18649 Distributed Embedded Systems
Soda Machine Portfolio Discussion

By Justin Ray, last modified 2010-02-16

Table of Contents

1. Introduction

The soda machine portfolio (SMP) is provided to show you a reasonably complete example of the portfolio that you will end up with after you have completed all the course projects. To provide additional insight about the projects and the design process, we have written this document to explain the some of the design decisions that went in to this project and provide some additional information about the design process.

We will start with some general discussion that applies to the project as a whole and then move into notes for specific projects. The project specific notes are roughly organized to start at the beginning of the semester and follow the SMP through each project, pointing out interesting details and useful tips along the way.

We suggest you read the whole discussion document and look through the entire soda machine portfolio after the UML design and End-to-End Design process lectures. Even if you don't understand every part of the design process, it will help to have a big picture overview and see how the different parts of the design process fit together. Then, as you go through the projects during the semester, you can refer back to the discussion for each project as you reach it.

Before you get started with this document, please read the Requirements I document in the SMP and in the elevator portfolio template. These documents describe the objects in the system and messages that are present on the networks. The discussion will make more sense to you if you are familiar with the parts of the system.

There are some parts of the simulation framework and the portfolio documents in the SMP that still use elevator terminology (e.g. Passenger instead of Customer). The example portfolio is a design of the soda machine system as a proxy for the elevator system, so throughout the SMP and this discussion, we will refer to similar parts of the two systems interchangeably.

2. Disclaimers

Read these disclaimers carefully before you base your design on what you see in the examples.

2.1 Not defect free (but close)

We have made every effort to make the SMP is as complete and error-free as possible. However, it's likely that there are some defects that we have not found. If you find a defect (or think you have) please notify the course staff and include as much information as you can (what document is affected, why you think it is a defect, etc.).

2.2 Not authoritative

We have also attempted to follow the guidelines and procedures set forth in the course projects. However, these change slightly from year to year (and sometimes during the semester), so there may be some discrepancies. **In the course, the project write-ups are the gold standard. If there is a conflict between the way the SMP is written and what is in the project write-up, follow the project write-up and notify the course staff.**

One specific way this manifests is that many projects require one person to create an artifact (requirements, test case, etc) and another person to perform a peer review on it. Since there was only one person involved in the generation of the SMP, the authorship and reviews were all done by the same person. But in your projects, you must have someone else do the reviews!

Another thing you will see in the example that you may not do in your project is modify the requirements or code for the system objects. The system objects used for the soda machine are less mature than those for the elevator, and several bugs found in the course of the project were actually related to the system objects. Although it is still possible that there are bugs in the elevator system objects, you may not modify these objects or their requirements unless the change is specified by a project write-up or you obtain *prior* permission from the course staff. Caveat: The entire source code is available to you, so feel free to make whatever modifications you want to the system objects when testing and debugging. Just realize that the code you submit must work correctly with the baseline simulation framework that we have provided.

2.3 Not the only way of doing things

The SMP is *not* the only way of doing things. In fact, design process is a synthesis task, and there is more than one right way to get to the goal of a reliable distributed system. That said, the projects are designed to teach you about software process and distributed design, so they take you down a fairly narrow road, and you will likely find yourself following an approach very similar to that taken in the SMP. The second half of the course offers more opportunities for innovation and open-ended design once you have become familiar with the elevator system and the design process.

2.4 May not be copied

This is an important one. You are expected to refer to the SMP to get an idea of how to approach the projects. Because the soda machine is a completely different (and much simpler) system, very little of the design content will be directly applicable to the elevator. Most of the structural content (HTML formatting, etc) is already provided in the portfolio template. **But you may not verbatim copy any part of the SMP documents or code into your projects. This includes notes, discussion, and any part of the code. Look as much as you like, but do your own work. We take cheating very seriously in this course, and copying from the SMP will be considered the same as copying from another student.** If you are unsure whether your result is too close to the SMP document, see a TA or the course instructor during office hours to double check that you are OK.

3. General Topics

This section contains discussion topics are too general to refer to a specific project or apply at a higher level to the entire design process.

3.1 Correspondence between the Elevator and the Soda Machine – Simplifying Assumptions

The purpose of the SMP is to give an example that is small enough that you can understand the whole design package while having enough complexity to make it interesting and reveal some important ideas about the design process.

In some ways, the elevator is very similar to the soda machine. For example, the VendMotorControl and the VendPositionSensors in the soda machine interact in a similar way to the DriveControl and AtFloor sensors in the elevator. But the VendMotor has much simpler dynamics (instant start and stop, constant speed) and has mechanical interlocks to protect it when it reaches the end of the rail. In the elevator, you must deal with the acceleration profile of the Drive and use the HoistwayLimit sensors to make sure you don't overrun the end of the shaft.

Even setting aside the physical differences between the elevator and the soda machine, there is one major difference that makes the soda machine much simpler than the elevator. The customer behaviors for the soda machine are much simpler than those of the elevator passengers. The customer model is defined in such a way that customer actions occur no more often than once per second, and some actions can be delayed by as much as eight seconds. This is not too unrealistic for interaction with a soda machine.

The most important implication of the simple customer behavior is that it allows us to assume that whatever action is currently in progress (vending a soda, returning coins, etc) will complete before another customer input arrives in the system.

More specifically, we can always assume that system is *quiescent* when it gets a new input. Consider what happens if we relax this assumption: What if the customer puts a coin in while a soda is being vended? There's no reason we can't handle that case, but then we need to add logic to the coin control to keep track of the fact that this coin is for the *next* vend cycle. What if the customer presses the coin return button and a soda button at the same time? What if the customer presses two different buttons? You can see how handling more complex customer behaviors would quickly add complexity to the design.

The elevator can receive inputs from multiple passengers at the same time, and the timing of these inputs is such that you cannot assume that the elevator is quiescent when they occur. You must take this complexity into account in your design. The way the projects are set up, we ease you into the complexity of the elevator by having you implement a very simple elevator for the mid-semester deadline, then adding more advanced behaviors later.

3.2 The Importance of Doing It Right and the Cost of Late-Found Bugs

The primary purpose of the course project is not just to produce a working elevator control system. Our primary goal is that you follow the design process and, by following it, come to understand how software process is important to producing software that is reliable and meets requirements. In the real embedded world, a system with amazing functionality and fancy extra features but of unknown quality isn't as valuable as a system with adequate functionality implemented in high quality software. This idea is reflected in the projects by how we assign most of the points to design process and relatively few points for design innovation.

In order to do well on the projects, you must do well following the rules and guidelines of the design process. One of the most important rules is traceability. The entire design portfolio is a living document that, at any given time, accurately reflects the current state of the design. Traceability is the glue that holds the different parts of the project together. It lets you know that you didn't miss anything, and if you find a defect in some part of the project, it lets you know which other parts of the design may be affected by that defect.

We *require* complete end-to-end traceability as a major part of the project grade. This means that what you did last week will be traced to what you are doing this week, and next week, and so forth. If you aren't thorough on your sequence diagrams, you will just have to redo them when you write requirements. If you don't redo the sequence diagrams, the incorrectness of (or lack of) traceability will make it clear that your requirements and sequence diagrams don't match up. If you aren't thorough when writing your requirements, then you will have to redo both the requirements and the sequence diagrams when you try to trace them to the statecharts. You will save yourself a lot of time and effort by doing the best, most thorough job you can on each and every project.

The further along in the process you are, the more a change in the project will cost you in terms of updating the design documents. Another way to see this is to look at the issue log for the SMP. Note that the early bugs affect just a few artifacts, but the later bugs touch almost every part of the portfolio.

3.3 The Importance of Bug Tracking – Bugs Don't Mean the Design is Bad

Bug tracking is an important part of design process. In the portfolio, bugs are tracked in the Issue Log. Some people think that if you find bugs, then that means that your system is not any good. As a blanket statement, that's just not true. The truth is that all designs have bugs. If you claim that your system is completely bug free and never had any bugs then you probably aren't looking hard enough or testing thoroughly.

The important thing about bug tracking is to understand how it improves your project. Don't get too caught up in having too many or too few bugs. Just report them in your Issue Log as honestly and completely as you can.

When you log a bug, take a minute to think about why the bug happened. Chances are, if you overlooked something in one place, you may have in others as well. Issue #10 is an example of this. The VendControl unit test revealed an implementation error, and further checking revealed that the same implementation error was present in the VendPositionControl as well.

If you see issues that affect multiple artifacts like issue #10, or you find yourself logging similar bugs over and over, then think about how you can improve your process to find these problems sooner. In the case if issue #10, you could add a step to the code review checklist to remind the reviewer to check that each message object is correctly initialized. If you do this, then bug tracking has helped you refine your process!

4. Project Specific Discussions

These discussion topics deal with aspects of a specific project or range of projects as noted in the title of each subsection

4.1 Project 2 – Sequence Diagram Numbering

The scenarios and sequence diagrams are a structured way learn about how the controllers in the distributed elevator cooperate to fulfill the high level requirements.

The project write-up requires that every arc have a unique number. This requirement is important because it lets you unambiguously refer to a specific arc in traceability, issue log entries, and other design documents.

In general, a sequential numbering is best because it is easier to understand. However, some sequence diagrams are very long (e.g. 1C or 2A). If you find an error and have to insert a new arc, it is acceptable to put a new number on the arc that is out of order with the rest of the diagram, although the arc should still be numbered so that it corresponds to the scenario numbering. For example, look at arc 4h in sequence diagram 2A. This is an arc that was added later in the design process (see Issue #6). It is not in order with the rest of the arcs, but its number begins with 4 because it corresponds to the fourth step in the scenario description.

The goal of the project is an unambiguous description of system behavior and a design that has complete and consistent traceability. The out-of-order arcs meet this goal because they are still uniquely numbered. The alternative (renumbering all the arcs and completely redoing traceability) is much more likely to introduce bugs in traceability.

4.2 Project 2 – Approaches to Defining Use Cases and Sequence Diagrams for Embedded Systems

The scenarios and sequence diagrams in the SMP are based on Customer use cases, so they deal with changes in the system that are initiated by the customers' actions. Since most people have used elevators (and soda machines) before, the easiest way to become familiar with the system is to think about how a person would interact with it. You will notice that a user action that results in a soda being vended gives rise to a series of events that encompass the entire system and most of the major system functionality. This is okay, because it gets you on the path for designing the system, but you can see that it results in the 1C and 2A sequence diagrams, which have a lot of duplication.

In an embedded system, the various controllers can also be actors (e.g. the Dispatcher on the elevator use case diagram). If this were not the case, there would be no way to describe a control loop (which isn't initiated by any user action).

If you start to view the system in terms of controller uses cases, then you get a series of simpler use cases that feed in to each other. Once consequence of this approach is that, if you write the sequence diagrams correctly, the system state at any given time corresponds to the preconditions on exactly one sequence diagram. In this view, you can think of the operation of the system as a whole as executing a series of sequence diagrams whose pre- and post-conditions dovetail into each other. Also, if you take this approach, the customer use cases are reduced to defining the change in system state that a user's action causes.

Let's consider the sequence diagram 2A as an example. We can decompose this large sequence diagram into several smaller use cases. The list below shows the use cases and gives a rough sketch of which parts of the original sequence diagram they would contain. The post-condition for each use case is the same as the precondition of the one that follows it.

- 2A - Customer presses a button when the correct amount has been paid
    - Arcs 1a through 1c – the customer's action only changes the system state
- 7A – ButtonControl indicates Choice Accepted
    - Arcs 2a and 2b and the related inputs to the ButtonControl
- 8A – VendPositionControl aligns VendCarriage
    - Arcs 3a through 3f
- 9A – Soda is Vended
    - Arcs 4a through 4g
- 7B – ButtonControl indicates Soda Empty
    - Arcs 5a and 5b and related inputs

Aside from creating much shorter sequence diagrams, there is one other benefit. Most of the other large sequence diagram (1C) will also fit into the controller use cases, so the 1C sequence diagram would be simplified to just the first couple of arcs. Not only have we greatly reduced the complexity, but we've eliminated a lot of the duplication.

If this approach works so well, why not use it in the soda machine example? The short answer is that while it may make sense to decompose the system into controller use cases, it's very difficult to write those scenarios unless you already have a very clear understanding of how the components of the system interact. Consider the example we worked out above for SD 2A. As a refinement step, to go from one giant sequence diagram to several small ones, it makes sense. But it would not be easy or straightforward if you tried to do it without the giant sequence diagram to give you the big picture. In theory, one could include a refinement step in the design process (between Projects 2 and 3) for generating controller use cases and sequence diagrams from sequence diagrams based on user use cases, but doing so would be beyond the scope of this project and the course.

If you do not understand the interactions of the controllers in the system well, writing sequence diagrams for customer/passenger use cases can help you to understand the system better. Then, as your understanding improves, you may refine your sequence diagrams along the lines of controller use cases to more succinctly describe the system operation. This refinement is useful because will likely improve your understanding of the system, but is not required for the course project.

4.3 Project 3/4 -- High-Level vs. Low-Level Requirements

Requirements in different parts of the design process serve different purposes. For example, the High Level Requirements in the Requirements I document tell you what is expected of the elevator/soda machine without telling you anything about how those goals are going to be achieved. These are the kind of requirements that might be produced by a customer in a real world project. At this level, it's very important to keep implementation details out of the requirements. The purpose of the design process is to produce a design that meets the requirements, so putting the implementation details (basically, design decisions) into the high level requirements is a reversal of the proper order of the design process. Writing proper high-level requirements is a skill that takes practice and experience to develop.

Requirements will become more low-level as the process progresses. The behavioral requirements you will write for projects 3 and 4 are extremely low level and really halfway to implementation. The form given for these requirements is extremely restrictive, but it can cover most cases.

Even for low-level requirements, there is still room to write requirements that describe a behavior without specifying implementation. One example is the ButtonControl requirement R2.4b (you can also look at the equivalent ET requirement, ER2.4b). The requirement (as of this writing) states:

R2.4. If Button[s] is equal to True AND mEmpty[s] is equal to False and mCoinCount is equal to SODA_COST and mVend is equal to False, then ButtonLight[s] shall be commanded to blink with a period of 1s.

The second half of this requirement doesn't strictly follow the formula for low-level requirements, since we don't specify a value to be set but rather an action to be taken. But this is probably the best way to write the requirement. Consider the alternative: If we want to be more specific (more formulaic), we need a notion of time and a way to keep track of the passage of time. But that requires us to make some assumptions about the way the controller is going to be implemented. Does the platform have a hardware timer? If we use a counter to track time, how do we know how often the controller will be run? Will there be a real-time clock that we can check? Any decision we make here about writing more specific requirements restricts the implementation choices we can make.

You are at a slight disadvantage in the course project because we have already provided a specific platform that you will use to implement the elevator, so it's hard to put yourself in the mindset that you don't know how things are going to turn out. But in the "real world", it is very likely that the platform would not be chosen at this point in the design process. It's also possible that you would carry the design to a certain point and then turn the design over to another team (or a subcontractor) to complete the implementation. In either case, the less restrictive the requirements are, the better.

As far as the project goes, there are not "right" and "wrong" requirements. Any set of requirements that completely and correctly specifies the elevator is technically correct. It's more a matter of "better" or "worse". This is a matter of experience, and one of the goals of the project is to help you develop that experience. As the design process progresses, you will see the consequences of the requirements that you wrote. Try to pay attention and learn from the process. Were your requirements easy or hard to implement? Did you have to make significant changes to your requirements in order to implement them? If so, consider what you needed to do to make your requirements "better", so that you can do that from the beginning in the future.

4.4 Project 4 –Time-Triggered Design

The time-triggered approach is not the only way to design a system, nor is it appropriate for all systems. Since time-triggered design is not usually taught in other courses, we force you to develop a pure time-triggered design so that you will have the approach available as a tool.

The shift we make from project 3 to project 4 is that, instead of responding to messages when they are received (responding to the message received event is the "event trigger"), the controller executes periodically (this is the "time trigger"). The controller makes decisions based on the current state of the system at the time of execution.

Time-triggered is more than just remembering the last value of messages that have been sent. Not only does the system remember message values, but each controller periodically transmits updates whether the value of the message has changed or not. This is what really makes the stored message values into state variables.

In the event triggered case (without periodic updates), you need (in the worst possible case) the entire message history of the system in order to have a complete picture of the current system state. In the time triggered case, the last full message round gives you a complete picture of system state.

One limitation of time-triggered design is that it cannot handle events that happen too quickly for the system to detect. Choosing the appropriate periods for controllers (based on the time constants of the system) is a very important part of the time-triggered design process. What if, for example, we ran the ButtonControl object with a one second period? If a button press lasted less than a second (very likely) and fell between executions of the controller, then the controller would not detect that the button press had occurred. So in the elevator (and the soda machine), we run the button controllers at 100ms because this is a granularity that will detect normal button presses.

4.5 Project 4 – Benefits of Time Triggered Requirements

To see how time-triggered design simplifies the requirements writing, compare the time-triggered and event-triggered requirements for the VendControl object. The first four ET requirements deal with setting state variables to remember message values. The last four requirements (eight if you count multipart requirements separately) deal with responding to the four different messages that might activate the vend mechanism. This entire set of requirements can be collapsed into the single multipart requirement R4.3.

4.6 Project 4 - Nested Statecharts

In the project write-up, we advise you to avoid nested statecharts. It is difficult to implement nested states correctly, and the semantics of the nested states can easily be misinterpreted. Because it is so difficult to get right, we do impose certain restrictions on how you may use nested states. These are spelled out in the Project 5 write-up, and you should read them carefully if you plan to use nested states.

If you look at the statechart for ButtonControl in the Requirements II document, you will see that we have used a nested statechart to describe the flashing behavior of the button during a vend cycle.

The first reason for using the nested statechart approach (even though we consider this to be sub-optimal in most cases) is to show you an example that follows the restrictions we impose in the project write-up. Note that arcs into and out of the VEND state point to the VEND state itself, and not to any substate. Also note that the initial state arc inside the VEND state points to the FLASH_OFF state. That means that every time the vend state is entered, the controller also enters the FLASH_OFF state.

The second reason for using the nested statechart approach is to illustrate some of the implementation difficulties that might arise when we do so. If you look at the implementation of the ButtonControl object, you will see that a separate state variable is needed to track the current state of the states inside the VEND superstate. In order to meet the initialization condition, this variable is initialized to FLASH_OFF whenever the main state machine is in any state other than VEND. A separate switch() statement is nested inside the VEND case of the main state machine to implement the inner state machine.

This implementation doesn’t seem so complex, but that is because this is a relatively simple statechart. Consider the case where the IDLE and EMPTY states also had substates, and the FLASH_ON and FLASH_OFF substates had sub-substates. In this case, you would be juggling 6 state variables and trying to keep track of which variables should be initialized in which states. It is very likely that there would be complex bugs associated with your implementation.

Finally, consider the case where our restriction on arcs that cross superstate boundaries is relaxed. In this case, you would be allowed to create a transition directly from FLASH_OFF to IDLE. But consider our implementation with one state variable for each set of nested substates in the statechart. Now, the transition checking inside the VEND substates can not only modify the currentVendState, they can also modify the currentOuterState as well. Allowing transitions across superstate boundaries breaks the abstraction of the nested states, and you are virtually guaranteed to have problems implementing such a statechart, which is why we do not allow those transitions in the project.

Since we suggest that you avoid nested statecharts, an alternate way to design the statechart for the ButtonControl object is given in Figure 1 below.

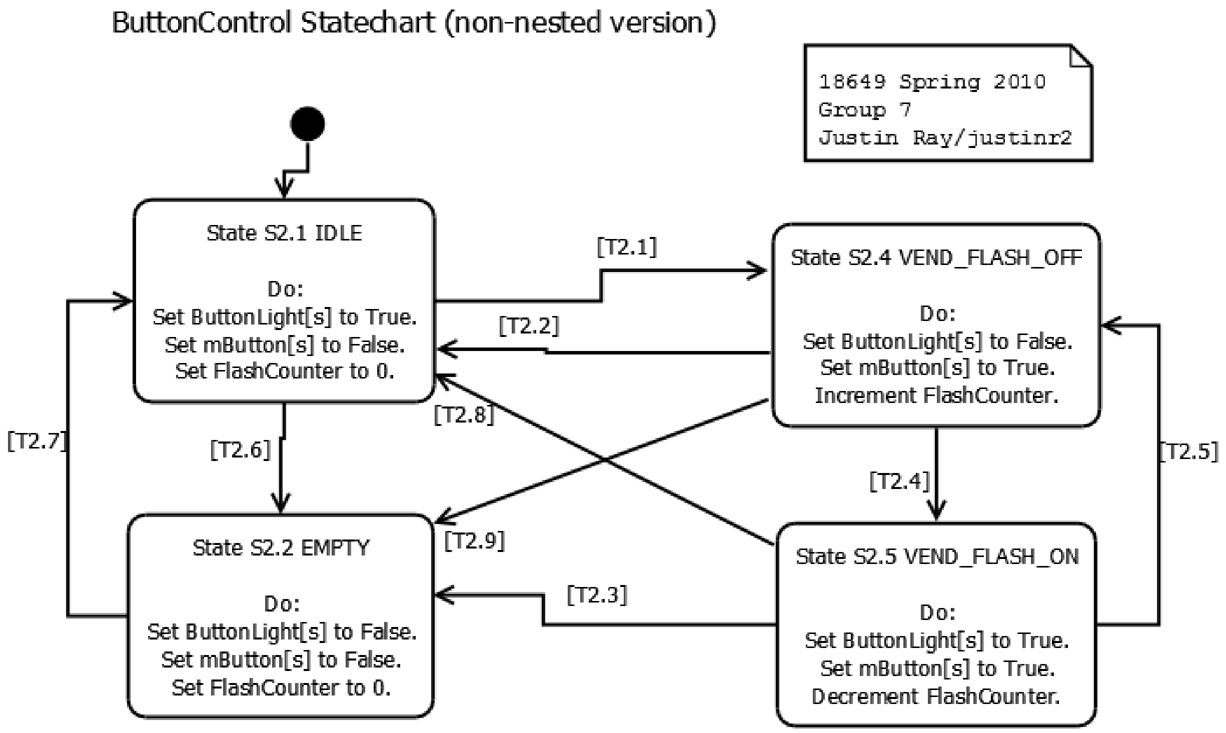

Figure 1: Alternate statechart for ButtonControl with no nested states.

We have essentially flattened the statechart. To do this, you must do several things:

- Create a top-level state for each substate (VEND_FLASH_OFF and VEND_FLASH_ON).
- The transitions that originally went into the VEND state (T2.1) should go into the new state that represents the initial state of the old VEND substates, in this case, VEND_FLASH_OFF.
- For transitions out of the VEND state, you must create a duplicate set of transitions out of each new state (VEND_FLASH_OFF and VEND_FLASH_ON). These are represented by T2.2 and T2.3 and the new transitions T2.8 and T2.9, where the guard conditions for T2.2 and T2.8 are the same, and the guard conditions for T2.3 and T2.9 are the same.

One important detail here is that when we create duplicate transitions in the flattened statechart, we give them unique numbers even though their guard conditions are the same. This avoids ambiguity in the specification and in traceability. If you altered the ButtonControl statechart along these lines, you would need to update the requirements-to-statecharts traceability and statecharts-to-code traceability to reflect the changes.

Again, we will reiterate: Nested statecharts are acceptable as long as they meet the guidelines spelled out in Project 5. However, we strongly advise you to stick with flat statecharts.

4.7 Project 4 – Pulsed Inputs and Outputs

Although the elevator does not have any inputs or outputs that are pulse oriented, the soda machine does. In order to detect a coin being inserted, the CoinControl object must detect the false-true-false pulse in the CoinIn input. Dealing with these pulses is inherently event-triggered. In this section, we will briefly describe how the pulse signal behavior is handled in statecharts. This will give you insight into how events may be handled in a pure time-triggered system.

In the behavioral requirements for CoinControl, there is a state variable "CoinReceived" that is used to describe the edge detection behavior. This variable is set true on a rising edge (R3.1a), and false on a falling edge (R3.6).

In the statechart, the CoinReceived state variable is implicitly implemented by the COIN_IN_1 and COIN_IN_2 states. The first state is entered when the CoinIn value becomes true (detect rising edge) and is executed only once due to the unconditional transition T3.2. The COIN_IN_2 state waits for the CoinIn value to become false (falling edge detector) before returning to the IDLE state.

In order to return a coin to the Customer, the CoinOut output must receive a false-true-false pulse as well. The pulse output is implemented for two scenarios: overpay, when a customer puts in an extra coin, and return, when the customer presses the coin return button. In both cases, the statechart requires a state for setting the CoinOut value true (the RETURN_1 and OVERPAY states), and a state for setting the CoinOut value false (IDLE performs this function in both cases).

During testing, it was found that the CoinOut output needed to be asserted for at least two periods in order to be reliably detected by the coin return object, so the OVERPAY_STRETCH and RETURN_STRETCH states were added (see Issue #16). The stretch states extend the time that the CoinOut output is set true. As an exercise, consider how you would modify the STRETCH states if you needed to stretch the pulse by more than one additional period.

In several places, you will note that there are some unconditional transitions. For example, consider T3.12 between OVERPAY and OVERPAY_STRETCH. There is no difference in the output of the controller in those two states. The only difference is that the CoinCount state variable is decremented in OVERPAY. Because of the unconditional transition, we guarantee that the decrement operation only happens one time. If we allowed event-triggered semantics into our statechart, we could implement these two states as a single state with an entry action or action on arc to decrement the CoinCount. The CoinControl state machine (as shown in the SMP) shows an acceptable way to implement the entry event using pure time-triggered semantics. There are a few places in the elevator where you might need to use this technique, but if you find yourself using it more than once or twice, it means you are missing the point of a time-triggered system and you should re-examine your approach or come to office hours for help.

4.8 Project 5 – Implementing Guard Conditions

Since java has a native Boolean type, you can use Boolean values directly in a test statement, as in:

```
  boolean a = false;
  if (!a) { //do something }
```

This is equivalent to:

```
  boolean a = false;
  if (a == false) { //do something }
```

The second form is what you will see in the example code (although it is not required in *your* code). This form, while less compact, is preferred because it is more explicit, and it more closely follows the notation used in the design.

4.9 Project 5 – Unit Testing Nested Statecharts

Because we have two possible internal states to VEND, a thorough test of the ButtonControl object must test both, hence the button_control_1.mf and button_control_2.mf tests (each one tests transitions out of a different VEND substate). This is another reason why the flattened statecharts are preferable. In the flattened statechart example given in Figure 1, each transition could be tested explicitly, which improves traceability and our certainty that the statechart has been completely exercised.

4.10 Project 5 – Unit Testing Negative Cases

The button_control_3.mf test is an example of a test that something checks to see if a transition does NOT occur. It checks that there is no transition from EMPTY to VEND if a button is pressed while the mEmpty signal is true.

4.11 Project 5/6 – Testability Problems with Unit and Integration Tests

This section covers tests of state changes that occur too rapidly to be reliably checked by the automated testing framework in the simulator. In these cases, manual verification of the controller operation (through checking debug messages) is required to complete the testing. This approach may be needed in Unit and Integration testing.

Sometimes there is a testability problem because some outputs occur for less than three message periods. The way the simulation framework is implemented, simultaneous events occur in pseudorandom order. This emulates jitter in a real system and reflects the idea that no two events are truly simultaneous. As a result, you will find that it takes longer than two message/controller periods before an output can be reliably tested. That is how long it takes state change messages to propagate through the system.

You can see an example of this testability problem in the testing of the pulse states in the CoinControl. Several of the states cannot be tested explicitly because the controller is only in that state for one period or because the outputs are identical to adjacent states. In this case, you should use the log messages from the controllers to verify that the appropriate states and outputs are correctly exercised in the tests. You can see examples of how to document these tests in the Unit and Integration test logs of the SMP.

Wherever possible, you should use the automated testing framework (assertions) to test your controllers and sequence diagrams. Automated tests are preferred because they can easily be re-run after changes are made to the system.

No part of the ELEVATOR design uses pulse detection or generation, so you should be able to generate reliable tests using the assertion framework. If you feel that you cannot adequately test your statecharts with the automated testing framework, then you MUST obtain TA approval for any test that uses log messages to verify a component.

These kinds of testing limitations are common in the real world. There is a whole area called "design for testability". The basic idea behind design for testability is that in addition to meeting functional requirements, you can add additional inputs or outputs to facilitate thorough testing.

One way to change the soda machine design to improve testability would be to impose an additional requirement that the pulse to the CoinOut actuator last at least 300ms. This would reduce the need for manual verification in the unit and integration tests because it would give the CoinOut output time to propagate reliably through the system. This change would be an acceptable one to make because it still meets the externally imposed requirement (a pulse of at least 150 ms) described in the Requirements I document.

4.12 Projects 5, 6, 7 – Usefulness of Testing

Look at the notes in the Unit, Integration, and Acceptance test logs and the related issue log entries. Each kind of testing identified bugs in the system. Some bugs are detectable at the individual controller level. These were detected in unit testing (at least one bug in each controller). Some bugs surface when testing the interaction between the controllers in integration tests, and finally, some bugs only emerge when the system is fully functional at the acceptance test level.

The bugs identified at the acceptance test were significantly harder to track down. Acceptance tests can be thousands of (simulated) seconds long and take several minutes to run to the point where the bug is exercised. It takes practice and familiarity with the system just to be able to inject breakpoints at the right time in the acceptance tests to study the operation of the system and identify the cause of a bug. This can be especially difficult if the bug occurs late in a long test. In acceptance tests, the problem is even harder because there are many objects instantiated and interacting, and the behavior of any one of them (or a combination of their behaviors) might be the cause of the bug.

By contrast, Unit Tests are the easiest to execute and debug. Typically, they only last for tens of simulated seconds, and there is only one object instantiated, so debugging is much simpler. You can also directly modify the inputs to the system (by altering the injected messages) to see if changes affect the way the bug manifests.

The point to take away from this is that the earlier in the testing process you can catch a bug, the easier it is to find and fix. For this reason, you should focus a lot of effort on Unit tests (and later, Integration tests) since thorough testing at these stages will greatly simplify your acceptance testing task.

4.13 Project 11 – Network Schedule Analysis

In Project 11, we impose limited bandwidth on the CAN network. The simulator has a bit-level CAN network simulation that accurately models the global priority behavior of real CAN networks. You can use the simulator to measure the bandwidth used by the system and compare it to the bandwidth you predicted in your theoretical analysis (using Rate Monotonic Scheduling). The simulated bandwidth should fall between the best and worst case scenarios predicted in your analysis.

In the SMP, this comparison revealed a flaw in the message period for the CoinReturn smart sensor (see Issue #17). This is just another example of how all the testing and analysis that is done on the system throughout the semester is designed to verify that you have built the system that you meant to build.

4.14 Projects 8-12 – A More Sophisticated Elevator

A large part of Projects 8 through 12 consists of extending and updating your elevator to add more sophisticated behaviors. There is no corresponding extension in the soda machine because, by this point in the semester, you have been exposed to all parts of the design process. You should continue to follow the rules and best practices described in the earlier projects and in this document as you complete your advanced elevator design.