has elapsed. Any future timeouts are canceled. Introducing the notion of interrupts requires us to modify the wake-up rule for the model to not only advance time, but also check for potential interrupt messages as well.

```
rl [wake-up] :
commwrap( <> received(client, SOCKET-NAME, INTR-STR)
  < client : TickClient |
  state : wait,
  internal : [ {C} in time T ],
  socket-name : SOCKET-NAME > )
=>
```

```
commwrap( <> < client : TickClient |
  state : run,
  internal : [
    {tick(C, recv->rat(INTR-STR)) recv->conf(INTR-STR)}
      in time recv->rat(INTR-STR) in time T
  ],
  socket-name : SOCKET-NAME > ) .
```

6.4 Case Studies

The model execution framework works for all Real-Time Maude specifications that use only one tick rule. Since the Command Shaper Pattern described in Section 4.2 was specified in this manner, we can directly use instances of the pattern to validate our Real-Time Maude emulation capabilities.

6.4.1 Pacemaker Simulation Case Study

We first apply the Command Shaper to the pacemaker pattern described in Section 4.3.5. Recall, the final wrapper object provided by the pattern is something of the form:

```
(tomod PARAM-PACEMAKER is pr EPR-WRAPPER-EXEC{Safe-Pacer} ...
   eq wrapper-init =
      < pacing-module : EPR-Wrapper{Safe-Pacer} |
         inside :
            < pacing-module : Pacing-Module |
               nextPace : t(0),
               period : safe-dur >,
         val : safe-dur,
         next-val : safe-dur,
         disp : t(period),
         stress-intervals : (nil).Event-Log{Stress-Relax} > .
... endtom)
```

The wrapper is placed around a pacing module, and the initial pacing rate is set as the default safe-duration (`safe-dur` is 750 ms or 80 heart beats per minute). We have of course verified this instantiation in Section 4.3.5. However, with the power of the model emulation framework, we can immediately use this

specification to run with an actual pacemaker. In this paper we demonstrate this emulation capability not on an actual pacemaker but on a pacemaker simulator (a Java widget that receives messages about when to pace and draws a simple line graph resembling an ECG trace). Before the system can be emulated with the pacemaker simulator, some interface information must be provided. The entire module providing all the necessary interface information is shown below:

```
(mod CREATE-TICKER is
   inc PARAM-PACEMAKER .
   inc TIME-CLIENT .
   inc SEND-RECEIVE-CLIENT .

   eq addr = "localhost" .
   eq port = 4444 .

   eq def-te = 1 .
   eq max-te = INF .
   eq time-grain = 10 . --- milliseconds

   op pacer-client : -> Oid .
   eq internal = wrapper-init .

   eq out-adapter(shock)
      = createSendReceiveClient(pacer-client, "localhost", 4451, "shock") .
   eq in-adapter(msg-received(pacer-client, "shocked\n"))
      = set-period(pacing-module, 50) .
endm)
```

The module first indicates that the TCP socket interface to the pacemaker simulator is *localhost* on *port* 4444. The default time elapse for one tick is 1 time unit. The maximum time elapse for one tick step is infinity (i.e. there is no maximum). The duration of one time unit is 10 milliseconds. The time units are in terms of milliseconds since the minimum time granularity provided by the Java time interfaces is 1 millisecond.

The equation for `internal` specifies that the internal configuration to be executed is the configuration defined by `wrapper-init` (as defined in `PARAM-PACEMAKER`). Also, the last two equations specify that the

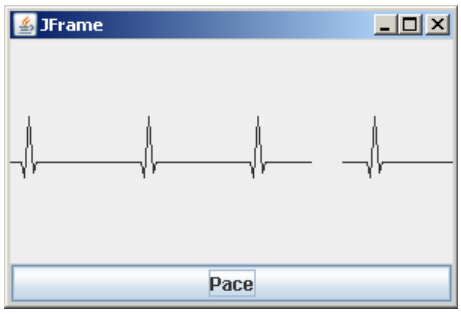

Figure 6.4: Trace from Pacemaker Simulator

output message shock should be mapped to a string "shock" sent over the socket, and upon receiving the acknowledgment message "shocked" set the pacing period to 500 ms (120 bpm - a really fast heart rate). The last equation creates the scenario where a stressful heart rate is always being sent to the pacing module. Since the command shaper pattern should prevent this unsafe behavior, we should see the pacing automatically slow down from 120 bpm after some time interval.

The module is executed by first reflecting the `CREATE-TICKER` module down to Core Maude (with the command `show all CREATE-TICKER`), and executing with the `erew` command. A snapshot of the "ECG" trace of the pacemaker simulator is show in Figure 6.4. For validation we measured the jitter for executing such a system – the physical time required to completely execute 0-time rules and finish communication (Figure 6.5). The results were obtained from a 1.67 GHz Dual-Core Intel Centrino with Maude running in Windows through Cygwin (tracing was turned off). The main thing to notice is that the jitter is mostly below 0.1 seconds and almost never exceeds 0.2 seconds. This amount of jitter is tolerable, since most medical devices need to respond in the order of seconds. The pacemaker is a bit more strict in terms of its timing requirements. To evaluate the suitability for the pacemaker, we plotted the recorded physical time duration between pacing events (Figure 6.6). Notice that in this example the heart rate increases (duration decreases) up to a limit and then the heart rate starts to decrease (duration increases) and the cycle repeats. The jitter in control seems tolerable, since there are no sharp spikes in the graph of the pacing durations.

6.4.2 Syringe Pump Case Study

The pacemaker emulation example was demonstrated through a simulated pacemaker, mostly because current pacemakers do not have external interfaces for setting when to pace (and rightly so). However, for devices such as electronic syringe pumps these interfaces are available. Recall our instantiated pump from 4.3.5 summarized below:

```
(tomod PARAM-PUMP is
```

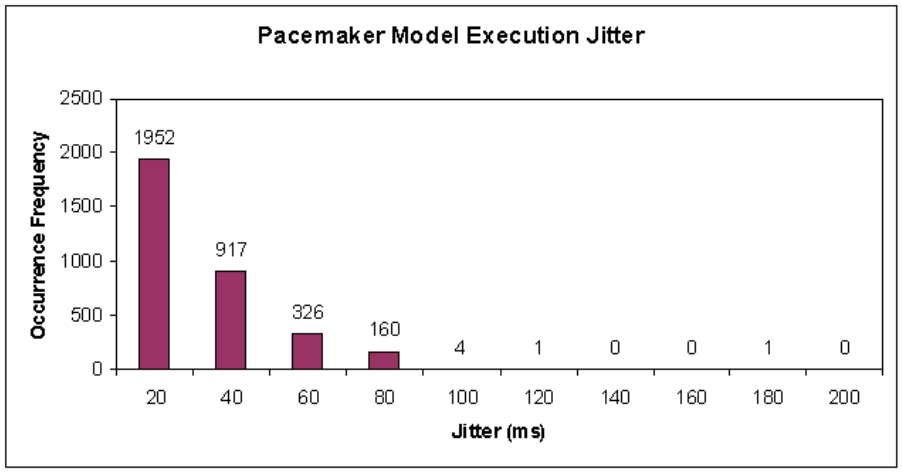

Figure 6.5: Model Execution Jitter Distribution

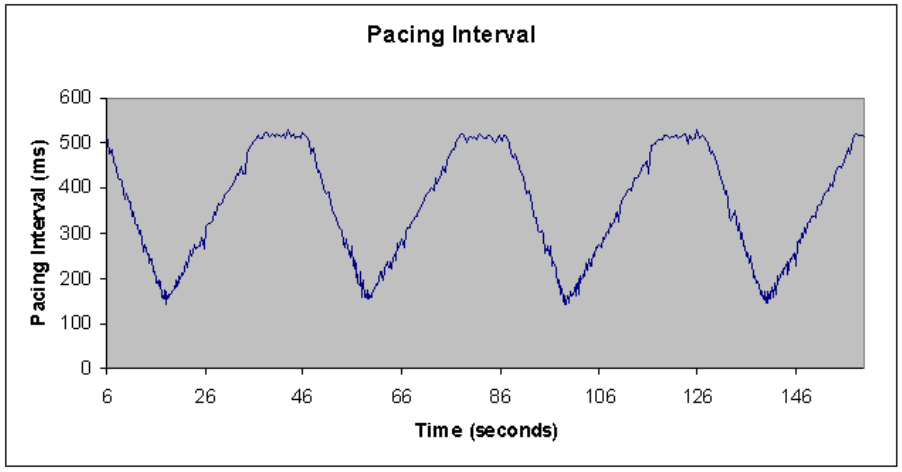

Figure 6.6: Pacing periods recorded by the pacemaker simulator (jitter effects are reflected by noise on the curve)

```
   pr EPR-WRAPPER-EXEC{Safe-Pump} .
   pr DELAY-MSG .
...
   eq msgs-init =
      delay(set-mode(pump-module, bolus), t(9))
      delay(set-mode(pump-module, bolus), t(11))
      delay(set-mode(pump-module, bolus), t(12)) ... .
   eq wrapper-init =
      < pump-module : EPR-Wrapper{Safe-Pump} |
         inside :
            < pump-module : Pump-Module |
               mode : base
            > base,
         val : base,
         next-val : base,
         disp : t(period),
         stress-intervals : (nil).Stress-Relax-Log > .
... endtom)
```

This module shows the initialized wrapper object for the pump, with the initial state being the base rate of infusion. Furthermore, there is also a set of delayed messages that will be sent to the pump. In the term `msgs-init`, the model will send bolus requests at 9 time units, 11 time units, 12 time units, ... after the start of execution for the system. Again, creating a simulated patient model, we can verified the safety of the instantiated pattern in Section 4.3.5. Instantiating the pump is similar to instantiating the pacemaker, except that there are a few more types of output messages.

```
(mod CREATE-TICKER is
   inc PARAM-PUMP .
   inc TIME-CLIENT .
   inc SEND-RECEIVE-CLIENT .

   eq addr = "localhost" .
   eq port = 4444 .
```

```
   eq def-te = 1 .
   eq max-te = INF .
   eq time-grain = 1000 . --- milliseconds

   ops pump-client pump-client' : -> Oid .
   eq internal = wrapper-init msgs-init .

   eq out-adapter(stop)
      = createSendReceiveClient(pump-client, "localhost", 1234, "STP") .
   eq out-adapter(base)
      = createSendReceiveClient(pump-client, "localhost", 1234, "RAT1")
      createSendReceiveClient(pump-client', "localhost", 1234, "RUN") .
   eq out-adapter(bolus)
      = createSendReceiveClient(pump-client, "localhost", 1234, "RAT2")
      createSendReceiveClient(pump-client', "localhost", 1234, "RUN") .
   var S : String .
   eq in-adapter(msg-received(pump-client, S))
      = none .
   eq in-adapter(msg-received(pump-client', S))
      = none .
endm)
```

The model is communicating with *localhost* on *port* 4444. The time granularity is 1 second. The internal configuration being executed is the wrapped pump as well as the set of messages that will deliver bolus requests. The output requests are handled by a Java thread listening on port 1234 and forwarding the request string to the actual *Multi-Phaser NE-500* Syringe Pump (Figure 6.7). A few important requests to the pump are: `STP` stop the pump, `RAT <n>` set infusion rate to `n` ml/hr, `RUN` start the infusion. Reflecting down the `CREATE-TICKER` module and executing with `erew` will now control the physical pump motor!

As a validation for correct pump control, we used a Salter Brecknell 7010SB scale to weigh the amount of liquid infused from the syringe pump over time (Figure 6.8). The data granularity is a bit rough, since the scale can only measure within a precision of 0.1 oz. For this example, to clearly distinguish between two pump states, we let the base rate of infusion be zero (horizontal parts of the graph) and the bolus rate be the maximum infusion rate provided by the pump (positive sloped parts of the graph). Bolus requests

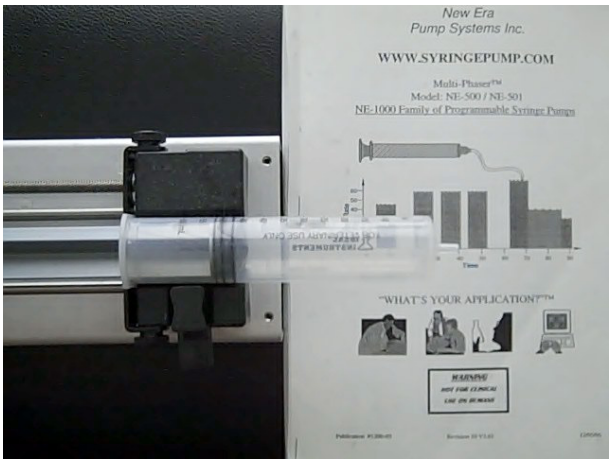

Figure 6.7: Multi-Phaser NE-500 Syringe Pump

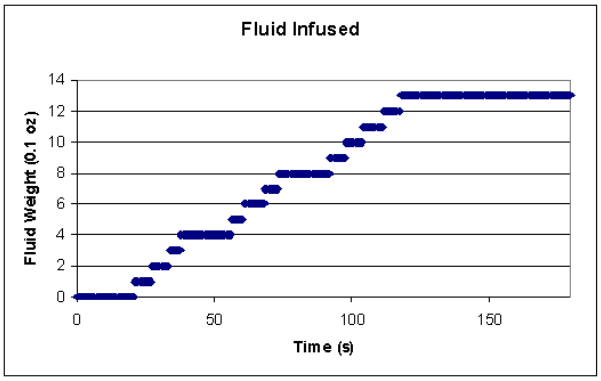

Figure 6.8: Infusion Volume over Time

are continuously sent to the pump. The safety properties require that bolus doses last no longer than 30 seconds, and there must be 10 seconds between bolus doses and at most 3 bolus doses for a window size of 3 minutes. The graph validates that these properties are indeed satisfied for this particular execution of the pump.

Chapter 7

Related Works

This dissertation describes specific examples of formal patterns for medical systems. It is thus natural to discuss the related works in: (1) application of formal methods to medical systems and software, and (2) formal specification and verification of software patterns.

7.1 Formal Methods and Medical Systems

Medical systems have always been quite simple when compared to other safety critical systems such as avionics or nuclear power plants. However, the infamous set of accidents cause by overdosage in the radiation therapy machine Therac-25 [31] immediately brought the issue of medical device safety to the attention of the entire world. This later let to work in formal specifications of radiation therapy machines [25] creating a domain specific language, and the synthesis of safety interlocks based on these specifications [24].

Controlling dosage has always been the a critical and prevalent aspect in most medical systems. The formal specification and analysis of CARA (computer-aided resusitation algoritm) was done simultaneously by different research groups at UPenn [7] and Stony Brook [45] using different specification techniques and verification tools. This later led to the development of formal specification for infusion pumps in general [10][27][53]. Further work has also been done at UPenn in building medical plug-and-play prototypes along with modeling and verification [9][8].

Pacemaker safety has also been emphasized in the recent years. The Software Quality and Research Laboratory (SQRL) at McMaster University issued the Pacemaker Formal Methods Challenge to formally specify and verify an entire pacemaker [32]. Currently, no research group has completed a full specification and verification of the pacemaker challenge, but researchers at UPenn have used it as a case study to explore model-driven design [26]. Aside form safety concerns, recent works have also brought the issue of pacemaker security into the spotlight [22]. Formal methods work has also been done in this area in terms of modeling systems and attackers in the medical domain [11].

On the source code level, static verification techniques have also been proposed for medical software bugs

in industrial scale software [28], and also put into practice at the FDA.

Our medical examples that we used to motivate our *command-shaper* pattern are heavily influenced by the case studies and challenges found in the related works above. We improved our understanding of infusion pump requirements through work on the GIP, and we improved our understanding of pacemaker systems by the resources provided to us by SQRL. That being said, all these related works model and verify these systems individually and separately. We go one step further and describe patterns of software design that can be applied to multiple systems.

Our pattern on network safety is closely related to [30], which also describes a network pattern to handle disconnects. However the protocol described in [30] does not lend itself to having any simple formal specification, and we have chosen to present a much cleaner heartbeat pattern for specification and analysis.

7.2 Formal Methods and Software Patterns

The terminology of software patterns became widely adopted with the Gang of Four (GoF) book [21]. Of course, this later spawned some famous follow up works [13, 48] that design entire systems only around the concept of patterns. In these works, patterns have always been a tool for modularity and portablity of code.

Since patterns were so widely used and many times incorrectly, it was natural to have extensive works on formalizing design patterns [4, 39, 47, 20, 51, 49, 18, 2]. These mostly focus on the original GoF patterns of which many are just structural without any behavior. In this dissertation, we solely focus on patterns that have behavior and ones that can satisfy predefined safety properties. This clearly restricts our patterns to be much smaller than the GoF design patterns, but in turn, we can provide more safety guarantees.

The use of parameterized modules to define patterns is also related to the idea of assume-guarantee aspects of patterns [38]. That is properties that we prove for a pattern, guarantees, will hold if the pattern is instantiated with a proper input. Thus, our work is also related to the plethera works on assume/guarantee reasoning [15, 1, 41, 52]. Of course, our work is not about this specific reasoning technique but about how to apply this technique to create useful patterns in medical systems.

There have also been similar works describing other patterns in other domains such as virtual synchrony in avionic systems [40, 35] and DoS protection for network security [14, 29, 6]. A general survey of various patterns related to distributed systems can be found in [33].

Chapter 8

Conclusion and Future Work

We have demonstrated that formal patterns are not only possible to describe in the simple and executable language of rewriting logic. We have also demonstarted that it is also very practical to use them for medical device safety. In all of our patterns, the actual pattern module description did not exceed 20 rewrite rules. Although our patterns are simple for a first attempt at tackling formal specification of patterns, they are still quite useful and practical for many systems. The point of using patterns is modularity and reusability. That is, patterns offer the hope of making larger, and more complex systems decomposable into simpler and smaller systems by application of these formal patterns. Of course, one could ask the question of whether it is necessary to describe such simple patterns like button debouncers in such rigorous detail. First, formal patterns, even when not yet proven correct, force us to be precise in describing a system. Furthermore, even simple patterns are worthwhile to specify in detail, especially if they occur often. As we have shown in the example of the *message repeater pattern*, the first and most intuitive design can easily be incorrect or at least can reveal the implicit assumptions that we have made with these systems. Of course, all specifications are inevitably bound to a representation. Just as design patterns are bound to object-oriented systems, our current design patterns are bound to actor models. The actor models provide a lot of flexiblity and provide a very general model of concurrency. However, real systems may be much more deterministic that actor models. For example, the button interface patterns are actually solved mostly at a level close to the hardware, which does not actually require the full nondeterminism offered by the actor model. Just as implementing and executing any algorithm requires choosing a particular programming language, specifying a pattern requires choosing a particular modeling paradigm. While this is a current limitation, we imagine that with more research in formal patterns, more general models of computation designed specifically with pattern specification in mind will emerge and allow more general patterns to be specified.

Within the confined framework and scope in which we have formally specified our patterns, we have successfully shown that many patterns can be formally specified using the established theories and frameworks of parameterized modules. Furthermore, we have shown how the parameterized modules are detailed enough to allow proof of generic safety properties or generic bisimulation results. Most of our patterns

can be seen as simple wrappers around objects that try to preserve certain behaviors, and their proofs of correctness are reasonably straightforward. However, it is worth remembering that the proofs apply in very generic contexts, since they are completely parameterized and the properties hold in any bisimulation. This means that repeated composition of many patterns that preserve assumptions and preconditions will allow the development of a large system that satisfies all the proven properties. Not only does this provide a new way of decomposing requirements of a system in a very effective manner, but it also provides the knowledge of when a system can be reused confidently and safely.

This dissertation presents a small but important start for exploring how formal patterns can be applied to the domain of medical systems. It is my hope that the many limitations of this work can be addressed in the future. We make many ideal assumptions such as having perfect clock synchronization between different medical devices. For future work, it would be important to explore the effects of clock skew during synchronization and how this can affect the model and design of the patterns described. Also, since we leverage our models on the power of the existing Maude rewriting logic framework, it was natural for us to use object-oriented actor models to describe system behaviors. However, these abstractions may need to be changed to model accurately the faults that occur in hardware designed systems that have less flexibility in the behaviors and interfaces of components and more parallel and synchronized communication methods. Furthermore, the case studies in this work are currently still limited, and we will of course need to apply our patterns to more complex modelled systems, such as the Generic Infusion Pump model described in [10]. And naturally, applying patterns to larger systems would require composition of many smaller patterns. While some patterns that provide full bisimulation results can be trivially composed, other patterns need much more consideration in order to be composed correctly to preserve the required safety properties. This highly nontrivial problem of pattern composition is one of the major challenges that must be addressed in future work before formal patterns can be ubiquitously used in larger scale medical systems.

Appendix A

Complete Maude Specifications of Models Used

A.1 Basic Modeling

A.1.1 Time Advancement Semantics

The formal model of any timed system starts by defining the notion of time advancement. We use the conventions used in the Real-Time Maude documentation[43] to ensure deterministic timed rewriting without missing any critical timed events. The `tick` operation defines how to advance a system by certain time durations, and the `mte` operation defines the maximum time elapse that can be used to ensure no critical intermediate states are missed. These concepts are specified more precisely in the `TICK-MTE-SEM` module:

```
(tomod TICK-MTE-SEM is inc LTIME-INF .
   op tick : Configuration Time -> Configuration .
   op mte : Configuration -> TimeInf .

   var C : Configuration .
   var T T' : TimeInf .
   crl [advance] : {C} => {tick(C, T)} in time T if T le mte(C) [nonexec] .

   var NC NC' : NEConfiguration .
   eq mte(none) = INF .
   eq mte(NC NC') = minimum(mte(NC), mte(NC')) .
   eq mte(NC) = zero [owise] .
   eq tick(none, T) = none .
   eq tick(NC NC', T) = tick(NC, T) tick(NC', T) .

   op hasMsg? : Configuration -> Bool .
   eq hasMsg?(none) = false .
   eq hasMsg?(NC NC') = hasMsg?(NC) or hasMsg?(NC') .
   eq hasMsg?(M:Msg) = true .
   eq hasMsg?(O:Object) = false .
endtom)
```

The `tick` operator and `mte` (maximum time elapsible) are used to provide deterministic semantics or the `advance` rule (tick rule). We also define `hasMsg?` to test when a configuration contains no messages.

Notice at this point, we have not specified any particular model of time (natural number time, rational number time, etc.), and we try to maintain the generality until specific models need to be defined. Notice that the maximum time elapse (`mte`) of a configuration is the `mte` of the component with minimal `mte` in the configuration. Any component with undefined `mte` will be assumed to have zero `mte`, and time will not be able to advance in that case. This is useful, because for messages that must be delivered instantaneously, time will not advance until the message disappears (is delivered) in the system. Also, tick rules work by ticking each individual component of the configuration (without using any context information) if the `mte` of the entire system is greater than zero.

A.1.2 Real-Time Components

After defining the semantics of time advancement, we define some typical building blocks for real-time systems. These include clocks and timers. Clocks encapsulate time values that are monotonically non-decreasing over time. Clocks can be either running or stopped. A running clock will advance its internal time by the same amount as the time advanced, and a stopped clock will hold its internal time constant over time. Timers encapsulate time values that are monotonically decreasing over time. A timer continues to decrease the encapsulated time by a value equal to the time advancement, and system time is no longer allowed to decrease when a timer's encapsulated time value reaches zero. It is assumed that some significant event happens at the time a timer reaches zero, and the event will instantaneously influence the state of the system before any further time advances. The module `RT-COMP` defines these notions:

```
(fmod RT-COMP is inc LTIME-INF .
   sorts RT-Comp Clock ClockAttr Timer .
   subsorts Clock Timer < RT-Comp .

   ops run stop : -> ClockAttr [ctor] .
   op c : Time ClockAttr -> Clock [ctor] .
   op t : Time -> Timer [ctor] .

   op clock0 : -> Clock . --- started clock
   op timer0 : -> Timer . --- expired timer
   ops stop run : Clock -> Clock .
   op stopped? : Clock -> Bool .
   op value : Clock -> Time .

   op no-timer : -> Timer .


   var T T' : Time .
   var CA : ClockAttr .
```

```
   eq mte(no-timer) = INF .
   eq tick(no-timer, T) = no-timer .

   eq clock0 = c(zero, run) .
   eq timer0 = t(zero) .
   eq stop(c(T, CA)) = c(T, stop) .
   eq run(c(T, CA)) = c(T, run) .
   eq stopped?(c(T, stop)) = true .
   eq stopped?(c(T, run)) = false .
   eq value(c(T, CA)) = T .

   op tick : RT-Comp Time -> RT-Comp .
   op tick : Clock Time -> Clock .
   op tick : Timer Time -> Timer .

   eq tick(c(T, run), T') = c(T plus T', run) .
   eq tick(c(T, stop), T') = c(T, stop) .
   eq tick(t(T), T') = t(T monus T') .

   op mte : RT-Comp -> Time .
   op mte : Clock -> Time .
   op mte : Timer -> Time .

   eq mte(c(T, CA)) = INF .
   eq mte(t(T)) = T .
endfm)
```

A.1.3 Delayed Messages

It is useful to have a notion of delayed messages. Essentially messages wrapped inside an operator with a timer, and the message is delivered (wrapper operator removed) once the timer expires. Not only does this allow modeling communication delay, but this is also necessary as to set inital states of systems that receive inputs over time. Aside from modeling messages with fixed delay, we also define the notion of a `TimeSet` that can model nondeterministic message delay that can take any delay out of a set of times.

```
(tomod DELAY-MSG is
   inc TICK-MTE-SEM .
   pr RT-COMP .

   sort DelayedMsg .
   subsort DelayedMsg < Msg .

   sort TimeSet .
   subsort Time < TimeSet .
```

```
   op __ : TimeSet TimeSet -> TimeSet [assoc comm] .

   op delay : Msg Timer -> DelayedMsg [ctor frozen] .
   op delay : Msg TimeSet -> DelayedMsg [frozen] .

   var M : Msg . var TM : Timer .
   var T : Time .

   ceq delay(M, TM) = M if TM == timer0 .
   eq tick(delay(M, TM), T) = delay(M, tick(TM, T)) .
   eq mte(delay(M, TM)) = mte(TM) .

   var TS : TimeSet .
   eq mte(delay(M, TS)) = zero .
   rl delay(M, T TS) => delay(M, t(T)) .
   rl delay(M, T) => delay(M, t(T)) .
endtom)
```

Delayed messages, are messages wrapped in a delay operator such that they are completely frozen until the delay timer runs out. This is useful for modeling communication delays as well as just to test inputs over time. Delayed messages can also take nondeterministic delays, where the messages may be delayed by any time in a set of possible times.

A.1.4 Event Logs

Naturally, the notion of safety in real-time systems may also depend on the behavior of systems over time. While we can use tools for timed temporal model checking, the counter examples found are quite convoluted to parse and understand. Furthermore, our systems are much simpler and mostly deterministic given a set of initial conditions (the only non-determinism comes from faults). Thus, it is easier for us to keep a log of events over time, and evaluate the safety properties as a predicate on these logs. Essentially this keeps a running history of the system in its current state, and this allows us to evaluate time related properties as just a predicate defined on the current state. Usually history is recorded based on a set of discrete events over time.

```
(fmod EVENT-LOG{E :: TRIV * (sort Elt to |EventType|)} is pr RT-COMP .
   sorts NoEventType{E} EventType{E} Event{E} .
   sorts !MtLog StoppedEventLog{E} EventLog{E} .
   subsorts E$|EventType| NoEventType{E} < EventType{E} .
   subsorts !MtLog < StoppedEventLog{E} < EventLog{E} < RT-Comp .

   op !none : -> NoEventType{E} [ctor] .
   op !e : E$|EventType| Clock -> Event{E} [ctor] .
   op type : Event{E} -> E$|EventType| .
```

```
op elapsed : Event{E} -> Time .
op stop : Event{E} -> Event{E} .
op stopped? : Event{E} -> Bool .

op tick : Event{E} Time -> Event{E} .
op mte : Event{E} -> Time .

var EV : Event{E} . var ET : E$|EventType| .
var C : Clock . var L L' : EventLog{E} .
var SL : StoppedEventLog{E} .
var T T' : Time .

eq type(!e(ET, C)) = ET .
eq elapsed(!e(ET, C)) = value(C) .
eq stop(!e(ET, C)) = !e(ET, stop(C)) .
eq stopped?(!e(ET, C)) = stopped?(C) .

eq tick(!e(ET, C), T) = !e(ET, tick(C, T)) .
eq mte(!e(ET, C)) = mte(C) .

op !nil : -> !MtLog [ctor] .
op __ : Event{E} StoppedEventLog{E} -> EventLog{E} [ctor] .
op stop : !MtLog -> !MtLog .
op stop : EventLog{E} -> StoppedEventLog{E} .
op append : !MtLog !MtLog -> !MtLog .
op append : EventLog{E} StoppedEventLog{E} -> EventLog{E} .

cmb EV SL : StoppedEventLog{E} if stopped?(EV) .

op log : !MtLog NoEventType{E} -> !MtLog .
op log : EventLog{E} EventType{E} -> EventLog{E} .

op tick : !MtLog Time -> !MtLog .
op tick : EventLog{E} Time -> EventLog{E} .
op mte : EventLog{E} -> Time .

eq stop(!nil) = !nil .
eq stop(EV L) = stop(EV) stop(L) .
eq append(!nil, L) = L .
eq append(EV L , L') = EV append(L, L') .
eq log(L, !none) = L .
eq log(L, ET) = !e(ET, clock0) stop(L) [owise] .

eq tick(!nil, T) = !nil .
--- only the latest event needs to be ticked
--- remaining event clocks are stopped
eq tick(EV L, T) = tick(EV, T) L .
eq mte(!nil) = INF .
```

```
   eq mte(EV L) = mte(EV) .
endfm)
```

A.2 Button Related Patterns

A.2.1 Button Press Model

The first step to modeling button related patterns is to understand how to model a button over time. A button can be modeled as a set of press and release events over time. Notice the similarity between the button press model and event log models. However, these are ultimately different in that the button press model is not used for execution and will be processed as an input. So even though button models and event logs are both timed list, they are used differently (e.g. button press model uses absolute time instead of relative time), and we create a separate model.

```
(fmod PRESS-RELEASE is inc LTIME-INF .
   inc CONVERSION .
   sorts Press Release .
   sort Event .
   subsort Press Release < Event .
   sort PressReleaseList .
   sorts MtList PressLastList ReleaseLastList .
   subsort MtList < PressLastList ReleaseLastList .
   subsorts PressLastList < PressReleaseList .
   subsorts ReleaseLastList < PressReleaseList .
   sort Input .
   subsort PressReleaseList < Input .

   op press : Time -> Press .
   op release : Time -> Release .
   op nil : -> MtList .
   op __ : PressLastList Release -> ReleaseLastList .
   op __ : ReleaseLastList Press -> PressLastList .

   var L : PressReleaseList .
   var T : Time .
   op t-last : PressReleaseList -> Time .
   eq t-last(nil) = zero .
   eq t-last(L press(T)) = T .
   eq t-last(L release(T)) = T .

   op valid? : PressReleaseList -> Bool .
   eq valid?(nil) = true .
   eq valid?(L press(T)) = valid?(L) and (t-last(L) lt T) .
   eq valid?(L release(T)) = valid?(L) and (t-last(L) lt T) .
```

```
   var L' : PressReleaseList .
   var E : Event .
   op _->_ : PressReleaseList PressReleaseList ~> PressReleaseList .
   op _->_ : Event PressReleaseList ~> PressReleaseList .
   eq L -> nil = L .
   eq L -> (L' E) = (L -> L') E .
   eq E -> L = (nil E) -> L .

   op length : Input -> Nat .
   eq length(nil) = 0 .
   eq length(L E) = s length(L) .
endfm)
```

After defining button input as a list, we can also convert this list to a set of delayed messages for defining the initial state of inputs in an actor model.

```
(tomod PRESS-RELEASE-MSGS is inc PRESS-RELEASE .
   inc DELAY-MSG .

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

   var L : PressReleaseList .
   var O : Oid .
   var T : Time .
   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)) .
endtom)
```

A.2.2 Button Fault Models

We can also use the model of button presses to define fault relations for the inputs.

```
(fmod BUTTON-FAULTS is pr PRESS-RELEASE .
   vars T T' : Time .
   vars R R' : Time .
   vars I I' I'' : Input .

   op sub-event : Press Input -> Bool .
   op sub-event : Release Input -> Bool .
   eq sub-event(press(T), nil) = false .
   eq sub-event(press(T), I release(T')) = sub-event(press(T),I) .
   eq sub-event(press(T), I press(T')) = (T == T') or sub-event(press(T),I) .
   eq sub-event(release(T), nil) = false .
   eq sub-event(release(T), I press(T')) = sub-event(release(T),I) .
```

```
   eq sub-event(release(T), I release(T')) = (T == T') or sub-event(release(T),I) .

   op sub-input : Input Input -> Bool .
   eq sub-input(nil, I') = true .
   eq sub-input(I press(T), I') = sub-input(I,I') and sub-event(press(T), I') .
   eq sub-input(I release(T), I') = sub-input(I,I') and sub-event(release(T), I') .

   op same-input : Input Input -> Bool .
   eq same-input(I,I') = sub-input(I,I') and sub-input(I',I) .

   op bounce-duration : -> Time .
   op space-duration : -> Time .
   op bounce-fault : Input Input -> Bool .
   eq bounce-fault(nil,nil) = true .
   eq bounce-fault(I release(T),I') = bounce-fault(I,I') .
   eq bounce-fault(I, I' release(T)) = bounce-fault(I,I') .
   eq bounce-fault(I press(T), I' press(T)) = bounce-fault(I,I') .
   ceq bounce-fault(I press(T), I' press(T')) = bounce-fault(I, I' press(T'))
      if T le (T' plus bounce-duration) /\ T gt T' .
   eq bounce-fault(I,I') = false [owise] .

   op stuck-duration : -> Time .
   op stuck-fault : Input Input -> Bool .
   eq stuck-fault(nil,nil) = true .
   eq stuck-fault(I release(T), I') = same-input(I release(T), I') .
   eq stuck-fault(I press(T), I' release(T')) = stuck-fault(I press(T), I') .
   ceq stuck-fault(I, I') = same-input(I,I') or stuck-fault(I,I'')
      if I'' release(T) press(T') := I' .
   eq stuck-fault(I, I') = false [owise] .

   op phantom-thresh : -> Time .
   op phantom-fault : Input Input -> Bool .
   eq phantom-fault(nil, nil) = true .
   eq phantom-fault(I press(T), I' press(T)) = phantom-fault(I,I') .
   eq phantom-fault(I press(T) release(T'), I' press(T) release(T')) = phantom-fault(I,I') .
   ceq phantom-fault(I press(T) release(T'), I' press(R) release(R')) = phantom-fault(I,I' press(R) release(R'))
      if T gt R' /\ T' lt (T plus phantom-thresh) .
   eq phantom-fault(I,I') = false [owise] .
endfm)
```

A.2.3 Debouncer Pattern

This is the full specification of the Debouncer Pattern described in detail in Section 3.3.3.

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

   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 .

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

   --- assumes that all messages sent to external objects are of sort |OutMsg|
endoth)

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

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

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

   vars T : Time .
   var O : Oid .
   var TM : Timer .
   vars C C' : NEConfiguration .

   op get-inside : Object ~> Configuration [frozen] .
   eq get-inside(< O : !Debouncer{|O|} | inside : C >)
      = C .

   op pi-inside : Configuration -> Configuration .

   eq pi-inside(C C') = pi-inside(C) pi-inside(C') .
   eq pi-inside(< O : !Debouncer{|O|} | >) = get-inside(< O : !Debouncer{|O|} | >) .
   eq pi-inside(C) = C [owise] .

   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)) .

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

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

   crl < O : !Debouncer{|O|} | timer : TM >
   => < O : !Debouncer{|O|} | timer : no-timer >
   if TM == timer0 .

   var IM OM : Msg .
   crl IM < O : !Debouncer{|O|} | inside : C >
      => < O : !Debouncer{|O|} | inside : IM C >
      if |dest|(IM) == O /\ IM =/= |press|(O) .
   crl < O : !Debouncer{|O|} | inside : OM C >
      => < O : !Debouncer{|O|} | inside : C > OM
      if |dest|(OM) =/= O .
endtom)
```

A.2.4 Dephantomizer Pattern

This is the full specification of the Dephantomizer Pattern described in Section 3.4.1.

```
(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)

(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 .
   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)) .

   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 .

   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)
```

A.2.5 Stuck Detection Pattern

This is the full specification of the stuck detection pattern described in Section 3.5.

```
(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)

(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 .

   vars T : Time .
```

```
   var O : Oid .
   var TM : Timer .
   var C : Configuration .
   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)) .

   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 .

   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)
```

A.3 Command-Shaper Pattern

The input theory for the Command-Shaper Pattern, which describes the stress-relax safety requirements.

```
(fth SAFE-STATE is
  inc TOTAL-ORDER * (
    sort Elt to Val,
    op _<=_ to _<=risk_
    ) .
  pr STRESS-RELAX-LOG .

  ops min-val max-val crit-val safe-val : -> Val .

  eq min-val <=risk safe-val = true .
  eq safe-val <=risk crit-val = true .
  eq crit-val <=risk max-val = true .

  op period : -> Time . --- period of wrapper dispatch

  op delta : Val Val -> Val .
  op tdelta-min : Val Val -> TimeInf .

  var V V' V'' : Val .
  eq [no-delta] : delta(V, V) = V .
  ceq [delta-bound] :
    ((V <=risk V'') and (V'' <=risk V')) or
    ((V' <=risk V'') and (V'' <=risk V)) = true
    if delta(V, V') = V'' .
  ceq [delta-mono] : delta(V, V') <=risk delta(V, V'') = true if V' <=risk V'' .
---  ceq [delta-max] : delta(V,V') = delta(V,V'') if delta(V,V') =/= V' /\ V' <=risk V'' .
  ceq [delta-max] : (delta(V,V') == delta(V,V'')) = true if delta(V,V') =/= V' /\ V' <=risk V'' .

  eq [no-tdelta] : tdelta-min(V, V) = zero .
  ceq [ind-tdelta] : tdelta-min(V, V') = tdelta-min(delta(V,V'), V') plus period
    if V =/= V' .
  ceq [tdelta-mono] : tdelta-min(V', V) le tdelta-min(V'', V) = true
    if V <=risk V' /\ V' <=risk V'' .

  op safe? : Stress-Relax-Log -> Bool .
  op norm : Stress-Relax-Log -> Stress-Relax-Log .

  var T T' : Time .
  var CA : ClockAttr .
  var SRE : SREvent .
  var L L' : Stress-Relax-Log .
  eq [initially-safe] : safe?(nil) = true .
  --- instantaneous safety equivalent to safety with stopped time
  eq [stopped-safe] : safe?(E(SRE, c(T, CA)) L) =
```

```
    safe?(E(SRE, c(T, stop)) L) .
  --- safety implies safety of all sub-logs
  eq [sub-safe] : safe?(E(SRE, c(T, CA)) L) implies safe?(L) = true .
  --- decreasing stress durations preserves safety
  ceq [stress-safe] : safe?(E(!stress, c(T', CA)) L) implies safe?(E(!stress, c(T, CA)) L) = true
    if T' ge T .
  --- safety is preserved while in a relaxed state
  eq [relax-safe] : safe?(E(!relax, c(T, CA)) L)  =
    safe?(L) .

  --- normalization preserves log structure
  --- norm-tick-sort
  cmb tick(norm(L), T) : Stress-Relax-Log
      if tick(L, T) : Stress-Relax-Log .
  --- norm-app-sort
  cmb append(L', stop(norm(L))) : Stress-Relax-Log
    if append(L', stop(L)) : Stress-Relax-Log .
  --- normalization preserves safety
  eq [norm-tick-safe] : safe?(tick(norm(L), T)) =
    safe?(tick(L, T)) .
  ceq [norm-log-safe] : safe?(append(L', stop(norm(L)))) =
    safe?(append(L', stop(L)))
    if append(L', stop(L)) : Stress-Relax-Log .
  --- normalization is idempotent
  eq [norm-tick] : norm(tick(norm(L), T)) = norm(tick(L, T)) .
  eq [norm-app] : norm(append(L', stop(tick(norm(L), T))))
    = norm(append(L', stop(tick(L, T)))) .
endfth)
```

The internal wrapped object used by the Command-Shaper is defined as another input theory as follows:

The structural part of the Command-Shaper Pattern is defined as follows:

to enforce the safety defined in the input theory `SAFE-STATE` is as follows:

```
(oth WRAPPED-OBJECT is
  inc SAFE-STATE .
  inc TICK-MTE-SEM .

  class Wrapped | set-val : Val .

  sort InMsg .
  sort OutMsg .
  subsorts InMsg OutMsg < Msg .

  msg set-val : Oid Val -> InMsg .

  var O : Oid .
  vars V V' : Val .
```

```
  eq mte(set-val(O, V)) = zero .

  rl [wrapped-recv] : set-val(O, V)
    < O : Wrapped | set-val : V' >
    => < O : Wrapped | set-val : V > .

  --- assumes that all messages sent to external objects are of sort OutMsg
  --- assumes that set-val is the only message of sort InMsg sent from external objects
endoth)
```

We now describe the behavioral part of the Command-Shaper Pattern to enforce safety constraints described in SAFE-STATE on the wrapped object defined in WRAPPED-OBJECT:

```
(tomod EPR-WRAPPER{X :: WRAPPED-OBJECT} is
  class EPR-Wrapper{X} | inside : NEConfiguration,
    next-val : X$Val, val : X$Val, disp : Timer,
    stress-intervals : Stress-Relax-Log .

  op _get-next-val : Object ~> X$Val [frozen] .
  op _get-val : Object ~> X$Val [frozen] .

  op _set-next-val_ : Object X$Val ~> Object [frozen] .
  op _set-val_ : Object X$Val ~> Object [frozen] .
  op _log-stress_ : Object EventType{SREvent} ~> Object [frozen] .
  op _norm-stress : Object ~> Object [frozen] .
  op _deliver : Object ~> Object [frozen] .

  var O : Oid .
  var V V' : X$Val .
  var L : Stress-Relax-Log .
  var E : EventType{SREvent} .
  var C : NEConfiguration .

  eq < O : EPR-Wrapper{X} | next-val : V > get-next-val = V .
  eq < O : EPR-Wrapper{X} | val : V > get-val = V .

  eq < O : EPR-Wrapper{X} | next-val : V > set-next-val V'
    = < O : EPR-Wrapper{X} | next-val : V' > .
  eq < O : EPR-Wrapper{X} | val : V > set-val V'
    = < O : EPR-Wrapper{X} | val : V' > .
  eq < O : EPR-Wrapper{X} | stress-intervals : L > log-stress E
    = < O : EPR-Wrapper{X} | stress-intervals : log(L, E) > .
  eq < O : EPR-Wrapper{X} | stress-intervals : L > norm-stress
    = < O : EPR-Wrapper{X} | stress-intervals : norm(L) > .
  eq < O : EPR-Wrapper{X} | inside : C, val : V > deliver
    = < O : EPR-Wrapper{X} | inside : (set-val(O, V) C) > .
endtom)
```

```
(fmod WRAPPER-AUX{X :: WRAPPED-OBJECT} is inc EPR-WRAPPER{X} .

  op cap : X$Val -> X$Val .
  op stress? : X$Val -> Bool .
  ops toStress? toRelax? : X$Val X$Val -> Bool .
  op inEnv? : X$Val Stress-Relax-Log -> Bool .

  var V V' : X$Val .
  ceq cap(V) = min-val if V <=risk min-val .
  ceq cap(V) = max-val if max-val <=risk V .
  eq cap(V) = V [owise] .

  eq stress?(V) = not (V <=risk crit-val) .
  eq toStress?(V, V') = not stress?(V) and stress?(V') .
  eq toRelax?(V, V') = stress?(V) and not stress?(V') .

  var L : Stress-Relax-Log .
  ceq [inenv-unreachable] : inEnv?(V, L) = false
    if tdelta-min(V, crit-val) == INF /\ stress?(V) .
  ceq [inenv-stress] : inEnv?(V, L) = safe?(L) and
    safe?(log(tick(L, tdelta-min(V, crit-val) plus period), !relax))
    if stress?(V) /\ tdelta-min(V, crit-val) :: Time .
  ceq [inenv-relax] : inEnv?(V, L) = safe?(L) if not stress?(V) .
endfm)

(tomod EPR-WRAPPER-EXEC{X :: WRAPPED-OBJECT} is
  inc WRAPPER-AUX{X} .

  op log-entry : X$Val X$Val ~> EventType{SREvent} .
  op next-val : Object ~> X$Val [frozen] .

  var O : Oid .
  var L : Stress-Relax-Log .
  var V V' V'' : X$Val .
  var T T' : Time .

  ceq log-entry(V, V') = !stress if toStress?(V, V') .
  ceq log-entry(V, V') = !relax if toRelax?(V, V') .
  eq log-entry(V, V') = none [owise] .

  ceq next-val(< O : EPR-Wrapper{X} |
    val : V, next-val : V', stress-intervals : L >)
    = delta(V, safe-val)
    if not inEnv?(delta(V, V'),
      log(L, log-entry(V, delta(V, V')))) .
  ceq next-val(< O : EPR-Wrapper{X} |
    val : V, next-val : V', stress-intervals : L >)
    = delta(V, V')
```

```
    if inEnv?(delta(V, V'),
      log(L, log-entry(V, delta(V, V')))) .

  var C : NEConfiguration .
  var M : X$OutMsg .

  crl [dispatch] :
    < O : EPR-Wrapper{X} |
      disp : t(T), val : V, next-val : V' >
    => (< O : EPR-Wrapper{X} | disp : t(period),
      next-val : V'',
      val : V''
      >
      log-stress log-entry(V, V''))
      norm-stress deliver
    if V'' := next-val(< O : EPR-Wrapper{X} | >)
      /\ T = zero .
  rl [recv] : set-val(O, V) < O : EPR-Wrapper{X} | >
    => < O : EPR-Wrapper{X} | next-val : cap(V) > .
  rl [forward] : < O : EPR-Wrapper{X} | inside : C M >
    => < O : EPR-Wrapper{X} | inside : C > M .

  eq mte(< O : EPR-Wrapper{X} | inside : C, disp : t(T) >)
    = minimum(T, mte(C)) .
  eq tick(< O : EPR-Wrapper{X} |
    stress-intervals : L,
    disp : t(T),
    inside : C >, T')
    = < O : EPR-Wrapper{X} |
      stress-intervals : tick(L, T'),
      disp : tick(t(T), T'),
      inside : tick(C, T') > .
endtom)
```

A.4 Heartbeat Pattern

Here, we provide the full specification of the final correct heartbeat pattern described in Section 5.3.3.

```
(oth COMM-PAIR is
   inc LTIME-INF .
   class Sender .
   class Receiver .

   op dest : Msg -> Oid .
   op sid : -> Oid .
   op rid : -> Oid .
   op repeat-msg? : Msg -> Bool .
```

```
   msg safe-msg : Oid -> Msg .

   op repeat-time : -> Time .
   op timeout : -> Time .

   eq repeat-time lt timeout = true .

   op init-sender : -> Configuration .
   op init-receiver : -> Configuration .
endoth)

(tomod REPEATER{X :: COMM-PAIR} is
   pr DELAY-MSG .
   class SenderWrap | internal : Configuration, time : Timer,
            repeat-msg : Configuration .
   class ReceiverWrap | internal : Configuration, time : Timer,
            repeat-msg : Configuration .

   op repeated : Oid Msg -> Msg .
   op repeated? : Msg -> Bool .

   vars M M' : Msg .
   var C : Configuration .
   var TM : Timer .
   var T : Time .

   var O : Oid .

   eq dest(repeated(O,M)) = O .
   eq repeat-msg?(repeated(O,M)) = false .
   eq repeated?(repeated(O,M)) = true .
   eq repeated?(M) = false [owise] .

   eq mte(< sid : SenderWrap | internal : C, time : TM >)
      = minimum(mte(C), mte(TM)) .
   eq mte(< rid : ReceiverWrap | internal : C, time : TM >)
      = minimum(mte(C), mte(TM)) .

   eq tick(< sid : SenderWrap | internal : C, time : TM >, T)
   = < sid : SenderWrap | internal : tick(C,T), time : tick(TM, T) > .
   eq tick(< rid : ReceiverWrap | internal : C, time : TM >, T)
   = < rid : ReceiverWrap | internal : tick(C, T), time : tick(TM, T) > .

   op init-sender-wrap : -> Configuration .
   eq init-sender-wrap =
   < sid : SenderWrap | internal : init-sender, time : no-timer,
      repeat-msg : none > .
```

```
op init-receiver-wrap : -> Configuration .
eq init-receiver-wrap =
< rid : ReceiverWrap | internal : init-receiver, time : no-timer,
   repeat-msg : none > .

--- send a nonrepeated message
crl [send-nonrepeat] : < sid : SenderWrap | internal : M C, time : TM >
=> < sid : SenderWrap | internal : C , time : no-timer,
repeat-msg : none > M
if dest(M) =/= sid /\ repeat-msg?(M) == false .

--- send a repeated message
crl [send-repeat] : < sid : SenderWrap | internal : M C, time : TM >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
repeat-msg : M >  M
if dest(M) =/= sid /\ repeat-msg?(M) == true .
--- need dest(M) == rid ???

--- repeat a message
crl [repeat] : < sid : SenderWrap | internal : C, time : TM, repeat-msg : M >
=> < sid : SenderWrap | internal : C , time : t(repeat-time),
repeat-msg : M >  repeated(dest(M), M)
if TM == timer0 .

--- forward msg to sender
crl [fwd-to-sender] : M < sid : SenderWrap | internal : C >
=> < sid : SenderWrap | internal : C M >
if dest(M) == sid .

--- forward a message that will be repeated
crl [fwd-with-repeat] : M < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : M C, time : t(timeout),
repeat-msg : M >
if dest(M) == rid /\ repeat-msg?(M) == true .

--- forward a message that is not repeated
crl [fwd-no-repeat] : M < rid : ReceiverWrap | internal : C, time : TM >
=> < rid : ReceiverWrap | internal : M C, time : no-timer >
if dest(M) == rid /\ repeat-msg?(M) == false /\ repeated?(M) == false .

--- receiving a message repeat that was seen
rl [repeated-seen] : repeated(rid,M) < rid : ReceiverWrap | internal : C, time : TM,
repeat-msg : M >
=> < rid : ReceiverWrap | time : t(timeout) > .

--- receiving a message repeat that was not seen
crl [repeat-not-seen] : repeated(rid,M) < rid : ReceiverWrap | repeat-msg : M' >
=> < rid : ReceiverWrap | >
```

```
   if M =/= M' .

   --- repeat message timeout
   crl [repeat-timeout] : < rid : ReceiverWrap | internal : C, time : TM >
   => < rid : ReceiverWrap | internal : safe-msg(rid) C, time : no-timer,
   repeat-msg : none >
   if TM == timer0 .

   --- forward out from receiver
   crl [fwd-from-recv] : < rid : ReceiverWrap | internal : M C >
   => < rid : ReceiverWrap | internal : C > M
   if dest(M) =/= rid .

endtom)
```

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[2] Jean-Raymond Abrial and Thai Son Hoang. Using Design Patterns in Formal Methods: An Event-B Approach. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC*, volume 5160 of *LNCS*, pages 1–2. Springer, 2008.

[3] Gul Agha, Svend Frlund, Rajendra Panwar, and Daniel Sturman. A linguistic framework for dynamic composition of dependability protocols, 1993.

[4] Paulo S. C. Alencar, Donald D. Cowan, and Carlos José Pereira de Lucena. A formal approach to architectural design patterns. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME*, volume 1051 of *LNCS*, pages 576–594. Springer, 1996.

[5] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, August 1977.

[6] M. AlTurki, J. Meseguer, and C. Gunter. Probabilistic modeling and analysis of DoS protection for the ASV protocol. *Electr. Notes Theor. Comput. Sci.*, 234:3–18, 2009.

[7] Rajeev Alur, David Arney, Elsa L. Gunter, Insup Lee, Jaime Lee, Wonhong Nam, Frederick Pearce, Steve Van Albert, and Jiaxiang Zhou. Formal specifications and analysis of the computer-assisted resuscitation algorithm (cara) infusion pump control system. *International Journal on Software Tools for Technology Transfer*, 2004.

[8] David Arney, Sebastian Fischmeister, Julian M Goldman, Insup Lee, and Robert Trausmuth. Plug-and-play for medical devices: Experiences from a case study. *Biomedical Instrumentation & Technology*, 43(4):313–317, 2009.

[9] David Arney, Julian M Goldman, Susan F Whitehead, and Insup Lee. Synchronizing an x-ray and anesthesia machine ventilator: A medical device interoperability case study. *International Conference on Biomedical Electronics and Devices, BioDevices*, 2009.

[10] David Arney, Raoul Jetley, Paul Jones, Insup Lee, and Oleg Sokolsky. Formal methods based development of a pca infusion pump reference model: Generic infusion pump (gip) project. *Proceedings of the High Confidence Medical Device Software and Systems (HCMDSS)*, 2007.

[11] David Arney, Krishna K Venkatasubramanian, Oleg Sokolsky, and Insup Lee. Biomedical devices and systems security. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 2376–2379. IEEE, 2011.

[12] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1):386–414, 2006.

[13] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Addison-Wesley, 1996.

[14] Rohit Chadha, Carl A. Gunter, José Meseguer, Ravinder Shankesi, and Mahesh Viswanathan. Modular preservation of safety properties by cookie-based DoS-protection wrappers. In *Proc. FMOODS 2008*, volume 5051 of *LNCS*, pages 39–58. Springer, 2008.

[15] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[17] G. Denker, J. Meseguer, and C. Talcott. Rewriting semantics of meta-objects and composable distributed services. Technical report, In Proceedings of the 3rd International Workshop on Rewriting Logic and Its Applications, 2000.

[18] Jing Dong, Paulo S. C. Alencar, Donald D. Cowan, and Sheng Yang. Composing pattern-based components and verifying correctness. *Journal of Systems and Software*, 80(11):1755–1769, 2007.

[19] F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, SRI International, 1999.

[20] Amnon H. Eden and Yoram Hirshfeld. Principles in formal specification of object oriented design and architecture. In Darlene A. Stewart and J. Howard Johnson, editors, *CASCON*, page 3. IBM, 2001.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] Daniel Halperin, Thomas S Heydt-Benjamin, Benjamin Ransford, Shane S Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 129–142. IEEE, 2008.

[23] Medical Devices and Medical Systems - Essential Safety Requirements for Equipment Comprising the Patient-Centric Integrated Clinical Environment (ICE). http://mdpnp.org/uploads/ICE_Part_I_draft_21Dec2008_N30_ web.pdf.

[24] J. JACKY. Verification, analysis and synthesis of safety interlocks. *Technical Report 91-04-01, Department of Radiation Oncology RC-08, University of Washington*, 1991.

[25] Jonathan Jacky. Formal specification for a clinical cyclotron control system. *SIGSOFT Softw. Eng. Notes*, 15(4):45–54, April 1990.

[26] Eunkyoung Jee, Insup Lee, and Oleg Sokolsky. Assurance cases in model-driven development of the pacemaker software. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 343–356. Springer, 2010.

[27] R Jetley and P Jones. Safety requirements based analysis of infusion pump software. *IEEE RTSS/SMDS*, 2007.

[28] Raoul Jetley, S Purushothaman Iyer, and P Jones. A formal methods approach to medical device review. *Computer*, 39(4):61–67, 2006.

[29] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selective verification. In *INFOCOM*, pages 529–537. IEEE, 2008.

[30] Cheolgi Kim, Mu Sun, Sibin Mohan, Heechul Yun, Lui Sha, and Tarek F Abdelzaher. A framework for the safe interoperability of medical devices in the presence of network failures. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 149–158. ACM, 2010.

[31] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 1993.

[32] Tom Maibaum and Alan Wassyng. A product-focused approach to software certification. *Computer*, 41(2):91–93, 2008.

[33] J. Meseguer. Taming distributed system complexity through formal patterns. *Science of Computer Programming*, 2013.

[34] José Meseguer. Membership algebra as a logical framework for equational specification. In *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 18–61, London, UK, 1997. Springer-Verlag.

[35] José Meseguer and Peter C. Ölveczky. Formalization and correctness of the PALS architectural pattern for real-time systems. In *12th International Conference on Formal Engineering Methods (ICFEM 2010)*, volume 6447, pages 303–320. Springer LNCS, 2010.

[36] José Meseguer, Miguel Palominob, and Narciso Martí-Oliet. Algebraic simulations. *The Journal of Logic and Algebraic Programming*, 79(2):103–143, February 2010.

[37] José Meseguer and Carolyn L. Talcott. Semantic models for distributed object reflection. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 1–36, London, UK, UK, 2002. Springer-Verlag.

[38] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[39] Tommi Mikkonen. Formalizing design patterns. In *ICSE*, pages 115–124, 1998.

[40] S.P. Miller, D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Proc. 28th Digital Avionics Systems Conference*. IEEE, 2009.

[41] J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.

[42] P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, 2007.

[43] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[44] Peter Ölveczky and José Meseguer. Completeness of Real-Time Maude Analysis (Extended Version). *Technical Report*, 2006.

[45] Arnab Ray and Rance Cleaveland. Unit verification: the cara experience. *International Journal on Software Tools for Technology Transfer*, 2004.

[46] ML Rogers, RW Nickalls, ET Brackenbury, FD Salama, MG Beattie, and AG Perks. Airway fire during tracheostomy: prevention strategies for surgeons and anaesthetists. *Annals of the Royal College of Surgeons of England*, 83(6):376, 2001.

[47] Motoshi Saeki. Behavioral specification of GOF design patterns with LOTOS. In *APSEC*, pages 408–415. IEEE Computer Society, 2000.

[48] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[49] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *ICSE*, pages 666–675. IEEE Computer Society, 2004.

[50] DJ Stouffer. Fires during surgery: Two fatal incidents in los angeles. *Journal of Burn Care & Research*, 13(1):114–117, 1992.

[51] Toufik Taibi and David Ngo Chek Ling. Formal specification of design patterns - a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

[52] M. Viswanathan and R. Viswanathan. Foundations for circular compositional reasoning. In *Proc. ICALP'01*. Springer LNCS, 2001.

[53] Yi Zhang, Paul L Jones, and Raoul Jetley. A hazard analysis for a generic insulin infusion pump. *Journal of diabetes science and technology*, 4(2):263, 2010.