A Constraint-Oriented Specification of a Steam-Boiler Controller in Higher Order Logic

Jim Alves-Foss
Laboratory for Applied Logic
Department of Computer Science
University of Idaho
Moscow Idaho, USA

Abstract

The increased complexity of control systems, user's desire for dynamic system reconfiguration and customization has brought about a need for computer controlled systems. To ensure the correctness and safety of those systems, it is essential for system developers to utilize formal approaches to specifying and validating those systems. This paper uses the steam-boiler specification problem to illustrate an approach to specifying and verifying the correct operation of a system control software using the HOL theorem proving system.

1 Introduction

> *Even within the field, formal methods are not viewed as a panacea but as a step that must be taken to put software engineering on the same solid footing as any other engineering field. While the specific techniques to choose and use will certainly vary, the key ideas are now in place...* – Gerhart, 1990[1].

The increased complexity of control systems, user's desire for dynamic system reconfiguration and customization has brought about a need for computer controlled systems (hybrid systems). To ensure the correctness and safety of those systems, it is essential for system developers to utilize methods that provide a high level of assurance. Much of the specification and development of current software systems occurs in a non-formal or semi-formal environment. Although testing and the use of standard development techniques have helped eliminate errors that may have otherwise been introduced into the system, several other

[1]Gerhart, S., "Applications of Formal Methods: Developing Virtuoso Software", IEEE Software, vol.7(5), (September 1990), pp. 6-10.

errors still get through. In light of this, we strongly support Gerhart's sentiments advocating the use of formal methods as a necessary tool for the development of reliable software systems.

Mathematical analysis of computer software and hardware is not an easy task. Every formal analysis done to date has been a virtuoso performance, carried out by experts in logic, specification, and mechanical reasoning. Research focusing on the techniques for simplification of formal analysis and dissemination of information for the application of formal methods techniques is essential.

The benefits of formal specifications have been discussed widely [BH95, Ger90, Hal90], but have been limited by the availability of automated tools to assist in the development, analysis and verification of formal specifications. Even as more of these tools are becoming available [Cra91] users are still finding it difficult to readily use these tools to gain the most benefits from formal methods. Part of this problem lies in lack of experience, large depth of knowledge and a set of working examples in the field.

This goal of this paper is to disseminate information about the formal design, specification and verification of hybrid control systems. Specifically we discuss the techniques for software specification and verification that we use in our work at the Laboratory for Applied Logic (LAL) at the University of Idaho, as we apply it to the steam-boiler controller example [Abr94, ABL95]. It is our hope that readers of this paper will be able to learn from this example and utilize similar approaches in their work.

2 HOL

To formally model the reliability properties of a hybrid system and to ensure the accuracy of our specification, we felt that it was necessary to develop the specifications and proofs of those specifications using a mechanical verification system. This prevents proofs from containing logical mistakes, and assures that the foundations on which the work is based are sound. Due to the nature of the proofs, which include quantification over sets of objects, we felt that a system which supports higher-order logic and a typed lambda calculus would facilitate our efforts. The HOL system was selected for this project due to its support for higher-order logic, generic specifications and polymorphic type constructs. Furthermore its availability, ruggedness, local support, and a growing world-wide user base made it a very attractive selection[2]. In this section we provide a brief description of HOL and its terminology

HOL is a general theorem proving system developed at the University of Cambridge [Gor87, CGM87] that is based on Church's theory of simple types, or higher-order logic [Chu40]. Although Church developed higher-order logic as a foundation for mathematics, it can be used for reasoning about computational systems of all kinds. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions thus permitting more general systems to be described.

HOL is not a fully automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

[2]The reader should note that the techniques discussed in this paper are not exclusive to the HOL system, but can be applied to a wide range of formal specification techniques.

Operator	*Application*	*Meaning*
=	`t1 = t2`	`t1` equals `t2`
,	`t1,t2`	the pair `t1` and `t2`
$\wedge$	`t1` $\wedge$ `t2`	`t1` and `t2`
$\vee$	`t1` $\vee$ `t2`	`t1` or `t2`
$\Longrightarrow$	`t1` $\Longrightarrow$ `t2`	`t1` implies `t2`
$\rightarrow$	`t1` $\rightarrow$ `t2` $\mid$ `t3`	if `t1` then return `t2` else `t3`

Table 1: HOL infix operators

1. Several built–in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher-order logic to derive a large number of theorems that follow from them.

2. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

3. A large collection of tactics. Examples of tactics include `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition, `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms, and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

4. A proof management system that keeps track of the state of an interactive proof session.

5. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theorems for later use. The metalanguage makes the verification system extremely flexible.

In the HOL system there are several predefined constants which can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written `"rand1 op rand2"` instead of in the usual prefix form: `"op rand1 rand2"`, which increases readability and understanding by customers and system designers. Table 1 depicts several of HOL's built–in infix operators.

Constants can also belong another special class called binders. A familiar example of a binder is $\forall$. If `c` is a binder, then the term `"c x.t"` (where `x` is a variable) is written as shorthand for the term `"c(`λ `x. t)"`. Table 2 shows several of HOL's built–in binders.

In addition to the infix constants and binders, HOL has a conditional statement that is written `a` $\rightarrow$ `b` $\mid$ `c`, meaning "if `a`, then `b`, else `c`."

Binder	*Application*	*Meaning*
$\forall$	`∀ x. t`	for all `x`, `t`
$\exists$	`∃ x. t`	there exists an `x` such that `t`
ε	`ε x. t`	choose an `x` such that `t` is true

Table 2: HOL binders

3 The Specification Process

In our work at the LAL we utilize a five step process for developing formal system requirement specifications. It is this five step process that we follow in this paper. The process enables us to divide the task of capturing and formalizing the user requirements into a set of specific tasks, some of which are independent and some of which are interrelated. It is important to realize that these steps are not necessarily completed sequentially, but often concurrently as part of an iterative (or cyclic) process where issues raised in one step need to be resolved in another step (possibly during a subsequent iteration).

1. *Get Requirements.* This task involves communicating with the user to isolate specific requirements. This can be a simple as a project outline or as complex as a full IEEE or ISO Software Requirements Specification Document. In any case, it is the information obtained in this task that drives the others. It is best if there exist traceability documentation relating issues in documents generated in this task to documents developed in the other tasks.

2. *Isolate Major Components.* Often a system will consist of a collection of major components (e.g., file server, network, hosts, database programs, controllers, sensors, etc.). The system being specified is defined as a composition of these components, whether completely hardware or software, or some hybrid.

3. *Define Sorts.* A *sort* is an abstract data type specifying an information object in the system. Communication between components, and internal information stored in the components (i.e., the component state) is specified with respect to these sorts. Sorts need to be defined for each type of communication entity exchanged between components and for each field of the system state. For each sort we need to define:

   (a) *Name.* The sort must have an understandable name that can be used later to identify elements of that sort (e.g., bool, nat, int).

   (b) *Inheritance.* Sorts may be created using other sorts either by extending their definition or by creating composite or results using them. (e.g., list of num).

   (c) *Operations.* For each sort there needs to be a collection of operations, defined in terms of a name and a signature (the sorts of the parameters and result). It is through these operations that the sort can be accessed and modified. These operations include constructors for creating elements of the sort (i.e., defining their syntactic representation), modifiers for changing elements of the sort, and selectors for returning values associated with the sort (e.g., predicates, counters, etc.). During the development of the system specification the sorts of results and parameters of these operations will be defined. This may result in requiring the specification new sorts not previously determined (e.g., an operation that returns a list of values will need a sort for that list).

(d) *Semantics.* For each operation (except for the constructors) we must define semantics. These semantics are defined for every element of the sort (for total functions) or some of the elements of the sort (for partial functions) for each parameter of the operation. The semantics can be written following one of two general approaches. Often a specification will consist of operations defined using a combination of these approaches.

   i. An operation can be specifically defined in terms of an implementation based on the constructors of the sort and other operations. The operation is defined for each possible syntactic representation of the sort of parameters (e.g., add(num,num) is defined in terms of 0 and `SUC(n)`).

   ii. An operation can be defined in terms of the behavior of its results. This usually occurs as a list of constraints for the operation where we define mandatory, permissible and/or forbidden constraints on the results of the operation.

4. *Define Component Behavior.* For each component of the system we need to specify the behavior of the component. This may occur in several phases. The first phase may specify generic behavior of the component. Additional phases will place constraints on the generic component. These constraints can be derived from the user requirements. We write component specifications in one of two manners:

   (a) *Black Box*: In the black box model we only specify the externally visible behavior of the system, usually in terms of a set of legal traces (sequences of inputs and outputs) of the system or as a set of state transitions.

   (b) *White Box*: In the white box model we not only specify externally visible behavior, but also internal behavior. This is useful for defining implementations and composite system specifications.

   Note that any one component specified above may be subsequently decomposed into subcomponents throughout the design phase using the above steps. During this decomposition it is important that the externally visible behavior of the system still satisfy the user requirements and constraints. However, we are permitted to start introducing implementation specific choices through these iterations. In the following section we will apply these steps to the steam boiler specification problem to provide examples of how this specification process is carried out. The specifications are written and evaluated in HOL.

5. *Composition.* In the final stage we define the valid composition of the components. We hook up input and output ports or traces, specifying valid interleaving between the components. At this point any internal connections/transitions can be hidden from the system's environment. We then prove that the composition satisfies user requirements (or correctly implements a high-level specification).

4 The Specification of the Steam Boiler

In this section we illustrate our approach to specifying the steam-boiler system using our methodology and HOL.

4.1 Requirements

There are several requirements that can be determined from the informal specification of the system [Abr94]. At the same time there are several requirements that are missing, are incomplete or are omitted and to be left as a design decision. The following is a list of all the requirements we have obtained from the informal specification. Due to space constraints we have left out some of the specific details of the controller, as they are available in the original informal specification.

Req-1. *Controller Behavior At Initialization.* To commence operation, the controller must establish itself with the physical units and initialize the water level in the boiler. Unfortunately the informal specification is very vague on several details for initialization, resulting in a limited set of requirements here. For example, it us unclear whether the controller will receive status messages from physical units until it has received the `PhysicalUnitsReady` message (except for the water level meter), yet the informal specification states that the initializing controller should detect failures in the physical units. Based on our understanding of the informal specification, we present the following requirements:

- Req-1.1. *Boiler Initialization.* The controller will not begin operation until it receives a `SteamBoilerWaiting` message from the physical units.
- Req-1.2. *Error Detection.* If the steam meter indicates steam leaving the boiler, or a transmission failure occurs, the controller must enter emergency stop mode.
- Req-1.3. *Water Level Initialization.* The initializing controller places an acceptable amount of water in the boiler using the `Valve` and pump control commands; utilizing the water meter to determine the current water level.
- Req-1.4. *Finish Initialization.* When water level is acceptable the controller should send a `ProgramReady` message until it receives the `PhysicalUnitsReady` message, at which point the controller enters normal mode (or degraded mode if any of the physical units are defective).

Req-2. *Controller Behavior in Operation.* At every stage the controller receives messages from the physical units, processes those messages and sends out new commands to the units.

- Req-2.1. *Processing Inputs.* Upon receipt of the messages from the physical units, the controller must process those messages to determine the status of the system and units. This status consists of values reported by operational units, calculation of values of failed units from information of operational units, system parameters, past information, and detection of the failure of units.

Req-2.1.1. *Detecting Transmission Errors.* In the problem description it is stated that the presence of transmission errors will result in a transition of the controller to emergency stop mode. A transmission failure is defined as the absence of a critical message or the presence of an unexpected message. Based on this information we define a transmission failure as i) the presence of a message from a non-existing physical unit ii) receipt of a corrupted message iii) presence of an initialization-related message iv) duplicate messages v) missing status report from any unit vi) conflicting status report from a physical unit or vii) an unexpected failure-related messages.

Req-2.1.2. *Determining Component Status.* Once all inputs are correctly received, the controller must process those messages to update the system state and determine the appropriate response. This requirement involves several sections:

Req-2.1.2.1. The collecting of reported information from each of the components.

Req-2.1.2.2. The calculation of expected values for each component based on previous data values and status, and system operational parameters.

Req-2.1.2.3. The comparison of the calculated values to the reported information.

- If reported values fall out of expected range for previously operational components, that component is now considered failed.
- If the reported status of a pump indicates a spontaneous change from the previous state, that pump is considered failed.
- If the reported status from a pump controller is inconsistent with that of the corresponding pump, which is not considered failed, then the pump controller is considered failed.

Req-2.1.2.4. The processing of failure handling messages. This includes failure detection acknowledgements and repair notifications.

Req-2.1.3. *Determine current mode.* Upon receipt and processing of messages from the physical unit, the controller can determine the correct operating mode. The mode will be one of four possible values:

Req-2.1.3.1. *Normal.* The controller is in normal mode if no failures are detected and no events have occurred to force the controller into emergency stop mode.

Req-2.1.3.2. *Degraded.* The controller is in degraded mode if there is a failure of physical units other than the water level meter and no events have occurred to force the controller into emergency stop mode.

Req-2.1.3.3. *Rescue.* The controller is in rescue mode if there is a failure of the water level meter and no events have occurred to force the controller into emergency stop mode.

Req-2.1.3.4. *Emergency Stop Mode.* The controller goes to emergency stop mode if i) there is a transmission failure ii) both the water level meter and some other physical unit have failed iii) the water level of the boiler risks reaching dangerous limits or iv) three successive Stop messages have been received.

Req-2.2. *Send Commands.* After processing the inputs and determining the proper system values, the controller is responsible for sending commands to the physical units. These commands must satisfy several constraints.

Req-2.2.1. *Mode.* The controller must notify the physical units of its current operating mode at every stage. We assume that in emergency stop mode that this is the only message the controller sends.

Req-2.2.2. *Failure Handling Messages.* The controller must send a failure detection notice to every unit that failed during that stage. It must also send a repair acknowledgement message for every received unit repair message.

Req-2.2.3. *Pump Controls.* The controller must send messages to open and close pumps to attempt to maintain the water level in the boiler within normal operating parameters. During rescue or degraded modes, these commands will be sent based on the calculation of minimum and maximum behavior of the system.

Req-2.2.4. *Implicit Constraints.* We have determined that there are a number of implicit constraints that must be satisfied by the controller when sending messages. These constraints do not appear in the original informal specification, but are required for correct operation of the system. Detecting constraints such as these is one of the difficult tasks in system specification and validation.

- No duplicate messages should be sent.
- No messages should be sent to non-existing devices (e.g., a message to pump 17 when there are only 4 pumps in the system).
- No conflicting messages should be sent (e.g., two different mode messages or OpenPump(3) and ClosePump(3)).
- No failure handling messages (or acknowledgements) should be sent unless the process determines that they should occur.
- No undefined messages should be sent.

4.2 The Components

There are several components in this system. However, in this paper we are limiting ourselves to specifying only the controller portion of the system, thus we treat the remainder of the system as the environment in which the controller is embedded. The controller consists of a device that receives inputs from the physical units and sends outputs to the physical units. These I/O events occur every five seconds and the time between inputs and outputs is minimal.

We have decided to model the controller as a simple device using state-based semantics. The controller is then considered a device that given a set of inputs and a previous state will generate a new state and a set of output commands. The predicate depicted in Figure 1 is illustrative of the form used throughout this paper to specify valid behavior[3]. A controller

[3]In practice we initially define a simple controller that avoids many of the requirements, such as initialization and failure control. In later iterations we add those additional requirements. Due to space constraints, we only discuss the final version of the specification here

```
Controller
  ⊢def ∀OldState Inputs NewState Outputs.
     Controller OldState Inputs NewState Outputs =
     (SystemMode OldState = Initialization) →
       (InitializingController System OldState Inputs NewState Outputs) |
       (OperatingController System OldState Inputs NewState Outputs)
```

Figure 1: Controller definition.

that satisfies the customer requirements given a previous system state and inputs will generate a new state and outputs such that the predicate `Controller` is true. The reader can read such a specification as:

> *A valid controller is a device which, for a specific set of system configuration parameters, when given an old state and a list of inputs will generate a new state and a list of outputs commands such that the predicate is true. This predicate is true if the controller satisfied the requirements of an initializing controller when in initialization mode and of an operating controller when not in initialization mode.*

4.3 The Sorts

When evaluating the informal specification we noticed several data types explicitly mentioned for this system. In addition, we determined that there were several data types implicit in the specification when viewed from our state-based model of the system. The HOL system, based as it is on higher order logic, provides a powerful type system that enables us to easily specify abstract data types. The user community has also provided a large collection of predefined data types and theorems related to those data types. Specifically there are types for booleans, natural numbers, reals, and lists. We will use these where necessary in the specification. However, we still need to create some additional sorts that are specific for this system; these sorts are discussed below.

4.3.1 The Steam Boiler

The first thing we noticed when reading the informal specification was the presence of system parameters for the boiler. The natural thing to do was to create a single data type encapsulating these parameters for use in the specification (Figure 2). In addition to the static parameters mentioned in the informal specification, we have included parameters indicating the number of pumps operating in the system as well as the time interval between message exchanges.

In the Boiler specification we an operation, `create_record`, from the record data type package developed by Phil Windley (one of the benefits of the large HOL user group is access to such system enhancement packages). This operation allows us to create a record data type consisting of a collection of fields with associated data types. It automatically generates

```
create_record `Boiler`
[`C`,  ":num";          % Maximum water capacity (x-liters)%
 `M1`, ":num";          % Minimum safe water level (x-liters)%
 `M2`, ":num";          % Maximum safe water level (x-liters)%
 `N1`, ":num";          % Minimum normal water level (x-liters)%
 `N2`, ":num";          % Maximum normal water level (x-liters)%
 `W`,  ":num";          % Maximum steam outcome rate (x-liters/sec)%
 `U1`, ":num";          % Maximum increase in steam outcome rate (x-liters/sec/sec)%
 `U2`, ":num";          % Maximum decrease in steam outcome rate (x-liters/sec/sec)%
 `P`,  ":num";          % Water flow from one pump (x-liters/sec)%
 `Num_Pumps`, ":num";   % Number of pumps%
 `DeltaT`, ":num"];;    % Time Interval between stages (sec)%
```

Figure 2: Boiler data type.

selector and constructor functions for the data type to provide us with the ability to access and manipulate fields of the record.

Note that we use the `num` data type throughout the record; this is a representation of natural numbers in the HOL system. We use natural numbers instead of real numbers to simplify the specification and to permit us to use the large number of tools available for reasoning about natural numbers in the HOL system. We assume that all values associated with the physical units will have some level of precision associated with them (e.g., milliliters). The notation "x-liters" is used to denote this level of precision and indicate that all units in the specification are reported at this level. Conversion between this notation and a real number representation may be necessary for these static parameters and values returned from the water level and steam rate meters. However, since we are dealing with a high level of abstraction, we will leave this conversion as an implementation detail.

4.3.2 Component Modes

The next thing we noticed in this specification was the explicit requirement for several modes of operation for the controller as well as the implicit discussion of modes of operation for the physical units. The current mode of the controller is communicated to the physical units at every time interval, while the modes of the physical units are calculated by the controller and maintained in the controller state information. We thus define the sorts `ControllerMode` and `PhysicalUnitMode` as enumerated types that defines these modes (Figure 3).

As standard practice, we have included an additional entry in each type for a special undefined data value. This undefined value permits us to explicitly specify when a value is unknown and must be initialized, or when a function can not return a valid result[4]. To define the full enumerated data type we once again use an operation from the record package, `create_enum`.

As well as creating the enumerated data type, this operation automatically proves uniqueness and induction theorems for this data type (where the induction is based on structural induction, which in this instance is a simple case-analysis). The listing of these

[4]In this specification we only use these values for the physical units in initialization mode.

```
create_enum `ControllerMode`
   `Initialization | Normal | Degraded | Rescue | EmergencyStop | undefMode`;;

create_enum `PhysicalUnitMode`
   `UnitInitialization | UnitNormal | FailureDetected | FailureReported |
    FailureAcknowledged | UnitUndefMode`;;
```

Figure 3: Unit mode data types.

theorems is rather long, but a partial listing is provided in Figure 4. Here we show the induction theorem, cases theorem and a partial listing of the theorem stating that each element of the enumerated data type is distinct for the controller mode data type. Similar theorems exist for the physical unit mode data type as well as the record data types generated by `create_record`.

4.3.3 Messages

Also explicitly mentioned in the specification is a collection of messages sent and received (Figures 5- 7). Again we have included an undefined value along with the specified values; for messages, this enables us to model corrupted messages from the network as well as invalid results from functions. In this paper, we only use it to detect corrupted messages for transmission failure and forbid its use for messages sent. As an example, we provide a partial listing of two theorems which can be automatically proven for these data types. (Figure 6). The first, `MSGSSENT_CASES_LEMMA`, states that a message sent is precisely one of the listed values, and the `MessagesSent_Distinct` theorem state that these messages are unique (we only provide a partial listing of this very long theorem). Other theorems related to properties such as isomorphisms, uniqueness and induction can also be automatically proven in the HOL system.

4.3.4 State

For our state-based model of the system we also need to specify a data type for current state of the system. We do this by creating a collection of record data types that specify the current state of each type of physical unit as well as the full controller (Figure 8). For each physical unit, the state involves the current reported values of that component as well as the operational mode determined by the controller. The pump state includes information about whether and open or close command is given during the current stage. The water level and steam rate meter states include information about minimum and maximum values derived from the reported values and calculated values based on the current mode of those components (i.e., if the component is functional we use the reported values, if it has failed, we use calculated values). The full controller state includes state information for each component (we use a list of records for the pumps and pump controllers), a current controller operating mode and a counter indicating the number of success `STOP` messages currently received from the operator.

```
ControllerMode_induct
  ⊢ ∀P'.
       P' Initialization ∧
       P' Normal ∧
       P' Degraded ∧
       P' Rescue ∧
       P' EmergencyStop ∧
       P' undefMode ⟹
       (∀C'. P' C')

ControllerMode_cases
  ⊢ ∀C'.
       (C' = Initialization) ∨
       (C' = Normal) ∨
       (C' = Degraded) ∨
       (C' = Rescue) ∨
       (C' = EmergencyStop) ∨
       (C' = undefMode)

ControllerMode_distinct
  ⊢ ¬(Initialization = Normal) ∧
     ¬(Initialization = Degraded) ∧
     ¬(Initialization = Rescue) ∧
     ¬(Initialization = EmergencyStop) ∧
     ¬(Initialization = undefMode) ∧
     ¬(Normal = Degraded) ∧
     ¬(Normal = Rescue) ∧
     ¬(Normal = EmergencyStop) ∧
     ¬(Normal = undefMode) ∧
     ¬(Degraded = Rescue) ∧
     ...
```

Figure 4: Partial listing of theorems automatically generated for enumerated data types.

```
let MessagesSent = define_type `MessagesSent`
  `MessagesSent =
     Mode ControllerMode |
     ProgramReady |
     Valve |
     LevelFailureDetection |
     SteamFailureDetection |
     LevelRepairedAck |
     SteamRepairedAck |
     OpenPump num |
     ClosePump num |
     PumpFailureDetection num |
     PumpControlFailureDetection num |
     PumpRepairedAck num |
     PumpControlRepairedAck num |
     undefMessageSent `;;
```

Figure 5: Messages sent data type.

4.4 The Behavior

Defining the behavior of a system is usually the most challenging part of a project. In this steam-boiler project there are a wide range of user requirements, both explicit and implicit, that need to be addressed. Since we are dealing with a high-level specification of the system, all definitions use a constraint-oriented black-box approach to system specification.

Instead of specifying an explicit implementation (which is tempting, but puts too many constraints on the actual system developer), we only specify externally visible behavior of the system. This behavior is defined in terms of mandatory, permissible and forbidden transitions and outputs.

It is important to understand that using this approach, we still must be careful to not force the designer into a particular implementation just because we made an arbitrary choice in the specification. For example, we have chosen to specify the messages sent between the controller and the physical units during a single stage as a list of messages. Not only do we not expect the designer to send the messages as a list, we have to be careful that we do not specify a specific ordering on messages in that list and thus require a specific order of processing based on an arbitrary decision. To avoid this problem, we have used the predicate `InList` to specify that a message is in a list, but not the location in that list[5]. For a further example, see the specific implementation discussed in Section 5.

In this section we discuss, in detail, the system specification we have derived for the steam boiler controller. We have organized this section so that it parallels the order of the user requirements presented in Section 4.1. We only present the final specification of the operating controller, leaving the specification of the initializing controller and simpler initial versions of the specification as an exercise to the reader.

[5]We could have avoided this problem by using sets of messages, but we wanted to be able to model duplicate messages – specifically so we could forbid them.

```
MSGSENT_CASES_LEMMA
  ⊢ ∀M.
      (∃C'. M = Mode C') ∨
      (M = ProgramReady) ∨
      (M = Valve) ∨
      (M = LevelFailureDetection) ∨
      (M = SteamFailureDetection) ∨
      (M = LevelRepairedAck) ∨
      (M = SteamRepairedAck) ∨
      (∃n. M = OpenPump n) ∨
      (∃n. M = ClosePump n) ∨
      (∃n. M = PumpFailureDetection n) ∨
      (∃n. M = PumpControlFailureDetection n) ∨
      (∃n. M = PumpRepairedAck n) ∨
      (∃n. M = PumpControlRepairedAck n) ∨
      (M = undefMessageSent)

MessagesSent_distinct
  ⊢ (∀C'. ¬(Mode C' = ProgramReady)) ∧
     (∀C'. ¬(Mode C' = Valve)) ∧
     (∀C'. ¬(Mode C' = LevelFailureDetection)) ∧
     (∀C'. ¬(Mode C' = SteamFailureDetection)) ∧
     (∀C'. ¬(Mode C' = LevelRepairedAck)) ∧
     (∀C'. ¬(Mode C' = SteamRepairedAck)) ∧
     (∀C' n. ¬(Mode C' = OpenPump n)) ∧
     (∀C' n. ¬(Mode C' = ClosePump n)) ∧
     (∀C' n. ¬(Mode C' = PumpFailureDetection n)) ∧
     (∀C' n. ¬(Mode C' = PumpControlFailureDetection n)) ∧
     (∀C' n. ¬(Mode C' = PumpRepairedAck n)) ∧
     ...
```

Figure 6: Partial example of automatically generated theorems for messages sent data type.

```
let MessagesReceived = define_type `MessagesReceived`
  `MessagesReceived =
     StopMsg |
     SteamBoilerWaiting |
     PhysicalUnitsReady |
     LevelRepaired |
     SteamRepaired |
     LevelFailureAck |
     SteamOutcomeFailureAck |
     Level num |
     Steam num |
     PumpRepaired num |
     PumpControlRepaired num |
     PumpFailureAck num |
     PumpControlFailureAck num |
     PumpState num bool |
     PumpControlState num bool |
     undefMessageReceived`;;
```

Figure 7: Messages received data type.

4.4.1 Component Decomposition

The behavior of the controller can be easily divided into two main phases: initialization and operation. In initialization the controller is attempting to establish communication with the physical units and establish an acceptable water level for the boiler. In operation the controller is actually processing inputs from the physical units and responding with appropriate commands. It is thus natural to divide the behavioral specification of the controller into two separate specification, one for each phase (Figure 1). As mentioned earlier, we have made some basic assumptions about the behavior of the controller. A controller will behave as an initializing controller if the previous controller mode was an initializing mode. This requires that the initializing controller eventually change to normal mode. At that time, the controller should behave as an operating controller. We assume that an operating controller should never return to initialization mode.

4.4.2 Operational Phase

In the operational phase, the controller must satisfy all of the requirements specified earlier (Req-2). In Section 4.1 these requirements are grouped into general categories. We define each requirement in one or more constraints of the system as depicted in Figure 9. The constraints may specify the behavior related to a single requirement or may combine the behavior of several requirements. For some of the requirements we have created constraints with respect to a single functionality, and with others we found it much more natural to group them per device. Therefore, there is not a direct one-to-one mapping between constraint predicates and user requirements (although a more complex mapping does exist).

In the requirements of the system we noted distinct phases of the controller during a stage: input collection, input processing, and output generating phases. We define these

```
create_record `PumpRecord`
[`ReportedPumpState`, ":bool";
 `CloseCmdGiven`, ":bool";
 `OpenCmdGiven`, ":bool";
 `PumpMode`, ":PhysicalUnitMode"
];;

create_record `PumpControlRecord`
[`ReportedPumpControlState`, ":bool";
 `PumpControlMode`, ":PhysicalUnitMode"
];;

create_record `WaterLevelRecord`
[`ReportedLevel`, ":num";
 `MinLevel`, ":num";
 `MaxLevel`, ":num";
 `WaterLevelMode`, ":PhysicalUnitMode"
];;

create_record `SteamRateRecord`
[`ReportedRate`, ":num";
 `MinRate`, ":num";
 `MaxRate`, ":num";
 `SteamMode`, ":PhysicalUnitMode"
];;

create_record `SystemState`
[`Pumps`, ":(PumpRecord)list";
 `PumpControls`, ":(PumpControlRecord)list";
 `LevelMeter`, ":WaterLevelRecord";
 `SteamMeter`, ":SteamRateRecord";
 `SystemMode`, ":ControllerMode";
 `StopCount`, ":num"
];;
```

Figure 8: State information data types.

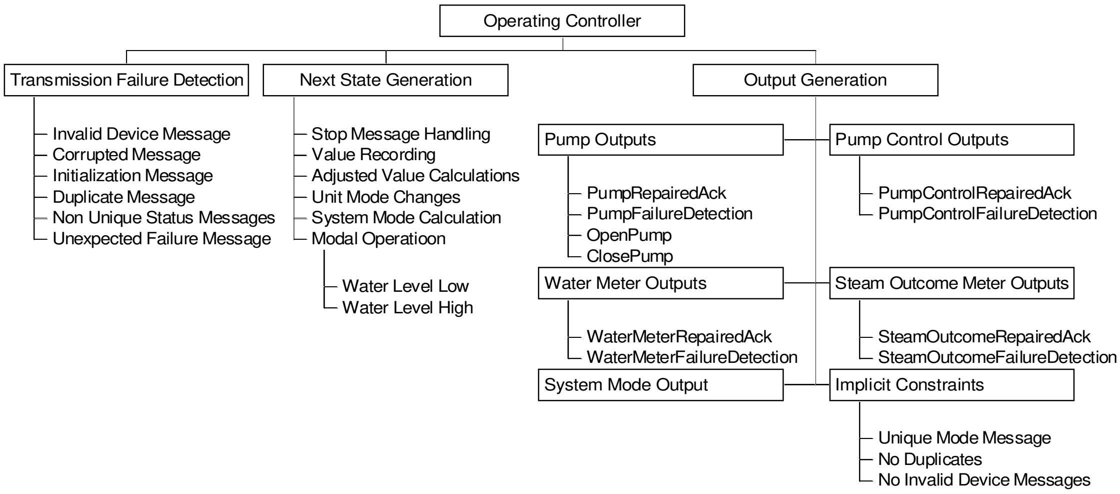

Figure 9: Constraints for Controller in Operation.

```
OperatingController
  ⊢def ∀System OldState Inputs NewState Outputs.
      OperatingController System OldState Inputs NewState Outputs =
      (DetectedTransmissionFailure System OldState Inputs →
       (Outputs = [Mode EmergencyStop]) |
       (GeneratesValidNextState System OldState Inputs NewState ∧
        ((SystemMode NewState = EmergencyStop) →
         (Outputs = [Mode EmergencyStop]) |
         GeneratesValidOutput System OldState Inputs NewState Outputs)))
```

Figure 10: Predicate defining an operating controller.

phases in terms of constraints grouped into three categories: detecting transmission failure, generating a valid next state, and generating valid outputs. For each of these phases we have defined a different predicate which is true if the system satisfies the constraints in that predicate. A valid operational controller must satisfy these predicates (Figure 10). We have made the assumption that a system in emergency stop mode need only transmit the appropriate mode message; therefore, we have explicitly indicated, upon the occurrence of a transmission error or a transition to the emergency stop mode for some other reason, that no further processing need occur (the remaining predicates do not have to be satisfied). We assume that the system must be re-initialized to recover from emergency stop.

4.4.2.1 Processing Inputs. The first phase of operation of the operating controller is the reception and processing of input messages from the physical units (Req-2.1). This processing involves data communication with the physical units. At the level of abstraction of the specification in this paper, we assume all inputs have arrived and we leave the specification of the details of this process to later. We do allow the existence of corrupted messages through improper message values or the `undefMessageReceived` data value, missing mes-

```
DetectedTransmissionFailure
 ⊢def ∀System OldState Inputs.
     DetectedTransmissionFailure System OldState Inputs =
     InvalidDeviceMessages System Inputs ∨
     undefMessageReceived InList Inputs ∨
     SteamBoilerWaiting InList Inputs ∨ PhysicalUnitsReady InList Inputs ∨
     Duplicates Inputs ∨
     ¬UniqueStatusMessages System Inputs ∨
     UnexpectedFailureHandlingMessage System OldState Inputs
```

Figure 11: Predicate defining a transmission failure.

sages, and duplicated messages. Even at our level of abstraction, we still need to determine if an error occurred in transmission (Req-2.1.1).

Detecting Transmission Errors. According to the user requirement, if a transmission failure occurs, the system must enter emergency stop mode. We further assume that the system need not satisfy any other processing constraints (except for the transmission of a Mode messages indicating the current system mode.)

A transmission failure occurs if we satisfy the predicate listed in Figure 11, which is true if any of the seven conditions specified in Req-2.1.1 are satisfied (each one is represented by a separate line in this definition, except that requirements v and vi are combined). The definitions for the predicates relating to these conditions are listed in Figures 12 & 13[6].

- An invalid device message occurs if the message came from a non-existent pump or pump controller (we number the devices consecutively starting at 0), see the HOL specification, `InvalidDeviceMessages` (Figure 12, for a definition of this property.

- An error occurs if an initialization message is received or an undefined (possibly corrupted) input is received.

- The `Duplicates` predicate is true if there exists more than one occurrence of the same message in the input list.

- A status message is considered unique if it is the only status message for that device in the list of current inputs, and there is precisely one such message for each physical unit (see `UniqueStatusMessages`). Note that this differs from duplicates in that it permits multiple status messages from the same physical device with different values being reported.

- We also assume that a transmission failure has occurred if we receive an improper failure handling message from a device (e.g., we receive a `FailureAcknowledged` message for a device which we do not consider to be failed). This constraint (Figure 13) is

[6] Note that these specifications also use the predicate `InList` extensively to avoid forcing any explicit ordering on the input messages.

```
InvalidDeviceMessages =
  ⊢_def ∀System Inputs.
      InvalidDeviceMessages System Inputs =
      (∃n.
        n ≥ (Num_Pumps System) ∧
        ((∃b. (PumpState n b) InList Inputs) ∨
         (PumpRepaired n) InList Inputs ∨
         (PumpFailureAck n) InList Inputs ∨
         (∃b. (PumpControlState n b) InList Inputs) ∨
         (PumpControlRepaired n) InList Inputs ∨
         (PumpControlFailureAck n) InList Inputs))

Duplicates
  ⊢_def ∀Messages.
      Duplicates Messages =
      (∃n m.
        ¬(n = m) ∧
        n < (LENGTH Messages) ∧
        m < (LENGTH Messages) ∧
        (EL n Messages = EL m Messages))

UniqueStatusMessages
  ⊢_def ∀System Inputs.
      UniqueStatusMessages System Inputs =
      (∀n.
        n < (Num_Pumps System) ⟹
        (∃_unique m. ∃b. PumpState n b = EL m Inputs) ∧
        (∃_unique m. ∃b. PumpControlState n b = EL m Inputs)) ∧
      (∃_unique m. ∃v. Level v = EL m Inputs) ∧
      (∃_unique m. ∃r. Steam r = EL m Inputs)
```

Figure 12: Predicates for conditions indicating transmission failure.

```
UnexpectedFailureHandlingMessage
  ⊢def ∀System OldState Inputs.
      UnexpectedFailureHandlingMessage System OldState Inputs =
      (∃n.
        (PumpFailureAck n) InList Inputs ∧
        ¬((PumpMode(EL n(Pumps OldState)) = FailureDetected) ∨
          (PumpMode(EL n(Pumps OldState)) = FailureAcknowledged))) ∨
      (∃n.
        (PumpControlFailureAck n) InList Inputs ∧
        ¬((PumpControlMode(EL n(PumpControls OldState)) =
           FailureDetected) ∨
          (PumpControlMode(EL n(PumpControls OldState)) =
           FailureAcknowledged))) ∨
      LevelFailureAck InList Inputs ∧
      ¬((WaterLevelMode(LevelMeter OldState) = FailureDetected) ∨
        (WaterLevelMode(LevelMeter OldState) = FailureAcknowledged)) ∨
      SteamOutcomeFailureAck InList Inputs ∧
      ¬((SteamMode(SteamMeter OldState) = FailureDetected) ∨
        (SteamMode(SteamMeter OldState) = FailureAcknowledged))
```

Figure 13: Additional predicate for conditions indicating transmission failure.

satisfied (and thus transmission failure detected) if the mode of the physical unit in the last state does not correspond to the failure handling message in the current list of inputs.

4.4.2.2 Generating Next State

The second set of constraints on the controller is based on the proper processing of inputs from the physical units, the evaluation of these inputs and the generation a new system state. This new state includes the actual reported values from the components, the calculated minimum and maximum values for those components, the operational mode of the unit (e.g., UnitNormal or FailureDetected) and any other state information which is useful (e.g., indications to send open and close pump messages) as specified in Figure 8. The predicate relating to this phase is specified in Figure 14.

- If we detect a transmission failure, or a third stop message the system needs to enter emergency stop mode. If we do not detect a transmission failure the system must satisfy all of the other constraints (this may be too restrictive, if we want the system to stop processing immediately upon receipt of the third stop message).

  The following constraints assume no transmission errors.

- If a stop message arrives we increment the stop message counter, otherwise we reset it to 0.

- For all input status messages, we record the reported values in the current system state as specified by `RecordsReportedValues` in Figure 15.

```
GeneratesValidNextState
  ⊢def ∀System OldState Inputs NewState.
      GeneratesValidNextState System OldState Inputs NewState =
      (DetectedTransmissionFailure System OldState Inputs ∨
       StopMsg InList Inputs ∧ (StopCount OldState = 2) ⟹
       (SystemMode NewState = EmergencyStop)) ∧
      (¬DetectedTransmissionFailure System OldState Inputs ⟹
       (StopMsg InList Inputs ⟹
        (StopCount NewState = SUC(StopCount OldState))) ∧
       (¬StopMsg InList Inputs ⟹  (StopCount NewState = 0)) ∧
       RecordsReportedValues System Inputs NewState ∧
       UpdatesAdjustedValues System OldState NewState ∧
       HandlesUnitModeChanges System OldState Inputs NewState ∧
       DetermineSystemMode System OldState NewState ∧
       ModalOperation System OldState NewState)
```

Figure 14: Generating next controller state.

```
RecordsReportedValues
  ⊢def ∀System Inputs NewState.
      RecordsReportedValues System Inputs NewState =
      (∀n b.
        (PumpState n b) InList Inputs ⟹
        (ReportedPumpState(EL n(Pumps NewState)) = b)) ∧
      (∀n b.
        (PumpControlState n b) InList Inputs ⟹
        (ReportedPumpControlState(EL n(PumpControls NewState)) = b)) ∧
      (∀v.
        (Level v) InList Inputs ⟹
        (ReportedLevel(LevelMeter NewState) = v)) ∧
      (∀r.
        (Steam r) InList Inputs ⟹
        (ReportedRate(SteamMeter NewState) = r))

UpdatesAdjustedValues
  ⊢def ∀System OldState NewState.
      UpdatesAdjustedValues System OldState NewState =
      (MinLevel(LevelMeter NewState) =
       AdjMinWaterLevel System OldState NewState) ∧
      (MaxLevel(LevelMeter NewState) =
       AdjMaxWaterLevel System OldState NewState) ∧
      (MinRate(SteamMeter NewState) =
       AdjMinSteamRate System OldState NewState) ∧
      (MaxRate(SteamMeter NewState) =
       AdjMaxSteamRate System OldState NewState)
```

Figure 15: Calculating values of physical units.

```
HandlesUnitModeChanges
  ⊢def ∀System OldState Inputs NewState.
      HandlesUnitModeChanges System OldState Inputs NewState =
      UpdatesPumpModes System OldState Inputs NewState ∧
      UpdatesPumpControlModes System OldState Inputs NewState ∧
      UpdatesLevelMode System OldState Inputs NewState ∧
      UpdatesSteamMode System OldState Inputs NewState
```

Figure 16: Physical unit mode updating.

- For the water meter and the steam meter we calculate minimum and maximum adjusted levels and record those in the state as specified by `UpdatesAdjustedValues` in Figure 15. The adjusted levels are based on reported values, component failure status and physical parameters of the system as specified in [Abr94, ABL95].

- The next state also contains the calculated failure mode of the physical units as specified by `HandlesUnitModeChanges` in Figure 16. Specifications for the proper updating of the water level meter mode and the pump modes are given in Figure 17. In these predicates we determine if the unit failed, if responses have returned from reported failures or if the unit is fine resulting in one of the modes specified in Figure 3; predicates for the other physical units are similar.

- Given the physical unit modes, old system state and inputs, we can calculate the new controller mode as specified by `DetermineSystemMode` in Figure 18. Here the system enters emergency stop mode if the water level is approaching dangerous values (see `RedLine` in Figure 18) otherwise it enters the appropriate mode based on the mode of the water level meter and the mode of all of the other physical units.

- The final constraint for next state generation involves control of the pumps as specified in Figure 19. There are a wide range of possibilities available for the specification of this portion of the controller operation, some of which are based on the particular mode of operation that the controller is in (hence the name modal operation). The basic premise here, as specified in the informal requirements (Req-2.2.3), is that the controller attempts to maintain a safe water level in the system (preferably within normal operating bounds `N1` and `N2`). This requirement could be interpreted several ways based on the current water level:

  - If the adjusted minimum water level is too low, and the adjusted maximum water level is not too high then:

    * Do not close more pumps than we open (the option we chose in the constraints given). This form of the specification permits designers to close one pump and open another if such behavior is required due to physical limitations of the system.

```
UpdatesLevelMode
  ⊢def ∀System OldState Inputs NewState.
      UpdatesLevelMode System OldState Inputs NewState =
      (let OldMode = WaterLevelMode(LevelMeter OldState) in
        let NewMode = WaterLevelMode(LevelMeter NewState) in
         (((OldMode = UnitNormal) ⟹
           (NewMode =
            ((¬(ReportedLevel(LevelMeter NewState) =
                AdjMinWaterLevel System OldState NewState)) →
             FailureDetected | UnitNormal))) ∧
          ((OldMode = FailureDetected) ⟹
           (NewMode =
            (LevelFailureAck InList Inputs →
             FailureAcknowledged | FailureDetected))) ∧
          ((OldMode = FailureAcknowledged) ⟹
           (NewMode =
            (LevelRepaired InList Inputs →
             UnitNormal | FailureAcknowledged)))))

UpdatesPumpModes
  ⊢def ∀System OldState Inputs NewState.
      UpdatesPumpModes System OldState Inputs NewState =
      (∀n.
        n < (Num_Pumps System) ⟹
        let OldPumpState = EL n(Pumps OldState) in
         let NewPumpState = EL n(Pumps NewState) in
          let OldMode = PumpMode OldPumpState in
           let NewMode = PumpMode NewPumpState in
            let OldReported = ReportedPumpState OldPumpState in
             let NewReported = ReportedPumpState NewPumpState in
              (((OldMode = UnitNormal) ⟹
                (NewMode =
                 ((CloseCmdGiven OldPumpState ∧ NewReported ∨
                   OpenCmdGiven OldPumpState ∧ ¬NewReported ∨
                   ¬(OpenCmdGiven OldPumpState ∨
                     CloseCmdGiven OldPumpState) ∧
                   ¬(NewReported = OldReported)) →
                  FailureDetected | UnitNormal))) ∧
               ((OldMode = FailureDetected) ⟹
                (NewMode =
                 ((PumpFailureAck n) InList Inputs →
                  FailureAcknowledged | FailureDetected))) ∧
               ((OldMode = FailureAcknowledged) ⟹
                (NewMode =
                 ((PumpRepaired n) InList Inputs →
                  UnitNormal | FailureAcknowledged)))))
```

Figure 17: Physical unit mode updating.

```
DetermineSystemMode
  ⊢def ∀System OldState NewState.
      DetermineSystemMode System OldState NewState =
      (SystemMode NewState =
       (((SystemMode OldState = EmergencyStop) ∨
         RedLine System OldState NewState) →
        EmergencyStop |
        let UnitsOK =
              (SteamMode(SteamMeter NewState) = UnitNormal) ∧
              (∀n.
                n < (Num_Pumps System) ⟹
                (PumpMode(EL n(Pumps NewState)) = UnitNormal)) ∧
              (∀n.
                n ≤ (Num_Pumps System) ⟹
                (PumpControlMode(EL n(PumpControls NewState)) =
                 UnitNormal))
        in
         ((WaterLevelMode(LevelMeter NewState) = UnitNormal) →
          (UnitsOK →  Normal | Degraded) |
          (UnitsOK →  Rescue | EmergencyStop))))

RedLine
  ⊢def ∀System OldState NewState.
      RedLine System OldState NewState =
      (AdjMinWaterLevel System OldState NewState) < (M1 System) ∨
      (AdjMaxWaterLevel System OldState NewState) > (M2 System)
```

Figure 18: Determining controller operating mode.

```
ModalOperation
  ⊢def ∀System OldState NewState.
      ModalOperation System OldState NewState =
      ((AdjMinWaterLevel System OldState NewState) < (N1 System) ∧
       (AdjMaxWaterLevel System OldState NewState) < (N2 System) ⟹
       (NumPumpCloseCommands(Pumps NewState)) ≤
       (NumPumpOpenCommands(Pumps NewState))) ∧
      ((AdjMinWaterLevel System OldState NewState) > (N1 System) ∧
       (AdjMaxWaterLevel System OldState NewState) > (N2 System) ⟹
       (NumPumpOpenCommands(Pumps NewState)) ≤
       (NumPumpCloseCommands(Pumps NewState)))
```

Figure 19: Pump control operation.

    * Open all of the pumps (the option we choose in the implementation specified in Section 5). This is drastic and may lead to oscillation problems in some systems, but is easy to specify and implement.
    * Manage pumps such that the expected water level will not decrease too much. This is the best solution in that it allows designers to take into consideration the current steam output and other operational parameters of the system to optimize a solution. Unfortunately, without more specific knowledge of the dynamics of the system this is very hard to specify. (The reader is encouraged to try to find a method for specifying this).
  - If the adjusted minimum water level is not too low, and the adjusted maximum water level is not too high then:
    * We are fine so leave it up to the designer (the option we took in the constraints given by not specifying this case).
    * Don't change any of the pumps (the option we chose in the implementation presented in Section 5 as a design choice).
  - If the adjusted minimum water level is not too low, and the adjusted maximum water level is too high we should close pumps instead of open pumps similar to the case when water was too low.
  - If the adjusted minimum water level is too low, and the adjusted maximum water level is too high then:
    * The choice we make is arbitrary so we leave it up to the designer (the option we took in the constraints given by not specifying this case).
    * Don't change any of the pumps (the option we chose in the implementation presented in Section 5 as a design choice).

4.4.2.3 Generating Valid Outputs. The third set of constraints on the controller is based on the outputs sent by the controller in its final phase of operation for the stage. These outputs must satisfy the explicit user requirements by sending the mode message, appropriate open and close messages, and failure handling messages, while satisfying all the implicit constraints (Req-2.2.4). The predicate `GeneratesValidOutput` in Figure 20 specifies this requirement, with supporting predicates given in Figure 21.

- Valid output for a physical unit involves sending acknowledgement messages for device repair messages, sending failure detection messages, and for the pumps sending pump open and pump close messages. Each of these messages is specified in the appropriate predicate in Figure 21 for the corresponding physical unit.

- The requirements also state that a controller mode message must be sent corresponding to the current system mode.

- Implicit requirements include the constraints that only one mode message be sent, no duplicate messages are sent, and that no messages are sent to invalid devices. The only

```
GeneratesValidOutput
  ⊢def ∀System Inputs NewState Outputs.
      GeneratesValidOutput System Inputs NewState Outputs =
      ValidPumpOutput System Inputs NewState Outputs ∧
      ValidPumpControlOutput System Inputs NewState Outputs ∧
      ValidWaterMeterOutput System Inputs NewState Outputs ∧
      ValidSteamMeterOutput System Inputs NewState Outputs ∧
      (Mode(SystemMode NewState)) InList Outputs ∧
      (∃unique i. i < (LENGTH Outputs) ∧ (∃x. EL i Outputs = Mode x)) ∧
      ¬Duplicates Outputs ∧
      (∀n.
        (OpenPump n) InList Outputs ∨
        (ClosePump n) InList Outputs ∨
        (PumpFailureDetection n) InList Outputs ∨
        (PumpRepairedAck n) InList Outputs ∨
        (PumpControlFailureDetection n) InList Outputs ∨
        (PumpControlRepairedAck n) InList Outputs ⟹
        n < (Num_Pumps System))
```

Figure 20: Predicate specifying constraints on output generation.

conflicting messages that could be sent are open and close messages to the same pump, which is forbidden in `ValidPumpOutput`. We assume that no corrupted messages will be sent and that failure handling messages are sent only as specified in the valid output predicates (we could change the implication operator in the predicates to a two-way implication to specify this requirement, but rather have left this for a further iteration of the specification).

4.5 The Composite System, Verification and Validation

At this level of abstraction, there is no true composite system but rather a set of constraints. The operating controller was specified in Figure 10 and the full controller specified in Figure 1. A more complex system (or lower level specification) would involve additional constraints on the interaction of the inputs, states and outputs of the components of the system, possibly including an abstraction mapping to a higher level, more abstract, specification.

Figure 22 is an example theorem that can be proven given the specification of the operating controller (this theorem was proven in the HOL system). The theorem states that any system which satisfies the `OperatingController` predicate (while not receiving a transmission error or entering emergency stop mode) will not update the system state to indicate that both open and close commands should be sent to the same pump. As an example to the reader we have also included the HOL proof tactic for this theorem. It involves simple case analysis and rewriting of the definitions (since this constraint was implied in the `ValidPumpOutput` predicate).

There are several other theorems that may need to be proven about such a specification when determining the safety and performance of the system. Development of these theorems should be performed in conjunction with the customer and system designers and presented

```
ValidPumpOutput
  ⊢def ∀System Inputs NewState Outputs.
      ValidPumpOutput System Inputs NewState Outputs =
      (∀n.
        n < (Num_Pumps System) ⟹
        ((PumpRepaired n) InList Inputs ⟹
         (PumpRepairedAck n) InList Outputs) ∧
        ((PumpMode(EL n(Pumps NewState)) = FailureDetected) ⟹
         (PumpFailureDetection n) InList Outputs) ∧
        (OpenCmdGiven(EL n(Pumps NewState)) ⟹
         (OpenPump n) InList Outputs) ∧
        (CloseCmdGiven(EL n(Pumps NewState)) ⟹
         (ClosePump n) InList Outputs) ∧
        ¬((OpenPump n) InList Outputs ∧ (ClosePump n) InList Outputs))

ValidPumpControlOutput
  ⊢def ∀System Inputs NewState Outputs.
      ValidPumpControlOutput System Inputs NewState Outputs =
      (∀n.
        n < (Num_Pumps System) ⟹
        ((PumpControlRepaired n) InList Inputs ⟹
         (PumpControlRepairedAck n) InList Outputs) ∧
        ((PumpControlMode(EL n(PumpControls NewState)) =
          FailureDetected) ⟹
         (PumpControlFailureDetection n) InList Outputs))

ValidWaterMeterOutput
  ⊢def ∀System Inputs NewState Outputs.
      ValidWaterMeterOutput System Inputs NewState Outputs =
      (LevelRepaired InList Inputs ⟹  LevelRepairedAck InList Outputs) ∧
      ((WaterLevelMode(LevelMeter NewState) = FailureDetected) ⟹
       LevelFailureDetection InList Outputs)

ValidSteamMeterOutput
  ⊢def ∀System Inputs NewState Outputs.
      ValidSteamMeterOutput System Inputs NewState Outputs =
      (SteamRepaired InList Inputs ⟹  SteamRepairedAck InList Outputs) ∧
      ((SteamMode(SteamMeter NewState) = FailureDetected) ⟹
       SteamFailureDetection InList Outputs)
```

Figure 21: Predicates for specific valid output generation.

```
NoPumpContradiction
  ⊢ ¬DetectedTransmissionFailure System OldState Inputs ⟹
     OperatingController System OldState Inputs NewState Outputs ∧
     ¬(SystemMode NewState = EmergencyStop) ⟹
     (∀n.
       n < (Num_Pumps System) ⟹
       ¬(OpenCmdGiven(EL n(Pumps NewState)) ∧
         CloseCmdGiven(EL n(Pumps NewState))))

Proof Tactic =
  REWRITE_TAC[OperatingController; GeneratesValidOutput; ValidPumpOutput]
  THEN COND_CASES_TAC
  THEN ASM_REWRITE_TAC[]
  THEN COND_CASES_TAC
  THEN ASM_REWRITE_TAC[]
  THEN STRIP_TAC
  THEN GEN_TAC
  THEN POP_ASSUM_LIST(λths. ASSUME_TAC(SPEC "n" (el 8 ths)))
  THEN STRIP_TAC THEN RES_TAC
  THEN ASM_CASES_TAC "CloseCmdGiven (EL n (Pumps NewState))"
  THEN ASM_CASES_TAC "OpenCmdGiven (EL n (Pumps NewState))"
  THEN ASM_REWRITE_TAC[]
  THEN RES_TAC
```

Figure 22: Theorem stating that no conflicting pump commands are given.

```
ModalOperationImpl
  ⊢def ∀System OldState NewState.
      ModalOperationImpl System OldState NewState =
      ((AdjMinWaterLevel System OldState NewState) < (N1 System) ∧
       (AdjMaxWaterLevel System OldState NewState) < (N2 System) ⟹
       (∀n.
         n < (Num_Pumps System) ⟹
         let NewPumpState = EL n(Pumps NewState)
         in
          (¬CloseCmdGiven NewPumpState ∧
           (OpenCmdGiven NewPumpState =
            (PumpMode NewPumpState = UnitNormal) ∧
            ¬ReportedPumpState NewPumpState)))) ∧
      ((AdjMinWaterLevel System OldState NewState) ≤ (N1 System) ∧
       (AdjMaxWaterLevel System OldState NewState) ≥ (N2 System) ∨
       (AdjMinWaterLevel System OldState NewState) ≥ (N1 System) ∧
       (AdjMaxWaterLevel System OldState NewState) ≤ (N2 System) ⟹
       (∀n.
         n < (Num_Pumps System) ⟹
         let NewPumpState = EL n(Pumps NewState)
         in
          (¬CloseCmdGiven NewPumpState ∧ ¬OpenCmdGiven NewPumpState))) ∧
      ((AdjMinWaterLevel System OldState NewState) > (N1 System) ∧
       (AdjMaxWaterLevel System OldState NewState) > (N2 System) ⟹
       (∀n.
         n < (Num_Pumps System) ⟹
         let NewPumpState = EL n(Pumps NewState)
         in
          (¬OpenCmdGiven NewPumpState ∧
           (CloseCmdGiven NewPumpState =
            (PumpMode NewPumpState = UnitNormal) ∧
            ReportedPumpState NewPumpState))))
```

Figure 23: Operation of a component that satisfies modal operation.

as a list of additional requirements or properties that need to be satisfied by the system.

5 An Implementation

So far, all we have provided is a specification for a high-level abstraction of the system. This specification uses a constraint-oriented approach to list the requirements that must be satisfied by an implementation of the system. In this section we present an example of a partial implementation of the system and discuss proofs of correctness of the implementation.

5.1 An Operating Controller Implementation

We have decided to specify a more concrete representation of the controller in terms of the decisions it makes when controlling the pumps. Specifically, we have specified a component, `ModalOperationImpl` (Figure 23) that evaluates the current calculated water

```
ModalImplSatisfies
  ⊢ ModalOperationImpl System OldState NewState ∧
     StateListSizeObl System OldState ∧
     StateListSizeObl System NewState ⟹
     ModalOperation System OldState NewState
```

Figure 24: Implementation satisfies constraints for modal operation.

```
NoPumpContradictionImpl
  ⊢ ModalOperationImpl System OldState NewState ∧
     StateListSizeObl System OldState ∧
     StateListSizeObl System NewState ⟹
     (∀n.
       n < (Num_Pumps System) ⟹
       ¬(OpenCmdGiven(EL n(Pumps NewState)) ∧
         CloseCmdGiven(EL n(Pumps NewState))))
```

Figure 25: Implementation satisfies no pump contradiction requirement.

level and steam rates, and controls the pumps accordingly. Specifically, the controller is based on the suggested operation in the addendum to the steam-boiler problem [ABL95]. If the minimum calculated water level is too low, and the maximum calculated water level is not to high, we turn on pumps (all of them). If the maximum calculated water level is too high and the minimum calculated water level is not too low, we turn off pumps (all of them). Otherwise we neither turn on or turn off pumps.

This is a simple controller, as it treats all pumps as a single unit by opening and closing them all at the same time, yet it does satisfy the constraints given earlier as stated by the theorem in Figure 24 (also proven in HOL). The predicate `StateListSizeObl` is used to specify that in the system state there are the same number of pump and pump control records as there are pumps in the system (required for inductive proofs).

This example provides a simple example of a refinement, where we have taken a higher level abstract specification and refined it into a specific implementation. The implementation does not provide all of the flexibility of the specification, but instead makes some specific design choices that still satisfy the constraints of the full specification.

In Figure 25 we present a theorem that demonstrates that the specific modal operation implementation satisfies the requirement that conflicting pump commands not be given. It is instructive to note that the constraint satisfies by this implementation directly enforces the constraint specified in the `ValidPumpOutput` predicate, thus enabling us to not enforce the constraint there when implementing the output function in the next section. This type of behavior occurs frequently in system design and implementation where a constraint listed in one part of the specification is actually satisfied in the implementation of another part of the specification. This is often referred to as *functional rearrangement* and is fine as long as the designers and implementors are aware of it and can validate that the implementation still

```
OutputImpl
    ⊢def ∀Inputs NewState Outputs.
        OutputImpl Inputs NewState Outputs =
        (Outputs =
         ((SystemMode NewState = EmergencyStop) →
          [Mode EmergencyStop] |
          APPEND
          [Mode(SystemMode NewState)]
          (APPEND
           (FailureControlResponses Inputs)
           (APPEND
            (PumpOutputs 0(Pumps NewState))
            (APPEND
             (PumpControlOutputs 0(PumpControls NewState))
             (APPEND
              (WaterMeterOutputs(LevelMeter NewState))
              (SteamRateOutputs(SteamMeter NewState))))))))
```

Figure 26: Output generation implementation.

satisfies the specification. Such rearrangements should be well documented in the system design documentation.

5.2 Output Generation Implementation

The second example implementation we provide here is for the full output generation unit. This unit satisfies all of the constraints specified for valid output generation. Specifically the implementation iterates through the system state generating messages for each of the physical units. The order is explicit in this implementation specifying mode message first, failure control responses (repair acknowledgements) next, and then pumps (in numeric order), pump controls (in numeric order), water meter and steam meter. The predicate defining this operation is given in Figure 26. The definitions for each of the physical unit operations is given in Figure 27.

This implementation again applies functional rearrangement by pulling out acknowledgements to repair messages into a separate module. This implementation, in conjunction with the modal operation implementation (when not in emergency stop mode), satisfies the valid output generation predicate (Figure 28). We require use of the modal operation implementation to satisfy the valid pump output constraint for non-conflicting pump-control messages. The proof of this theorem requires the proof of many lemmas, induction and rewriting. Although tedious and time consuming it is not insurmountable, especially when divided into several lemmas (one for each constraint of `GeneratesValidOutput`).

It is interesting to note that performing this proof helped us fine-tune the specification by making us add an additional clause (`i<(LENGTH Outputs)`) to the mode message uniqueness constraint that was not in our original high-level specification (the modified version appears in Figure 20). This constraint is required since the `EL` operation on lists (used to specify this constraint) is only partially defined, and is not defined for empty lists

```
FailureControlResponses
  ⊢def (FailureControlResponses[] = []) ∧
     (∀x L.
       FailureControlResponses(CONS x L) =
       ((∃n. x = PumpRepaired n) →
        CONS(PumpRepairedAck(Value x))(FailureControlResponses L) |
        ((∃n. x = PumpControlRepaired n) →
         CONS
         (PumpControlRepairedAck(Value x))
         (FailureControlResponses L) |
         ((x = SteamRepaired) →
          CONS SteamRepairedAck(FailureControlResponses L) |
          ((x = LevelRepaired) →
           CONS LevelRepairedAck(FailureControlResponses L) |
           FailureControlResponses L)))))

PumpOutputs
  ⊢def (∀n. PumpOutputs n[] = []) ∧
     (∀n p L.
       PumpOutputs n(CONS p L) =
       APPEND
       ((PumpMode p = FailureDetected) →
        [PumpFailureDetection n] | [])
       (APPEND
        (OpenCmdGiven p →  [OpenPump n] | [])
        (APPEND
         (CloseCmdGiven p →  [ClosePump n] | [])
         (PumpOutputs(SUC n)L))))

PumpControlOutputs
  ⊢def (∀n. PumpControlOutputs n[] = []) ∧
     (∀n p L.
       PumpControlOutputs n(CONS p L) =
       APPEND
       ((PumpControlMode p = FailureDetected) →
        [PumpControlFailureDetection n] | [])
       (PumpControlOutputs(SUC n)L))

WaterMeterOutputs
  ⊢def ∀WRec.
      WaterMeterOutputs WRec =
      ((WaterLevelMode WRec = FailureDetected) →
       [LevelFailureDetection] | [])]

SteamRateOutputs
  ⊢def ∀WRec.
      SteamRateOutputs WRec =
      ((SteamMode WRec = FailureDetected) →
       [SteamFailureDetection] | [])
```

Figure 27: Physical unit output implementation modules.

```
OutputImplSatisfies
  ⊢ ¬(SystemMode NewState = EmergencyStop) ⟹
      (ModalOperationImpl System OldState NewState ∧
       OutputImpl Inputs NewState Outputs ∧
       StateListSizeObl System OldState ∧
       StateListSizeObl System NewState ⟹
        GeneratesValidOutput System Inputs NewState Outputs)
```

Figure 28: Output implementation validation theorem.

(which occurs when the argument specifies an entry beyond the end of the list). Fully specifying this constraint limits us to worrying about precisely those messages in the output list.

6 Conclusions

In this paper we have presented an outline of a five step process that we use to develop specifications of computer systems (software, hardware or hybrid systems). To demonstrate this process, we have applied it to the steam-boiler controller specification problem [Abr94] in the HOL theorem proving system. We have found that the approach outlined here provides the following benefits:

- It divides the specification problem into clearly definable steps.
- It allows us to clearly define a high-level abstraction of the system, in terms of data types and their semantics.
- It allows us to utilize both constraint-oriented and implementation-oriented specification approaches, depending on the desired level of abstraction and designer needs.
- Specifications are easily refinable and extensible.
- Using the HOL system we can prove properties of the specification and validate that lower-level implementations satisfy higher-level specifications.

We have also found that, when mapping user informal requirements into high-level specifications, it is much easier to start with a constraint-oriented specification that consists of a collection of predicates defined over all values of the data types. This is due to the fact that user requirements are often specified in terms of required, permissible and forbidden system behaviors (in other words constraints). These constraints can be grouped into functional untis which later can be implemented by system modules.

When refining the specifications during implementation it is sometimes necessary to employ functional rearrangement of the constraint groupings to more closely match the implementation modules (such as the rearrangement we employed in the output generation module). It is also sometimes unnecessary to explicitly implement certain constraints in an implementation module if other modules guarantee that constraint (as with the consistent pump control command requirement).

References

[ABL95] J. Abrial, E. Boerger, and H. Langmaack. Additional information concerning the physical behaviour of the steam boiler. 1995.

[Abr94] J. Abrial. Steam-boiler control specification problem. 1994.

[BH95] J.P. Bowen and M. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4), July 1995.

[CGM87] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[Cra91] D. Craigen. Tool support for formal methods. In *IEEE 13th International Conference on Software Engineering*, pages 184–185, 1991.

[Ger90] S. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, 7(5):6–10, September 1990.

[Gor87] M. Gordon. A proof generating system for higher-order logic. Technical Report 103, University of Cambridge Computer Laboratory, January 1987.

[Hal90] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.