Isolette–Precise Requirements

November 2, 2015

As stated in the course outline, you are required to read parts of the Handbook [1] each week. The Isolette example is described in [1] (Appendix A). The Handbook omits E/R-descriptions and function tables. In this document we discuss how to write a precise requirements document (e.g. for the Isolette) by providing E/R-descriptions, function tables and validation of the requirements.

Revisions

Date	Revision	Description
22 October 2015	1.0	Initial specification of Assignment: Isolette system
23 October 2015	2.0	Revised E/R descriptions, Alarm Off Hysteresis and Non-circular data-flow
02 November 2015	2.1	Revised the definition of held_for.

Contents

List of Figures

1. The Isolette

As stated in the course outline, you are required to read parts of the Handbook [1] each week. In this document we suggest how you might make the requirements precise using function table for the Isolette described in [1] (Appendix A). The Handbook omits E/R-descriptions and function tables. Some of the main learning outcomes for this course are:

- Demonstrate that you can write E/R-descriptions.
- Demonstrate that you can describe a complete and disjoint function table for a safety critical system (the Isolette) and that you can also derive important safety invariants that the function table must satisfy (to validate your specification).
- A LaTeX template (supplied in the SVN) is provided for precise documentation. You will need to complete the parts highlighted in yellow. By completing the provided template you will see the overall structure of a precise requirements documents.

As stated above, you will complete a precise Requirements Document. You will need to do further requirements elicitation from the customer (i.e. the instructor and TAs) where requirements are unclear. The answers you obtain to the elicitation questions must be recorded in the E/R-descriptions and function table (where relevant).

2. E/R-descriptions

Required Reading: Read Section 2, Section 3, Section 6, Appendix A and Appendix F of *Get it Right the First Time: Writing Better Requirements* [2]—available in the SVN. A summary of some of the above information follows.

2.1. What are requirements?

Understand and agree upon what users want before attempting to create solutions.

Finding out what is needed instead of rushing into presumed solutions is the key to every aspect of system development. Most technical problems can be solved, given determination, patience, a skilled team—and a well-defined problem to solve.

A precise requirements document will contain all the information needed by the

developers to build the system wanted by the users—and no more (i.e. it should not be polluted with design and implementation detail).

2.2. Why do requirements?

Writing requirements is not an end in itself. But it is also not merely extra work or an additional activity to ensure checks and balances or to meet a standard. Requirements have a real purpose in the development of any system. They are essential:

- To show results the users want from the system.
- To show traceability back to sources and the history of changes.
- To show what the organization needs.
- To show what the system must do.
- To form a basis for the design and design optimization (without being polluted by design or implementation detail)
- To enable a logical approach to change management.
- To partition the work out to contractors.
- To act as a foundation for testing and payment.
- To test the system or any of its parts during development.
- To communicate the basics about the system in non-technical terms to all participants

2.3. Using Everyday Language

We have stressed mathematically precise function tables and checking them for completeness and disjointness using PVS. However, there is also room for the use of informal English to describe requirements.

Everyday language is (most of the time) the only medium that users and developers share. Everyone can immediately understand requirements so written; therefore you might think writing in plain language would be the ideal way to define user needs.

But simple text isn't good at showing how different user needs fit together. After all, you want to make a system, not a mass of unrelated functions. So, you need to make a structure that will organize the requirements.

The structure must give readers insight into the text. A good structure shows the requirements at different levels of detail and allows readers to focus on one section at a time. The users have to understand and dominate the user requirements, whoever wrote them. A combination of textual requirements and a scenario-like structure of section headings is very effective. This can be supplemented if you use simple diagrams.

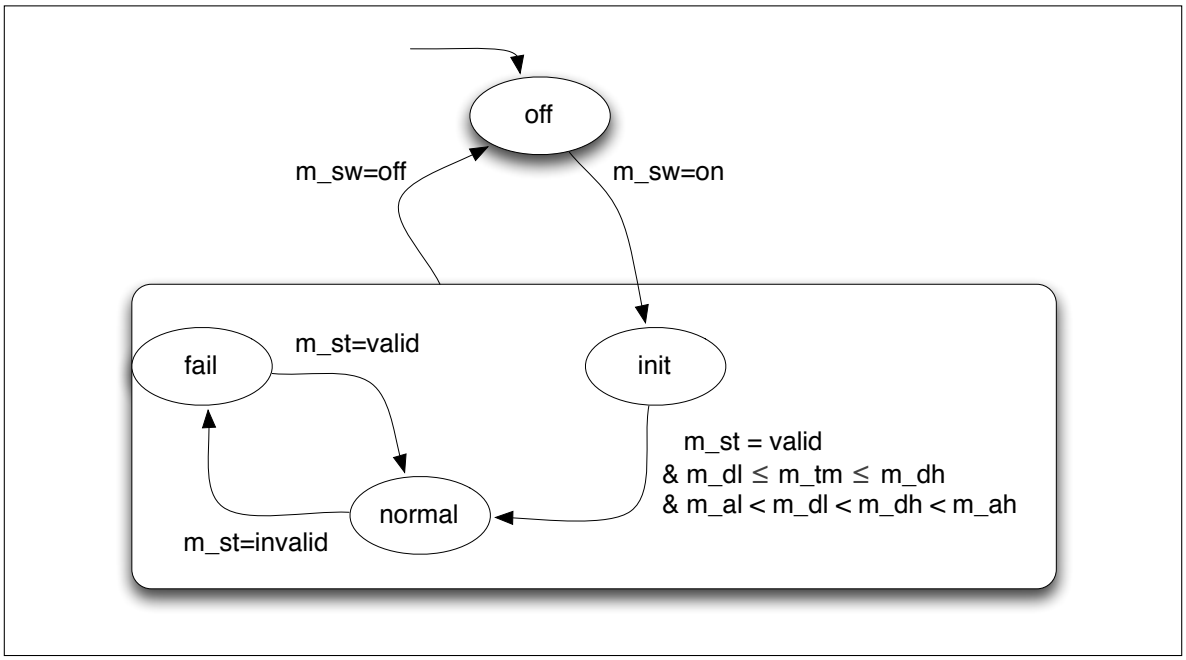


Figure 1: Statechart for the modes variable *c_md*

2.4. Criteria for good requirements

Section 3 of [2] contains criteria for good requirement descriptions such as:

- Is it correct? (does it capture a real user need; does it ask for something possible, do-able, legal)
- Is it complete? (a complete sentence)
- Is it clear? (unambiguous and not confusing)
- Is it consistent? (not in conflict with other requirements)
- Is it verifiable? (can we determine that the system meets the requirement?)
- Is it traceable? (uniquely identified and can be tracked)

2.5. Descriptions must be atomic and numbered

Section 2 of [2] contains the definition of good requirement descriptions. The main idea is to break the descriptions into numbered atomic sentences for clarity and traceability.

The system under design (SUD) in this case is a computer *controller* for the Isolette. The first requirement addresses the required modes of the controller.

REQ1	The *controller* shall operate in one of four modes: *off*, *init*, *normal* and *fail*.	See statechart in Fig. 1.

The atomic description is numbered REQ1 in the left cell. The description is in the middle cell. This description must be a complete sentence (subject and predicate) that expresses a *testable* user need (in this case for the nurse to understand the modes of the SUD).

The right hand cell provides further links to other parts of the requirements document such as function tables or use cases (for traceability). For REQ1, the right hand cell provides a link to a UML statechart (Fig. 1) which describes the sequencing of the modes. UML class diagrams, sequence charts and statecharts are often useful in conveying informal information.[1]

System modes are distinct discrete behaviours of the system externally *visible* to the users of the system.

In our case, the system modes will be explicitly displayed to nurses operating the Isolette. The nurse will only place a baby in the Isolette in the normal mode. An alarm is activated in the fail mode (see REQ3).[2] Given that the modes of the controller are visible to nurses (users of the system), the modes are not implementation detail—and may thus appear in the requirements document.

Identification of the major system modes is helpful when writing the detailed system requirements. Initially, it may not be clear if all the system modes have been identified or if all the modes listed are actually needed. The usefulness of any proposed modes will become clear during detailed system requirements specification, and the need for any additional modes will become more apparent.

Since the system modes are so closely related to the externally visible system behaviour, poor system mode design can lead to mode confusion that may cause the operator to become confused about what mode the system is in. This is an important safety concern that has been implicated in several safety critical accidents [1].

Some of the potential sources of mode confusion include lack of appropriate feedback to the operators, errors in interpreting or entering information in different modes, inconsistent system behaviours in different modes, different operator authority limits in

[1] You are required to familiarize yourselves with these diagrams.

[2] We have deviated from [1]. In [1], once the system is in the failed mode, the only way for it to re-enter the normal mode is for the nurse to turn the Isolette off and then on. In or case we raise an alarm in the failed mode, but return to normal if the error condition disappears. Your function tables must describe all these changes precisely and in detail.

different modes, silent (unannounced) mode transitions, and unintended side effects of mode transitions. Developing systems in which the modes are clearly indicated to the operators, and the system behaviour is consistent, easily anticipated, and understood by the operators, is a challenging task.

REQ2	In the *normal* mode, the temperature controller shall maintain current temperature inside the Isolette within a set temperature range (the *desired* range).	The *desired* temperature range is *m_dl* .. *m_dh*. If the current temperature *m_tm* is outside this range, the controller shall turn the heater on or off via the controlled variable *m_hc* to maintain the desired state.

It is important to provide a rationale for each E/R-description. For REQ2, we have:

Rationale: The *desired temperature range* will be set by the nurse to the desired range based on the infant's weight and health. The controller shall maintain the current temperature within this range under normal operation.

Safety critical requirements such as REQ2 emerge from a hazard analysis that must be performed by the safety engineers. The following relevant hazard was identified through the safety assessment process:

- **H1**: Prolonged exposure of Infant to unsafe heat or cold;
- *Classification*: catastrophic;
- *Probability*: $< 10^{-9}$ per hour of operation.

To ensure that probability of hazard H1 is 10^{-9} per hour of operation, the following derived safety requirement shall apply to the Isolette controller:

REQ3	In *normal* mode, the controller shall activate an alarm whenever • the current temperature falls outside the *alarm* temperature range (either through temperature fluctuation or a change in the alarm range by an operator), or • a failure is signalled in any of the input devices (temperature sensor and operator settings).	The alarm temperature range is $m\_al\,..\,m\_ah$. Monitored variable m_st shows "invalid" when any of the input signals fail.

REQ4	Once the alarm is activated, it becomes deactivated in one of two ways: • The nurse turns off the Isolette; • The alarm has lasted for 10 seconds, and after 10 seconds or more the alarm conditions are removed.	Refer to the relevant tables of monitored and/or controlled variables and function tables.

The following is an E-description because it does not describe a function that the SUD might enforce; rather, it is a constraint imposed by the environment over which the SUD has no control.

ENV5	The current temperature received from the sensor is a a real number in the range 68.0 to 105.0°F.	Refer to the relevant tables of monitored and/or controlled variables and function tables.

This is important information. The computer controller need not deal with temperatures above or below the stated range. We may also use ENV5 as an assumption when we provide function tables–which may omit ranges that will not occur (in completeness and disjointness analysis). Here is another example of an E-description:

ENV6	The desired and alarm temperatures received from the operator are all in increments of 1°F.	Refer to the relevant tables of monitored and/or controlled variables and function tables.

As before, it is important to provide a rationale for each E and R description. For example, for ENV6, the rationale is: Marketing studies have shown that customers prefer to set temperatures in 1 degree increments. A resolution 1°F is sufficient to be consistent with the functional and performance requirements specified in the rest of the document.

We can mix the atomic descriptions with informal text showing rationale as we have done here. By placing the atomic E/R-descriptions in a box, we allow them to stand out from the informal text. The same procedure is followed in mathematical text books where theorems and definitions are placed in a box or a special numbered environment for easy reference.

2.6. Real-time and the held-for operator

Given that REQ4 provides a time (10 seconds) for the persistence of the alarm alert, appropriate attention must be paid to timing issues. Review the course slides on timing resolutions (TR) and response allowances (RA). See also [3] (also in the course SVN).[3]

Reference [3] provides a useful operator—the *held-for* operator. We use a simpler theory than [3] (see Appendix A), which we also explain in more detail and with examples below.

Our definition of the held_for operator is provided in Fig. 2. The figure also provides an example using the held-for operator on a monitored variable *input(i)* — shown in green. The arguments of the held-for operator are:

- A predicate in DTIME— in or case the function $hi : [DTIME \rightarrow \mathbb{B}]$.
- A duration, i.e. a non-negative real—in our case 2 seconds.

[3]The discussion of tolerances in the [3] is out of scope for this assignment. However, the rest of the article provides an nice review of timing resolutions and response allowances.

Note that in Fig. 2, $duration/\delta = 1$. The sequence of conjectures (all can be proved) demonstrates whether $held\_for(hi, 2)(i)$ holds at $i \in 0..4$.

- $held\_for(hi, 2)(0)$ does not hold (see conjecture check1a). In fact, the held-for operator is always false at $i = 0$ due to the required consequent $j \geq 0$ for all j (note that its type is $j \in \mathbb{Z}$). So even if predicate hi would hold at all time instants, the hold-for operator will not hold until that instant where hi has held for a duration of 2 seconds (the earliest point being $i = 2$).
- Thus, $held\_for(hi, 2)$ does not hold at $i = 1$ because there has been an insufficient duration of highs. Note that $hi(0) =$ FALSE and $hi(1) =$ FALSE.
- The first instance at which $held\_for(hi, 2)$ holds is at $i = 4$ (see check6)—where $hi(2) =$ TRUE and $hi(3) =$ TRUE and $hi(4) =$ TRUE. .

Thus we might have a requirement that asserts that when $held\_for(hi, 2)(i)$ holds on the monitored input an alarm is sounded (e.g. $alarm(i) =$ TRUE) at the controlled output.

In Fig. 3, we use the held-for operator to describe duration of the output (the controlled variable). Here care must be exercised not to specify output in a circular manner. The monitored variable is *pressure* (in green) and the controlled variable is *alarm* (in red).

In the figure, a use case is provided. The timing resolution is 0.5 seconds (i.e. $\delta = 0.5$). The pressure does not go high until $i = 2$ (i.e. at $t = 1$ second). Once the pressure goes high, the alarm must sound for a duration of 1.5 seconds.

What response function table will make the Use Case hold?

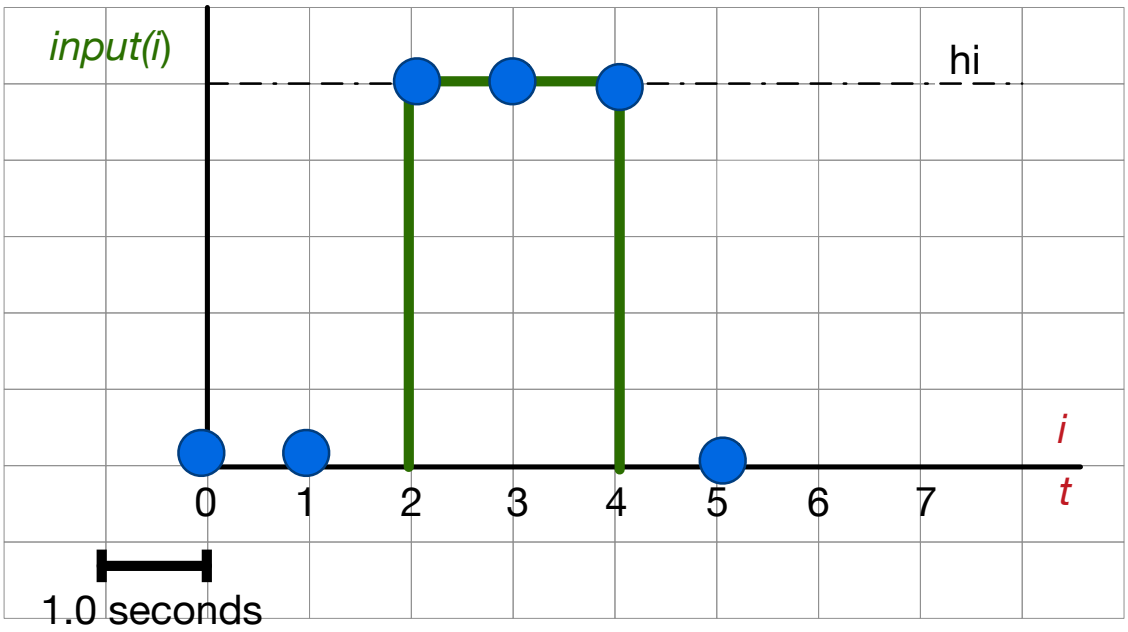
Fig. 4 provides a function table *reponse* that will ensure the truth of the Use Case, using the hold-for operator.

The predicate $held\_for(p, 1.5)(i - 1)$ holds at $i = 6$ because in the immediately preceding three digital time units ($j \in 2\,..\,5$) spanning 1.5 seconds, $pressure(j)$ holds true.

However, at instant $i = 4$ and below, the held-for predicate is false as shown in Fig. 5. The left operand ($0 \leq j$) of the consequent $0 \leq j \wedge p(j)$ in the held-for definition ensure that the held-for is always false at instant $i = 0$, and remains so until the duration has ticked off.

Note that the response function table specifies $alarm(i) =$ FALSE when $p(i) < hi \wedge held\_for(alarm, 1.5)(i - 1)$, i.e. in describing the alarm output at instant i we must use the hold-for operator on the previous instant $(i - 1)$ of the output so as not to run into circular definitions. The complete theory is shown in Fig. 6.

```
  DURATION: TYPE = nnreal
  held_for(p: pred[DTIME], d: DURATION)(i:DTIME): bool =
    (FORALL (j: int):
    i - (d / delta) <= j AND j <= i IMPLIES 0 <= j AND p(j))
```




```
  delta: posreal = 1.0
  IMPORTING Time[delta]
  input(i:DTIME): real =
    COND
      i < 2 -> 1.0,
      2 <= i AND i <= 4 -> 5.0,
      i > 4 -> 1.0
    ENDCOND
  hi(i:DTIME): bool = input(i) = 5

  check1: CONJECTURE held_for(hi,2)(0) = hi(0)
  check1a: CONJECTURE hi(0) = False

  check2: CONJECTURE (held_for(hi,2)(1) = (hi(0) AND hi(1)))
         AND (NOT held_for(hi,2)(1))
...
  check6: CONJECTURE held_for(hi,2)(4) = (hi(2) AND hi(3) AND hi(4))
         AND (held_for(hi,2)(4))
```

Figure 2: Held-for operator on a monitored variable *input(i)*

```
p:     [DTIME -> real]   % Pressure
alarm: [DTIME -> bool]
hi: real
i: VAR DTIME
usecase: CONJECTURE
         (FORALL i: response(i))
    AND p(0) < hi AND p(1) < hi
    AND p(2) = hi AND p(3) = hi
    AND p(4) < hi AND p(5) < hi AND p(6) < hi
  IMPLIES
         (NOT alarm(0)) AND (NOT alarm(1))
    AND alarm(2) AND alarm(3) AND alarm(4) AND alarm(5)
    AND (NOT alarm(6))
```

held_for(alarm,1.5)(i-1), i.e. $d/\delta = 1.5 seconds/0.5 = 3$

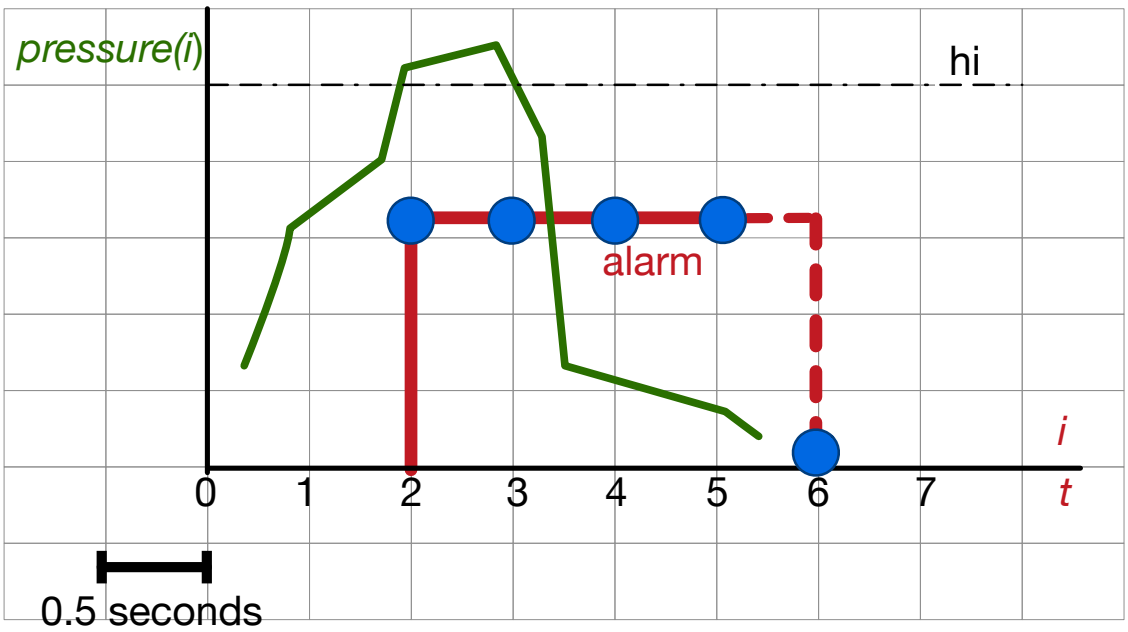


Figure 3: Use case for a 1.5 second output alarm when input pressure goes high

```
response(i:DTIME): bool =
    COND
        i = 0 -> NOT alarm(0)
      , i > 0 ->
        COND
            p(i) >= hi
          -> alarm(i)=TRUE,
            p(i) < hi AND held_for(alarm,1.5)(i-1)
          -> alarm(i) = FALSE,
            p(i) < hi AND NOT held_for(alarm,1.5)(i-1)
          -> alarm(i) = alarm(i-1)
          ENDCOND
   ENDCOND
```

held_for(alarm,1.5)(i-1), i.e. $d/\delta = 1.5 seconds/0.5 = 3$

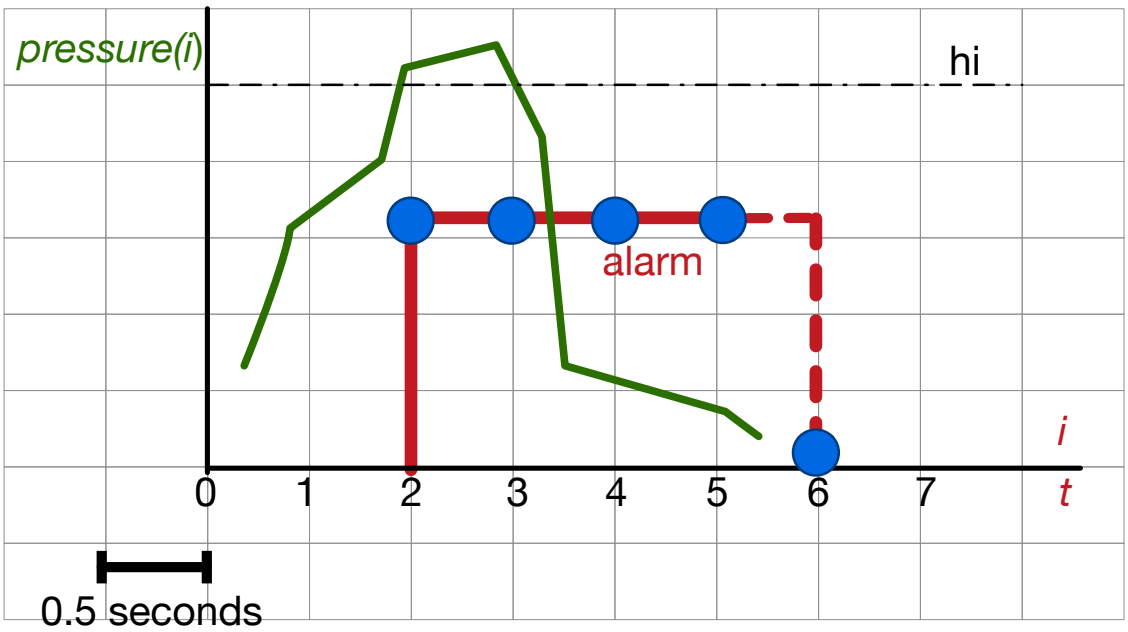
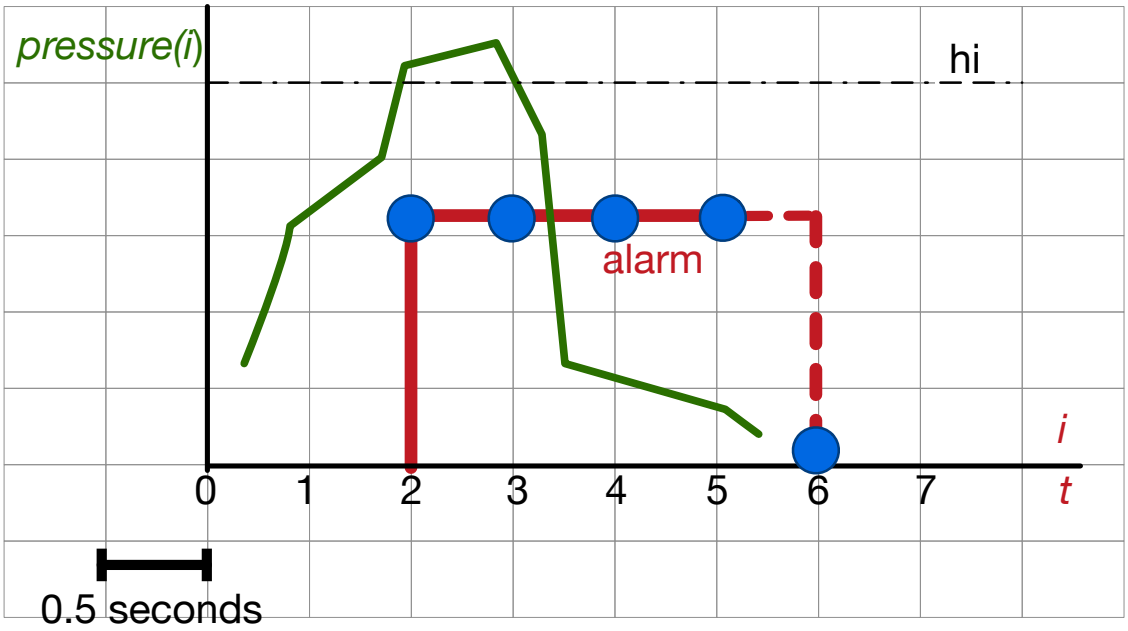


Figure 4: Using the held_for operator: $held\_for(p, 1.5s)(i)$



i		**held_for(pressure,1.5)(i-1)**
0		false
1	¬ alarm(0)	false
2	¬ alarm(0) ¬ alarm(1)	false
3	¬ alarm(0) ¬ alarm(1) alarm(2)	false
4		false
5		false
6	alarm(2) alarm(3) alarm(4) alarm(5)	true

Figure 5: Table for the truth of the held-for operator

```
alert: THEORY
BEGIN
  delta: posreal = 0.5 % TR = 0.5 seconds
  IMPORTING Time[delta]

  p:     [DTIME -> real]  % Pressure
  alarm: [DTIME -> bool]

  hi: real

  response(i:DTIME): bool =
    COND
        i = 0 -> NOT alarm(0)
      , i > 0 ->
        COND
            p(i) >= hi
          -> alarm(i)=TRUE,
            p(i) < hi AND held_for(alarm,1.5)(i-1)
          -> alarm(i) = FALSE,
            p(i) < hi AND NOT held_for(alarm,1.5)(i-1)
          -> alarm(i) = alarm(i-1)
          ENDCOND
   ENDCOND

  i: VAR DTIME
  usecase: CONJECTURE
           (FORALL i: response(i))
      AND p(0) < hi AND p(1) < hi
      AND p(2) = hi AND p(3) = hi
      AND p(4) < hi AND p(5) < hi AND p(6) < hi
    IMPLIES
            (NOT alarm(0)) AND (NOT alarm(1))
      AND alarm(2) AND alarm(3) AND alarm(4) AND alarm(5)
      AND (NOT alarm(6))
END alert
```

Figure 6: Example of a function table using the held-for operator

2.7. Pitfalls to avoid

See Sections A and F of [2] for pitfalls to avoid.

For example, don't try to fit multiple needs into a single atomic description. R-descriptions which contain conjunctions (words that join sentences together) are dangerous. Problems arise when readers try to puzzle out which part applies, especially if the different clauses seem to conflict, or if the individual parts apply separately. Dangerous conjunctions include and, or, with, also.

Example of a Bad description: *The battery low warning lamp shall light up when the voltage drops below 3.6 Volts, and the current workspace or input data shall be saved.*

Don't ramble. Long rambling sentences, especially when combined with arcane language, references to unreachable documents, and other defects of writing, quickly lead to confusion and error.

Example of a Bad description: *Provided that the designated input signals from the specified devices are received in the correct order where the system is able to differentiate the designators, the output signal shall comply with the required framework of section 3.1.5 to indicate the desired input state.*

Refrain from designing the system. Requirements specify the design envelope, and if we supply too much detail we start to design the system. Going too far is tempting for designers, especially when they come to their favourite bits. Danger signs include names of components, materials, software objects/procedures, and database fields.

Example of a Bad description: T*he antenna shall be capable of receiving FM signals, using a copper core with nylon covering and a waterproof hardened rubber shield.*

Specifying design rather than actual need increases the cost of systems by placing needless constraints on development and manufacture. Often knowing why is much better than knowing what.

For Assignment 1, only the the parts highlighted in yellow, in the sequel, are to be done.

3. Changes in the specifications from REMH [1]

For the Assignment, for simplicity, clarity and safety—there are some changes in the specification of the Isolette, from those descriptions provided in [1]. Some of the changes include:

- The statechart for the modes is in Fig. 1.

- The monitored and controlled variables of the SUD are provided in the assignment template.[4]
- The R-description REQ4 requires that the alarm is held for 10 seconds.
- See below.

3.1. Constraints on the desired/alarmed inputs

The SUD is the computer *controller* to regulate temperature. Everything else, including the Operator Interface is in the exosystem (i.e. in the environment of the controller). The Appendix-A specification of the Isolette [1, pA-11] states:

> EA-OI-4: The Lower Alarm Temperature will always be less than or equal to the Lower Desired Temperature of $-1°$F.
>
> **Rationale**: If the Lower Alarm Temperature is greater than or equal to the Lower Desired Temperature, the Alarm could be activated while the Current Temperature is still in the Desired Temperature Range.
>
> EA-OI-6: The Lower Desired Temperature will always be less than or equal to the Upper Desired Temperature of $-1°$F.

The above constraints are stated in [1] as E-descriptions (i.e. environmental descriptions) on the Isolette. What EA-OI-4, for example, implies is that the *controller* can assume that *m_al* is always less than *m_dl* (like a precondition). Who ensures that this constraint is satisfied? Apparently there is a device in the exosystem called the Operator Interface whose function is to ensure that this constraint is obeyed.

However, for the purposes of this Assignment, we may not rely on these constraints on the monitored variables in specifying the computer controller (the SUD). Thus the controller shall generate an alarm if there is a problem with the relationship between the desired and alarmed inputs. **Rationale**: This provides an extra check on the alarm and desired inputs, given the safety critical nature of the Isolette.

Thus, the only constraints that we may assume are on the types and ranges of the desired and alarm inputs (as stated in Appendix A of REMH). For example, we may assume that desired lower temperature range of *m_d* is $97 .. 99$ and the lower alarm temperature range is always $93 .. 98$.

3.2. Alarm Off Hysteresis and Displayed Temperature

There appears to be an inconsistency in [1] in the specifiaction for turning the alarm off. A snippet of the specification is shown in Fig. 7. If we translate REQ-MA-3 to

[4] `a1-Isolette.pdf`.

- REQ-MA-3: If the Monitor Mode is NORMAL and the Current Temperature is greater than or equal to the Lower Alarm Temperature and less than the Lower Alarm Temperature +0.5°, or the Current Temperature is greater than the Upper Alarm Temperature -0.5° and less than or equal to the Upper Alarm Temperature, the value of the Alarm Control shall not be changed.

  Rationale: This provides a hysteresis that prevents transient alarms, see figure A-7.

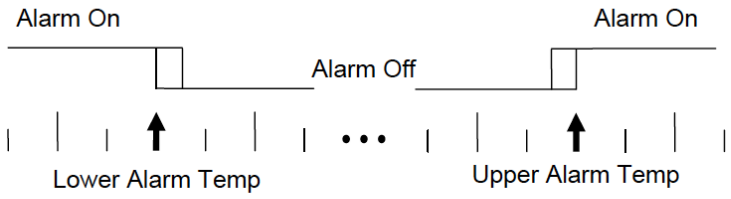


Figure A-7. Transient Alarm Hysteresis

- REQ-MA-4: If the Monitor Mode is NORMAL and the value of the Current Temperature is greater than or equal to the Lower Alarm Temperature +0.5° and less than or equal to the Upper Alarm Temperature -0.5°, the Alarm Control shall be set to Off.

  Rationale: This turns the alarm off at the same moment that the Displayed Temperature shows a value greater than the Lower Alarm Temperature and less than the Upper Alarm Temperature.

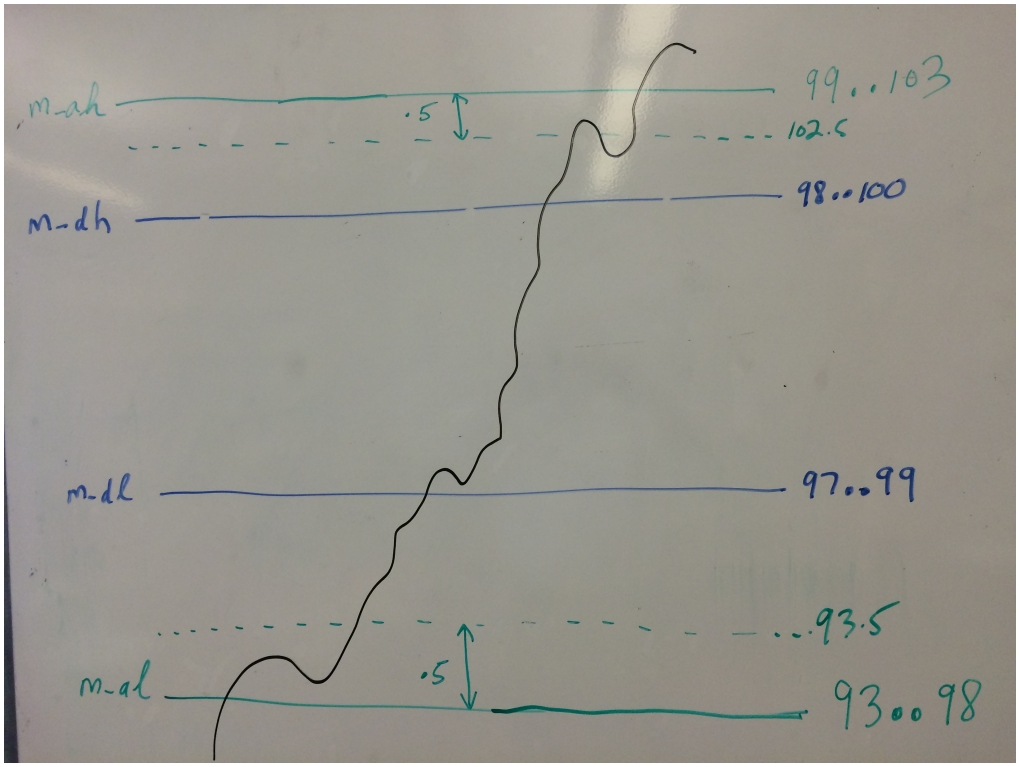


Figure 7: Hysteresis requirement for management of Alarm Off in [1]

mathematical notation we find that the no change hysteresis band is defined by:

$$\text{no-change hysteresis band for low alarm}: \quad m\_al \leq m\_tm < m\_al + 0.5 \tag{1}$$
$$\text{no-change hysteresis band for high alarm}: \quad m\_ah - 0.5 < m\_tm \leq m\_ah \tag{2}$$

We assume, for simplicity, that standard engineering rounding will be used, i.e.

$$c\_td = \lfloor m\_tm + 0.5 \rfloor \tag{3}$$

Thus for example, assume that the high alarm is on when the current temperature is above 103 and then decreases into the hysteresis band (NC). When the temperature further decreases to 102.5 which is outside of the hysteresis band according to (2), the alarm is turned off. However, standard rounding means that *c_td* displays 103—meaning the alarm should still be on. This inconsistency will be a source of confusion for the nurses.

We thus need to change (2) to:

$$\text{no-change hysteresis band for high alarm}: \quad m\_ah - 0.5 \leq m\_tm \leq m\_ah \tag{4}$$

The proof that (1) and (4) are consistent with rounding scheme (3) is provided in Fig. 8.

4. Function Tables Require Non-circular Data Flow

When writing function tables that depend upon each other, it is important to avoid circularity in the data flow.

Consider the two inputs m_1 and m_2 (the monitored variables) in Fig. 9 which are square waves coming from the ecosystem. The goal is that (a) the output c_1 (a controlled variable) shall provide a count of the total number of times the the inputs change from 0 to 1 and (b) c_2 shall provide a count of the average number of transitions over time.

It is convenient to introduce internal states s_1 and s_2 to specify the counts of up-edges coming into the system as shown in Fig 10. Output c_1 is defined in terms of the internal states, and output c_2 in terms of c_1.

4.1. Circularity and data-flow

In Fig 10, $c_1(i) = s_1(i) + s_2(i)$. Is there any circularity in the data flow? As it happens, this specification is valid. Fig. 11 provides conditions under which non-circular data-flow is preserved. Function tables must be written in a fashion that avoids circularities.

$$c\_td = \lfloor m\_tm + 0.5 \rfloor \tag{0}$$

$$\text{no-change hysteresis band for low alarm} : m\_al \leq m\_tm < m\_al + 0.5 \tag{1}$$

$$\text{no-change hysteresis band for high alarm} : m\_ah - 0.5 \leq m\_tm \leq m\_ah \tag{4}$$

The condition that turns the alarm off (if the alarm had been on for temperatures above m_ah) is $m\_tm < m\_ah - 0.5$, which is the negation of the left inequality of (4). The rationale behind REQ-MA-4 tells us that the moment when the alarm is turned off—the displayed temperature is strictly below the upper alarm temperature m_ah. Hence the first equivalence below. Likewise for the second equivalence.

```
  (1a) ⌊x⌋ < y ≡ x < y
  (1b) y ≤ ⌊x⌋ ≡ y ≤ x
    with x ∈ R, y ∈ Z
  (0) c_td = ⌊m_tm + .5⌋


proof of c_td < m_ah ≡ m_tm < m_ah-.5

      c_td < m_ah
    =  { (0) }
      ⌊m_tm+.5⌋ < m_ah
    =  { (1a) }
      m_tm+.5 < m_ah
    =  { real arith. }
      m_tm < m_ah-.5

proof of m_al < c_td ≡ m_al+.5 ≤ m_tm

      m_al < c_td
    =  { integer arith. }
      m_al+1 ≤ c_td
    =  { (0) }
      m_al+1 ≤ ⌊m_tm+.5⌋
    =  { (1b) }
      m_al+1 ≤ m_tm+.5
    =  { real arith. }
      m_al+.5 ≤ m_tm
```

Figure 8: Proof that hysteresis bands (1), (4) are consistent with rounding scheme (3)

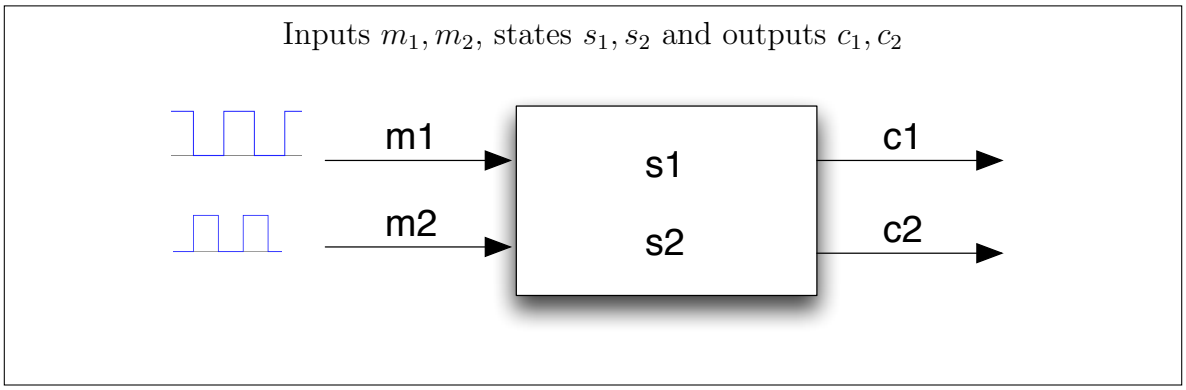


Figure 9: Specification of a Function Block

		$s_1(i) : \mathbb{N}$
$i = 0$		0
$i > 0$	$m_1(i\text{–}1) = 0 \ \wedge \ m_1(i) = 1$	$s_1(i\text{–}1) + 1$
	$m_1(i\text{–}1) = 1 \ \vee \ m_1(i) = 0$	NC

	$c_1(i) : \mathbb{N}$
true	$s_1(i) + s_2(i)$

		$s_2(i) : \mathbb{N}$
$i = 0$		0
$i > 0$	$m_2(i\text{–}1) = 0 \ \wedge \ m_2(i) = 1$	$s_2(i\text{–}1) + 1$
	$m_2(i\text{–}1) = 1 \ \vee \ m_2(i) = 0$	NC

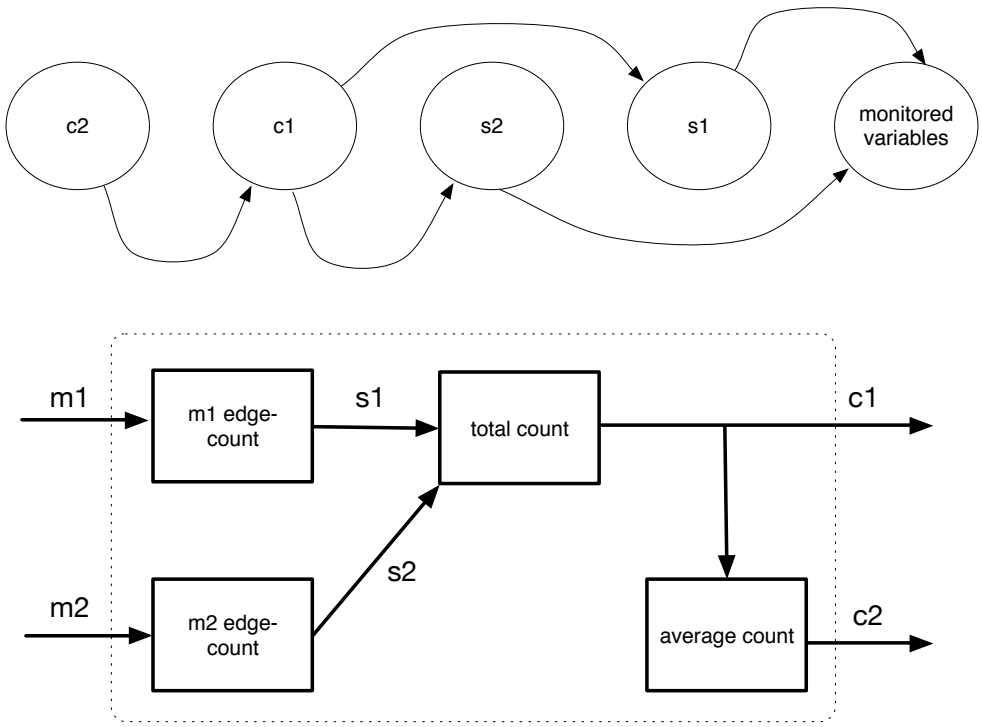
	$c_2(i)\mathbb{R}$
$i = 0$	0
$i > 0$	$c_1(i)/i$

Figure 10: Outputs c_1 and c_2 defined in terms of internal states s_1, s_2

Consider, for example, a different definition: $c_1(i) = c_1(i) + 1$. Here there is obviously a problem because the expression reduces to $0 = 1$, which is false. One cannot define $c_1(i)$ in terms of itself!

What about $c(i) = c(i\text{–}1) + 1$? That is ok, because c(i–1) is the value of $c(i)$ in the past. We are not asserting something that is incoherent. Rather, we are asserting that value of c at instant i is one more than its value in the previous instant. This is why the definitions of s_1 and s_2 are also valid as they only refer to their values in the past.

In Fig. 10, $c_2(i) = c_1(i)/i$ (for positive time instances). Can we define $c_2(i)$—a controlled variable—in terms of another controlled variable? Yes, provided there is no circularity. The following is circular and thus incoherent:



Let $x(j \,..\, k)$ stand for values of variable x at instants j to k, i.e. $x(j), x(j+1), x(j+2), \ldots, x(k)$. Then the data-flow is non-circular provided:

$$c_2(i) = f_1(c_2(0 \,..\, i\text{–}1), c_1(0 \,..\, i), s_2(0 \,..\, i), s_1(0 \,..\, i), m_2(0 \,..\, i), m_1(0 \,..\, i)) \quad (5)$$

$$c_1(i) = f_2(c_2(0 \,..\, i\text{–}1), c_1(0 \,..\, i\text{–}1), s_2(0 \,..\, i), s_1(0 \,..\, i), m_2(0 \,..\, i), m_1(0 \,..\, i)) \quad (6)$$

$$s_2(i) = f_3(c_2(0 \,..\, i\text{–}1), c_1(0 \,..\, i\text{–}1), s_2(0 \,..\, i\text{–}1), s_1(0 \,..\, i), m_2(0 \,..\, i), m_1(0 \,..\, i)) \quad (7)$$

$$s_1(i) = f_4(c_2(0 \,..\, i\text{–}1), c_1(0 \,..\, i\text{–}1), s_2(0 \,..\, i\text{–}1), s_1(0 \,..\, i\text{–}1), m_2(0 \,..\, i), m_1(0 \,..\, i)) \quad (8)$$

The monitored variables, being read-only, can always be referred to at any current or past instant. The control and state variables can also be written to—so it is with these that we have to exercise caution to ensure that the data-flow is valid.

Figure 11: Topological sort order: No circularities in the function tables in Fig. 10

Circular:

$$c_2(i) = c_1(i) + 1$$
$$c_1(i) = c_2(i) + 1$$

which asserts that $0 = 2$.

So what are the general rules to avoid circular data flows. The ability to place the variable dependencies in topological order will ensure that there are no circular data flows. The dependency graph for the function tables in Fig. 10 is in Fig. 11. Since there is no circularity in the dependency graph, the function tables are valid. See equation (6) in Fig. 11 for the general constraints on writing connected function tables (modules).

4.2. Using abstract state variables

It is not necessary to use internal state to describe the system. It is possible to write the control variables in the current instant purely in terms of the monitored variables at the current instant and earlier. For example, using the counting quantifier #, we may write:

$$\begin{aligned} c_1(i) = (\#j \in 1\,..\,i : m_1(j-1) = 0 \ \wedge \ m_1(j) = 1) \\ + (\#j \in 1\,..\,i : m_2(j-1) = 0 \ \wedge \ m_2(j) = 1) \end{aligned}$$

References

[1] US FAA. Requirements Engineering Management Handbook. Technical Report DOT/FAA/AR-08/32, U.S. Department of Transportation Federal Aviation Administration, June 2009.

[2] Teleogic. Get it Right the First Time: Writing Better Requirements. Technical Report MN-DOR-ED-GTR80-05-01, Telelogic, 2007.

[3] Alan Wassyng, Mark Lawford, and Xiayong Hu. Timing Tolerances in Safety-Critical Software. In J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM 2005*, volume LNCS 3582, pages 157–172. Springer-Verlag, 2005.

A. PVS Time theory for held-for operator

```
Time[delta: posreal]  : THEORY
BEGIN
  % digital time
  DTIME : TYPE = nat
  init(i : DTIME) : bool = i = 0

  % pseudo, digitized real time
  RTIME : TYPE = {t : nnreal | (EXISTS (i : DTIME) : t = i * delta)}

  % actual time
  TIME : TYPE = nnreal

  % Positive DTIME
  POS_DTIME: TYPE = posnat

  % conversions
  r2d(t: RTIME): DTIME = t / delta
  d2r(i: DTIME): RTIME = i * delta

  DURATION: TYPE = nnreal
  held_for(p: pred[DTIME], d: DURATION)(i:DTIME): bool =
    (FORALL (j: int):
    i - (d / delta) <= j AND j <= i IMPLIES 0 <= j AND p(j))

END Time
```

B. Outline of a Precise Requirements Document