2003-06-25

Comparative Rapid Prototyping, A Case Study

Luqi||Shing, M.; Puett, J.; Berzins, V.; Guan, Z.; Qiao, Y.; Zhang, L.; Chaki, N.; Liang, X.; Ray, W.; Brown, M....

IEEE

Luqi et al., "Comparative rapid prototyping: a case study," 14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings., San Diego, CA, USA, 2003, pp. 210-217.

http://hdl.handle.net/10945/63977

Comparative Rapid Prototyping, A Case Study [*]

Luqi, M. Shing, J. Puett, V. Berzins, Z. Guan, Y. Qiao, L. Zhang
N. Chaki, X. Liang, W. Ray, M. Brown, and D. Floodeen

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA

{ luqi, shing, jfpuett, berzins, zguan, yqiao, lzhang,
nchaki, xliang, wjray, mlbrown, dlfloode}@nps.navy.mil

Abstract

This paper presents a case study that explores the effectiveness of parallel conceptualization efforts to expose potential requirements issues in rapid prototyping. The case study consists of developing and comparing five design alternatives to model the safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump using a set of computer aided Software Engineering Automated Tools (SEATools). The prototyping effort showed that users can efficiently create / modify multiple parallel models and reason about their complexity using SEATools. The study also illustrates the usefulness of comparative rapid prototyping to identifying strengths and weaknesses of alternative designs, improving the final result. The case study also exposed numerous omissions and discrepancies in the requirements document and highlighted useful future enhancements for SEATools.

1 Introduction

Computer aided prototyping frees designers from implementation details by executing specifications via reusable components. Prototyping helps engineers firm up requirements and explore design alternatives. This paper presents a case study that explores the effectiveness of parallel conceptualization efforts to expose potential requirements issues in rapid prototyping. It consists of developing five design alternatives to model the safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump using a set of computer aided Software Engineering Automated Tools (SEATools), which is a PC-based computer-aided environment to support the modeling, analysis, prototyping and runtime testing of the systems under development. The SEATools is based on the Prototyping System Description Language (PSDL) [1-2], which is a high-level language designed specifically to support the conceptual modeling of real-time embedded systems. (Refer to the accompanying paper [3] for a detailed description of SEATools and PSDL.)

2 The CARA Software

CARA is a typical example of safety-critical software. This software is being developed by the Walter Reed Army Institute of Research to improve life support for trauma cases and military casualties [4-6] and has been used as a case study by several software engineering research groups [7-8]. CARA's purpose is to monitor a patient's blood pressure and to automatically administer intravenous (IV) fluids via a computer-controlled pump as needed to restore intravascular volume and blood pressure. The main functionalities of CARA include:

1. Monitoring patient's blood pressure.
2. Controlling a high-output infusion pump for patient resuscitation.
3. Displaying the vital information concerning the patient and the system to the caregiver.
4. Recording all the data to a log file.
5. Alerting the caregiver with an alarm during emergency situations.

As with most safety-critical software systems, CARA has extreme dependability requirements. There are two aspects to dependability: (a) ensuring the software meets its requirements, and (b) ensuring that requirements match the real-world needs of the system stakeholders. Our research has focused on applying our computer aided prototyping tools to address the second goal in the context of the case study.

[*] This research was supported in part by the U.S. Army Research Office under grant number 40473-MA-SP.

3 Comparative CARA Software Designs

To evaluate the effectiveness of SEATools in a parallel conceptualization effort, we formed five design teams, consisting of 2-3 people per team, and used SEATools to analyze the requirements of the CARA software. The goal of each prototyping effort was to develop an executable model to facilitate the analysis and understanding of the CARA requirements (especially with respect to timing and safety). In the interest of brevity only the top levels of each prototype design are presented in this paper. Readers can refer to [7] for the full detailed designs.

3.1 Design Model #1

The overall system environment consists of just three main components: The stretcher (called the *LSTAT*) is assumed to provide the majority of patient related information (e.g. blood pressures), the *Infusion_Pump* is the main item for control, and the *CARA* software system is the system driving the *Infusion_Pump* based on data received from the *LSTAT* (Figure 1).

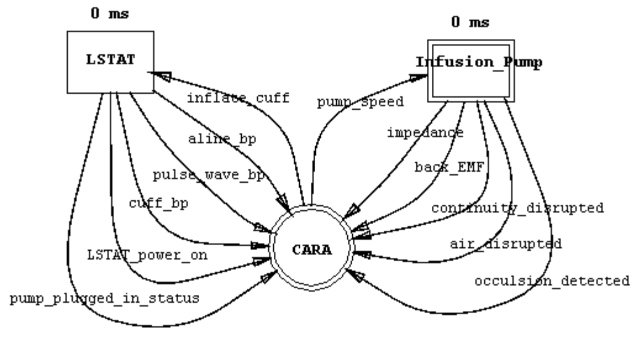


Figure 1. The top level of the CARA Software Model #1

In this design, the *CARA* architecture is organized into three modules as shown in Figure 2. The *Pump_Contol_Module* resolves an accurate blood pressure from various sources and then determines the appropriate flow rate for the pump. The *IO_Module* displays information and receives inputs from the CARA operator via a graphical user interface (GUI), and the *Management_Module* monitors the status of the pump, the lines, and the system for logging data into the historical resuscitation file. Figure 3 shows the decomposition of the *Management_Module*. Central to this module is a *manual_mode_interlock* operator that is primarily responsible for returning the system to a manual mode in case of any component failure. Feeding this operator is data from the *line_monitor* that monitors the sensor readings from the pump and LSTAT. Also feeding the *manual_mode_interlock* is a *processor_watchdog*. This watchdog is implemented on a separate processor and will sense any failure of the main processor and alert the operator (via the display). One final element of the *Management_Module* is the *resuscitation_file* where all information about the changing state of the CARA system is continually recorded.

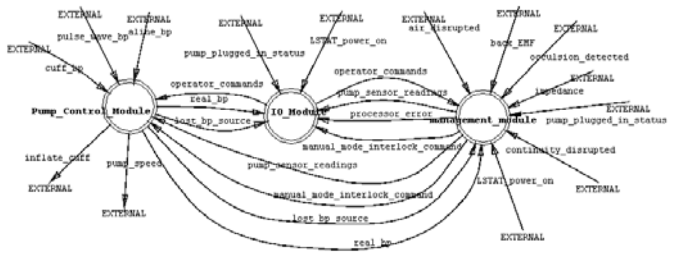


Figure 2. The CARA software architecture of Model #1

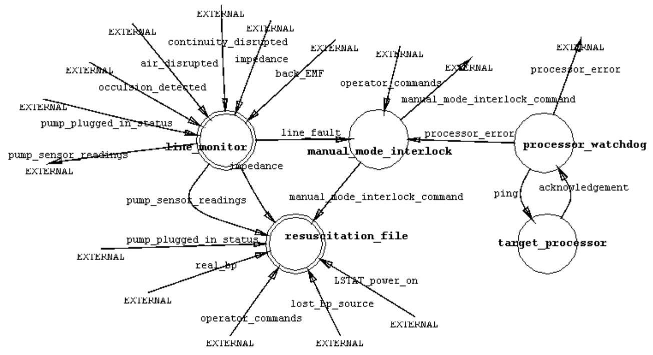


Figure 3. The Management module of Model #1

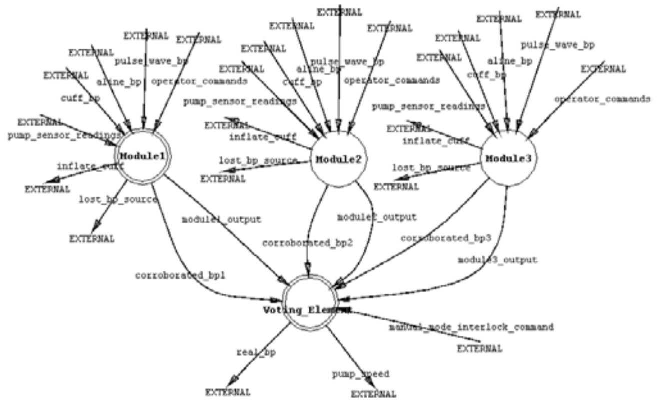


Figure 4. The Pump_Control module of Model #1

Figure 4 shows the *Pump_Control* module. This is the major safety critical module of the CARA. It is responsible for resolving an accurate blood pressure reading from the various input sources and then using this blood pressure to determine the correct input to the pump when the system is in auto-control mode. Because of the safety critical nature of this module, the design team chose to implement this with Triple Modular Redundancy (TMR). This specific safety architecture was not called for explicitly in the requirement statement; however, the safety environment implicitly requires some form of redundancy to ensure that the proper commands are sent to the pump. The TMR architecture uses three concurrent modules performing similar functions and producing similar output, but using different internal algorithms in their calculations. The *Voting_Element* operator is then responsible for determining which output to use. *Module1* is the only module of the three that has been fully decomposed. Modules 2 & 3 could be decomposed similarly to *Module1* but would use different algorithms (inserted at the programming stage).

3.2 Design Model #2

This design has four layers. The top level gives the outline of the infusion pump. It includes the *CARA* software, *pump*, *LSTAT*, *pressure_gauge* and *patient* (Figure 5).

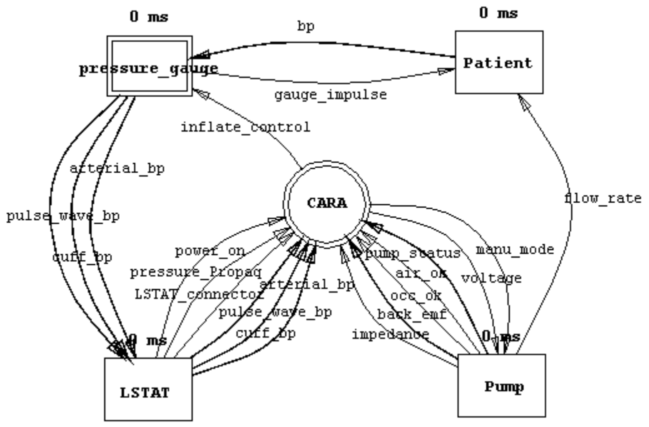


Figure 5. The top level design of Model #2

The *CARA* module is the central part of the prototype. It is further subdivided into a *monitor* module, a *control_alg* module, a *disp_alert_process* module and a *log_resu* module (Figure 6).

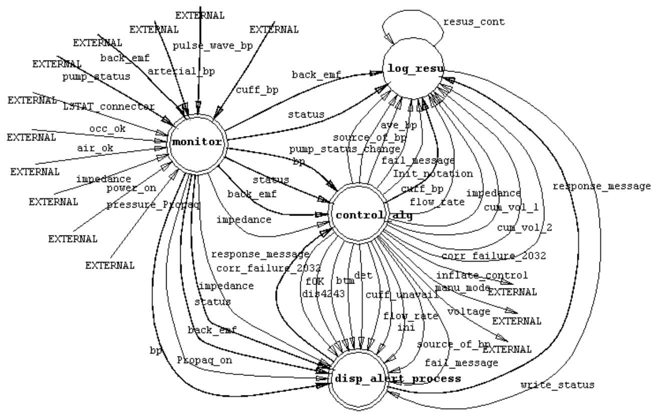


Figure 6. The CARA module of Model #2

The *control_alg* module is the main part of the CARA (Figure 7). It is responsible for calculating the flow rate and the cumulative volume of the pump according to the signal from back EMF and providing the voltage to control the pump flow rate. The algorithm also performs several other functions on the monitored signals.

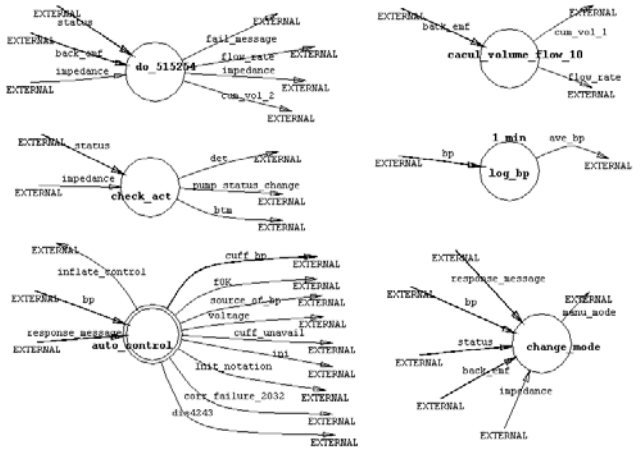


Figure 7. The control_alg module of Model #2

3.3 Design Model #3

In this model, the CARA software is split into two processes (Figure 8). Most of the controlling operation and computation is encapsulated in the *CARA_algorithm* operator. The *CARA_interaction* operator is responsible for information display control, alarm signal management, resuscitation file recording, and operator override handling. The overall architecture assumes two external sources (*pump* and *LSTAT_patient*), which provide data and receive control information to/from the CARA.

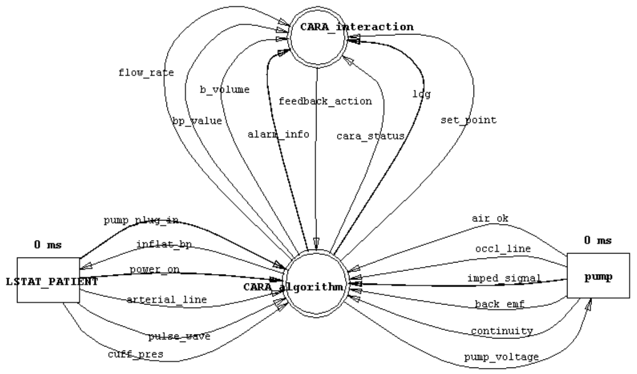


Figure 8. The top level design of Model #3

Figure 9 shows the decomposition of the *CARA_algorithm* module, which consists of six components. The sub-modules, *power_monitor*, *pump_monitor*, *bp_monitor* and *bp_corroborate*, are responsible for monitoring and validating the power on/off of the LSTAT, the connections and status of the signals from the pump, and the signal and status of the blood pressure from the LSTAT. Information derived from these monitors is fed into the other two modules, *pump_manual_control* and *pump_auto_control*. They are responsible for monitoring and controlling the infusion pump under two different modes of operation along with other important tasks (e.g., validating and corroborating blood pressure, switching control modes from automatic to manual and vise versa, etc.). The decomposition of the *pump_auto_control* sub-module is shown in Figure 10.

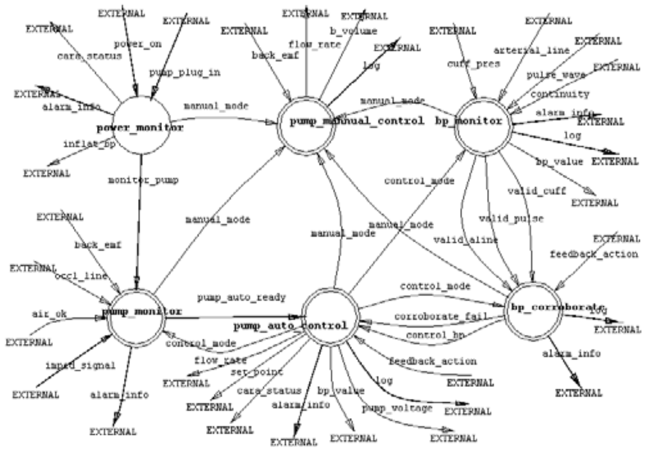


Figure 9. The CARA_Algorithm module of Model #3

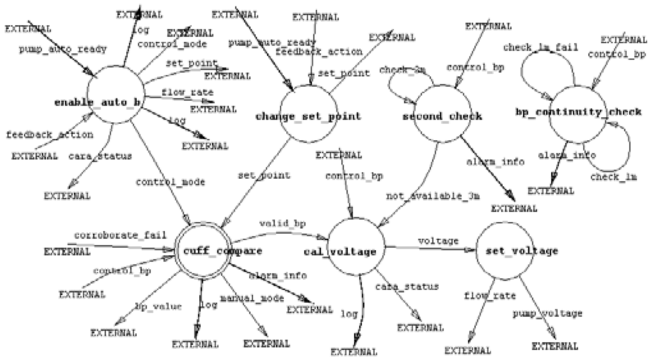


Figure 10. The pump_auto_control module of Model #3

3.4 Design Model #4

One of the major differences between this design and the others discussed in this paper is the inclusion of a *test_instrumentation* module in the top layer (Figure 11). The *test_instrumentation* module provides a GUI for the user to control the status of the pump, patient and the LSTAT simulations during prototype execution. Central to this model is the *CARA* module, which is decomposed into six sub-modules shown in Figure 12. The *monitor_bp* and *monitor_pump* modules are responsible for monitoring and validating blood pressures from the LSTAT and monitoring signals from the infusion pump respectively. Outputs from these modules are fed to the *control_pump* module to determine the voltage that drives the pump rate. They also lead to the *manage_alarm* and the *log_n_display_msg* modules to alert the user as needed. Inputs from the users are processed by the *manage_user_input* module and the resultant events are sent to the appropriate modules for further processing.

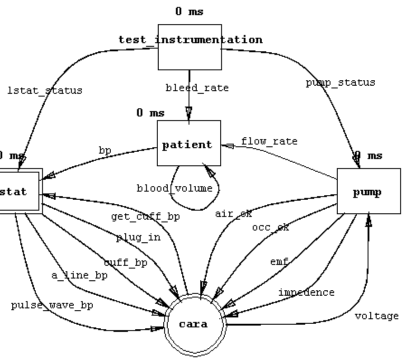


Figure 11. The top level design of Model #4

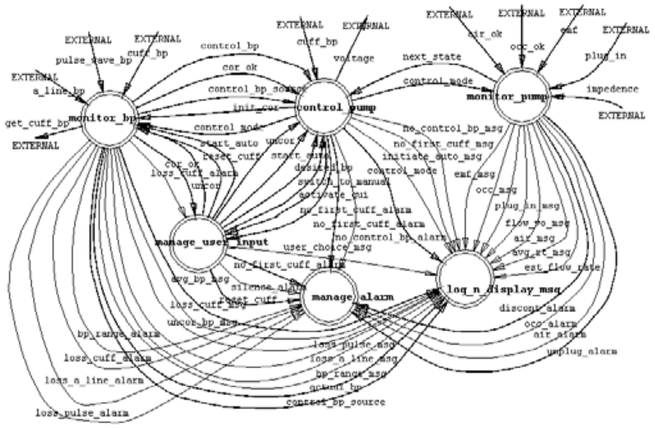


Figure 12. The CARA module of Model #4

Figures 13 shows the decomposition of the *control_pump* module. It models a state machine with 3 possible states {auto_off, auto_ready, auto_on}.

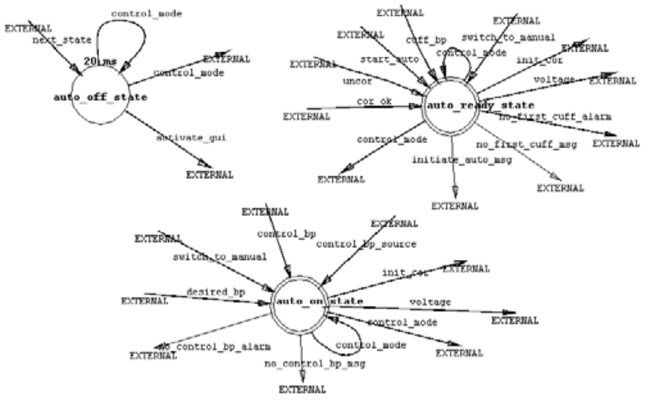


Figure 13. The control_pump module of Model #4

Depending on the value of the *control_mode* stream, only one of the three sub-modules can be triggered at any time. The *auto_ready_state* module is responsible for establishing the initial blood pressure when the user wants to start auto_control. It invokes the corresponding functions in the *monitor_bp* module to obtain the cuff pressure and corroborate the result with the other beat-to-beat blood pressure readings. It then uses the result to decide if the system is safe to enter into the auto control mode. The *auto_on_state* module (Figure 14) establishes the cuff blood pressure, computes the corresponding voltage to keep the pump working at the desired flow rate, and monitors the information from the *monitor_bp*, *monitor_pump* and *manage_user_input* modules to determine if it is necessary to switch back to manual mode.

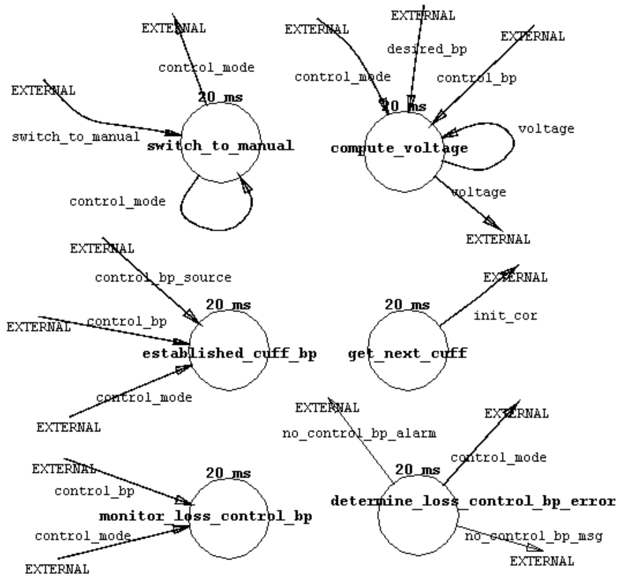


Figure 14. The auto_on_state module of Model #4

3.5 Design Model #5

This model consists of several cohesive subsystems that are created based on separation of concerns. The top layer of the model consists of the CARA software and four external sub-systems: *simulated_patient*, *LSTAT*, *pump* and *alarm* (Figure 15).

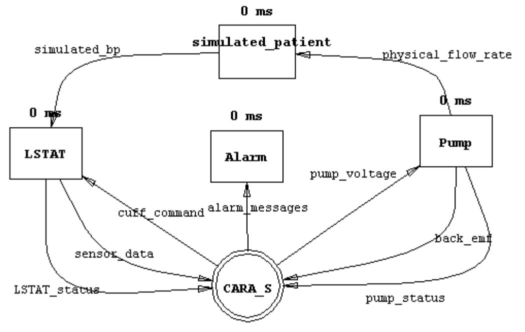


Figure 15. The top level of Model #5

Figure 16 shows the internal composition of the CARA software, which consists of six sub-modules: *BP_monitor*, *Safety_Monitor*, *Pump_monitor*, *Pump_controller*, *Logger* and *Display*.

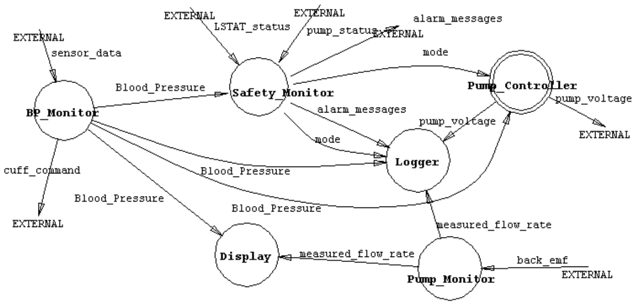


Figure 16. The CARA module of Model #5

4 Comparison of the Designs

To allow the greatest amount of design freedom between the designs, each team generated its prototype using only the supplied CARA requirements documentation [4-6]. Then, after completing the prototypes, the teams met and agreed on a set of criteria against which each prototype was assessed. These criteria included: architectural understandability, simplicity of design, requirements coverage, and incorporated safety features.

4.1 Understandability of the Model Architecture

Understandability of the model architecture is a common goal of all the designs. The design teams attempted to accomplish this goal via decomposition based on separation of concerns. Such an approach resulted in very similar top-level designs. They are all made up of the CARA control software interacting with the external environment consisting of the pump and the LSTAT.

The top-level designs reflect two different assumptions of the intended prototype. Models #1 (Figure 1) and #3 (Figure 8) model the prototypes as open systems, where the pump and the LSTAT modules represent interfaces to the actual hardware. Models #2 (Figure 5), #4 (Figure 11) and #5 (Figure 15) model the prototypes as closed systems. These models all include a simulated patient to model the effect of the IV infusion on the patient's blood pressure. In addition, model #4 includes a *test_instrumentation* module to facilitate run-time testing. The different interpretations of the intended prototype highlight the effectiveness of comparative rapid prototyping in exposing hidden assumptions early in the requirements analysis process.

Figures 2, 6, 9, 12 and 16 reveal the top-layer designs of the CARA software. All design teams identified the following functions of the CARA software: monitoring pump, monitoring blood pressure, controlling pump, log-monitoring blood pressure, controlling pump, logging and displaying messages, and managing user input and alarm. However, they varied in how these functions are presented in the hierarchical decompositions. Model #1 (Figure 2) provides the simplest design. It groups these major functions into three modules – *Pump_Control_Module* (for monitoring blood pressure and controlling pump), *IO_Module* (for displaying messages and managing user input), and *Management_Module* (for monitoring pump and logging messages). Model #2 (Figure 6) groups all monitoring functions (monitoring pump and monitoring blood pressure) into a single *monitor* module and buries the user input management deep inside the *control_alg* module. Model #3 (Figure 10), on the other hand, separates the message display and manage user input functions from the CARA control software and places them in a *CARA_interaction* module in the top level of the prototype (Figure 9). Models #4 (Figure 12) and #5 (Figure 16) have very similar designs. They both lay out all six functions explicitly in the top-layer of the CARA software.

Model #1 maintains its elegance and simplicity as we follow the decomposition to the lower levels of granularity. The whole architecture is easily followed and understood. The segregation of the identified safety critical functions from non-safety critical functions greatly enhances the visibility of the safety-critical aspects of the design. Model #5 also attempts to segregate the safety-critical functions from the non-safety critical ones in its design, but more work is needed to help flush out the design. Models #2, #3 and #4 suffer from varying degrees of complexity at the lower levels of granularity. Model #2 (Figure 7) attempts to divide the detail activities of the control algorithms into six concurrent processes, but fails to capture the event/response relationship among these processes. Model #3 gives a fairly complete design; though not as elegant as model #1, its architecture is easily understood. Model #4 gives the most detailed design. It attempts to reduce the complexity of the graph through the use of triggering conditions and timer operators. For example, its *pump_control* module models a state machine consisting of three states {*auto_off_state*, *auto_ready_state* and *auto_on_state*} (Figure 13). Depending on the value of the *control_mode* state stream, exactly one of the three processes in *pump_control* module can be activated at all times.

4.2 Simplicity of the Design

Simplicity of the design, in general, can be accomplished with sparse diagrams. Again, model #1 gives the simplest design. While all designs limit the number of operators in all levels to at most seven operators, they all suffer from varying degrees of data stream overcrowding. One way to solve this problem is by moving the functions around to form weaker coupled modules. For example, model #4 can greatly simplify the top-layer of the CARA software shown in Figure 12 if it follows model #1's design and places the *monitor_bp* module inside the *control_pump* module. Another way to reduce the number of data streams is through the use of composite streams. For example, combining all alarm signals into a common stream and letting the alarm manager differentiate the different sources and priorities of the signals and process them accordingly.

4.3 Requirements coverage

Due to inconsistency in the requirements and the limited time the teams has to work on the prototypes, total requirements coverage was not achieved in any of the designs. All designs covered most of the stated requirements in a broad sense, but they vary in detail when it comes to capturing the logic of the procedures stated in the requirements. All designs are able to identify the major functions of the CARA software and group them into different modules. With the exception of model #5, all models cover ~90% of the high level requirements and at least 50% of the detailed ones. For example, only models #3 (Figure 10) and #4 (Figure 14) attempted to model Requirements 27.1 – 27.4:

When the cuff pressure is being used for control:
If the mean BP is 60 or below, cuff pressures will be taken once per minute;
If the mean BP is (60 - 70], cuff pressures will be taken once every 2 minutes;
If the mean BP is (70 - 90], cuff pressures will be taken once every 5 minutes;
If the mean BP is above 90, cuff pressures will be taken once every 10 minutes.

4.4 Safety Aspects of the design

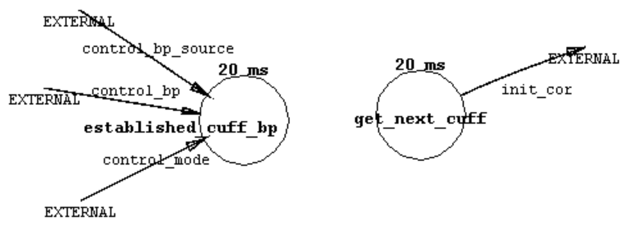
The design requirements, even with the questions and answers [5] provided, are very incomplete. In particular, the requirements documents do not identify or prioritize the functions, especially from a safety criticality perspective. As a result, the majority of the models (#2, #3 and #4) do not differentiate between safety-critical and non-critical functions.

The segregation of the identified safety critical functions from non-safety critical functions greatly enhances the safety of the design in models #1 and #5. In addition to those specified in the requirements document, model #1 includes two additional safety features in the design. First, it implements Triple Modular Redundancy (TMR) within the principal safety critical module (the *pump_control_module* shown in Figures 2 and 4). Second, it implements a processor watchdog function (on a separate processor) to alert the operator in cases of main processor failure (Figure 3). The redundant architecture on the blood pressure corroboration promises to substantially reduce the potential for a faulty monitor to drive the infusion when in fact it should not.

5. Conclusion

The prototyping case study revealed a lot of omissions and discrepancies in the requirements document [6]. The document does not provide any performance requirements – it provides primarily design requirements. For example, the system is required to monitor the infusion pump continuously, without explaining how continuous is "continuously", nor defining the maximum response time for the system to detect and handle a discontinuity event. There is no requirement for any kind of redundancy (with exception of redundancy implied by BP measurements). We went ahead and implemented a TMR (Triple Modular Redundancy) architecture on the key *CARA* algorithm module in Model #1.

The experiments showed that the prototyping language PSDL can effectively be used to model complex embedded software. PSDL's triggering guards and execution guards provide very convenient means for users to specify state machines explicitly without resort to target code. The timer feature is very useful in modeling complicated timing policies. For example, figure 17 shows a very simple way to model Requirements 27.1 – 27.4.



```
TRIGERRED BY ALL control_bp
    IF control_bp_source = cuff and
    control_mode =  auto_on_mode
MAXIMUM RESPONSE TIME 1 sec
MINIMUM CALLING PERIOD 50 sec
MAXIMUM EXECUTION TIME 50 ms
RESET TIMER one_min_timer
RESET TIMER three_min_timer
RESET TIMER five_min_timer
RESET TIMRE ten_min_timer
START TIMER one_min_timer
    IF control_bp <= 60
START TIMER three_min_timer
    IF control_bp > 60 and control_bp <= 70
START TIMER five_min_timer
    IF control_bp > 70 and control_bp <= 90
START TIMER ten_min_timer
    IF control_bp > 90
```

```
PERIOD = 30 sec
FINISH WITHIN 1 sec
MAXIMUM EXECUTION TIME
    50 ms
TRIGGERED IF
    one_min_timer >= 1 min or
    three_min_timer >= 3 min or
    five_min_timer >= 5 min or
    ten_min_timer >= 10 min
RESET TIMER one_min_timer
RESET TIMER three_min_timer
RESET TIMER five_min_timer
RESET TIMER ten_min_timer
```

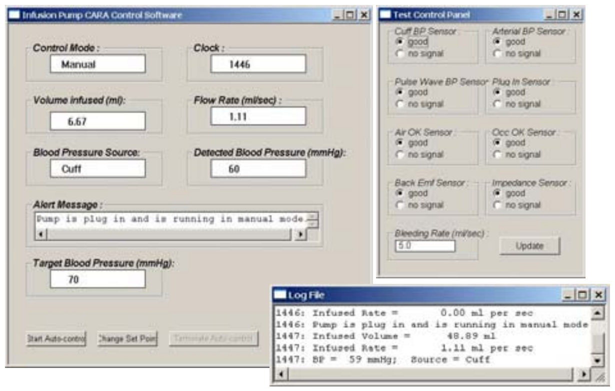
Figure 17. Model for a Cuff Blood Pressure Monitor Policy

The sporadic operator *established_cuff_bp* is triggered each time it receives a new cuff blood pressure reading and starts the appropriate timer for the next cuff-reading event. The *get_next_cuff*, on the other hand, polls the timers once every 30 seconds, and issues the init_cor command to the *monitor_bp* module if any of the active timer reaches its preset time trigger.

Since this design implements the policy via PSDL directly, users can accommodate any policy changes easily without the need to modify the source code. Moreover, since a cuff-reading event may also be generated by other conditions like the loss of a beat-to-beat blood pressure source detected by the *monitor_bp* module, the above design avoids any conflict with such event since it will reset its timers automatically whenever it receives a new cuff blood pressure reading.

SEATools provides the essential facilities for users to create and modify the models. It is very easy to reason about complexity using SEATools. When there are many data streams from one operator to another, such complexity is easy to identify and correct (e.g., by using a user-defined type for the stream). It is also easy to see when a particular module or operator is too complex and needs to be further decomposed. By trying to fully implement each requirement in the model it was clear which requirements were fully and consistently specified and which were not.

The suite provides an effective means to perform requirement consistency and understandability checking. It provides some degree of computer-aided consistency checking and data entry propagation at the user interface level, and complete semantic check and control code generation via the translator and the scheduler. For example, the executable prototype for Model #4 (Figure 18) has 20 composite operators and 89 atomic operators. The executable prototype consists of 14.7K lines of source code, 8.5K of which are generated by the translator and the scheduler of the SEATools.


Infusion Pump CARA Control Software

Control Mode : Manual
Clock : 1446
Volume infused (ml): 6.67
Flow Rate (ml/sec) : 1.11
Blood Pressure Source: Cuff
Detected Blood Pressure (mmHg): 60
Alert Message : Pump is plug in and is running in manual mode
Target Blood Pressure (mmHg): 70

Test Control Panel

Cuff BP Sensor : [✓ good] no signal
Arterial BP Sensor : [✓ good] no signal
Pulse Wave BP Sensor : [✓ good] no signal
Plug In Sensor : [✓ good] no signal
Air OK Sensor : [✓ good] no signal
Occ OK Sensor : [✓ good] no signal
Back Emf Sensor : [✓ good] no signal
Impedance Sensor : [✓ good] no signal
Bleeding Rate (ml/sec) : 5.0

Log File

```
1446: Infused Rate =       0.00 ml per sec
1446: Pump is plug in and is running in manual mode
1447: Infused Volume =      48.89 ml
1447: Infused Rate =       1.11 ml per sec
1447: BP =  59 mmHg;  Source = Cuff
```

Figure 18. The Graphical User Interface of the Executable Prototype for Model #4

The case study also highlights many useful future enhancements for SEATools, which include abstraction for data streams, visual queues for the declaration and use of timers, multiple views for requirements traces, better facilities for constructing user defined types, and the ability to designate which operators will be implemented on separate processors. It will also be very helpful if SEATools can provide quantitative comparison of the designs in terms of specific metrics such as complexity and requirement coverage met. Work is underway to incorporate such functions into the SEATools environment [9].

This case study illustrates the many benefits of comparative rapid prototyping of safety-critical software systems. It is apparent that such an approach is valuable for reasoning about the complexities, positive features, and shortcomings of alternative designs as well as for identifying and resolving inconsistency in software requirements sets.

References

[1] Luqi, V. Berzins, R. Yeh, "A prototyping language for real time software", *IEEE Transactions on Software Engineering*, 14(10), October 1988, pp. 1409-1423.

[2] Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Computer Languages*, 18, 1993, pp. 77-103.

[3] D. Drusinsky and M. Shing, "Verification of Timing Properties in Rapid System Prototyping", Proc. 14th Rapid System Prototyping Workshop, San Diego, 9-11 June 2003.

[4] WRAIR Dept. of Resuscitative Medicine, *Narrative Description of the CARA software*, Proprietary Document, WRAIR, Silver Spring, MD, Jan 2001.

[5] WRAIR Dept. of Resuscitative Medicine, *CARA Pump Control Software Questions, Version 6.1*, Proprietary Document, WRAIR, Silver Spring, MD, Jan 2001.

[6] WRAIR Dept. of Resuscitative Medicine, *CARA Tagged Requirements, Increment 3, Version 1.2*, Proprietary Document, WRAIR, Silver Spring, MD, March 2001.

[7] R. Alur, D. Arney, E. Gunter, I. Lee, W. Nam and J. Zhou, "Formal Specifications and Analysis of the Computer Assisted Resuscitation Algorithm (CARA) Infusion Pump Control System", Proc. Integrated Design and Process Technology (IDPT), 2002.

[8] Luqi, M. Shing, V. Berzins, J. Puett, Z. Guan, Y. Qiao, L. Zhang, N. Chaki, X. Liang, W. Ray, M. Brown and D. Floodeen, "SEA Environment for CARA Software", in the Tech. Report NPS-SW-03-001, Naval Postgraduate School, Monterey, CA, 2003, pp. 169-196.

[9] J. Puett, **"**Holistic Framework for Establishing Interoperability of Heterogeneous Software Development Tools and Models", *Proc. 24th Intl. Conf. on Software Engr.*, Orlando Florida, May 2002, pp. 729-730.