2003

Requirements-document-based prototyping of CARA software

Luqi

Springer-Verlag

Int J Softw Tools Technol Transfer (2004) 5: 370–390
http://hdl.handle.net/10945/51488

Int J Softw Tools Technol Transfer (2004) 5: 370–390 / Digital Object Identifier (DOI) 10.1007/s10009-003-0116-7

Requirements-document-based prototyping of CARA[1] software

Luqi[1], Z. Guan[1], V. Berzins[1], L. Zhang[1], D. Floodeen[1], V. Coskun[2], J. Puett[1], M. Brown[1]

[1] Software Engineering Automation Center, Naval Postgraduate School, 833 Dyer Road, Monterey, CA 93943, USA
e-mail: {luqi,zguan,berzins,lzhang,dlfloode,jfpuett,mlbrown}@nps.navy.mil
[2] Turkish Naval Academy, Istanbul, Turkey
e-mail: vedatcoskun@dho.edu.tr

Published online: 19 December 2003 –

Abstract. Computer-aided prototyping evaluates and refines software requirements by defining requirements specifications, designing underlying compositional architecture, doing restricted real-time scheduling, and constructing a prototype by using reusable executable software components. This paper presents a case study of the Computer Assisted Resuscitation Algorithm (CARA) software for a casualty intravenous fluid infusion pump and explores the effectiveness of performing rapid prototyping with parallel conceptualization to expose requirements issues. Using a suite of prototyping tools, five different design model alternatives are generated based on the analysis of customer requirements documents. Further comparison is conducted with specific focus on a sample of comparative criteria: simplicity of design, safety aspects, requirements coverage, and enabling architecture. The case study demonstrates the usefulness of comparative rapid prototyping for revealing the omissions and discrepancies in the requirements document. The study also illustrates the efficiency of creating/modifying parallel models and reason for their complexity by using the tool suite. Additional enhancements for the prototyping suite are highlighted.

Keywords: Rapid prototyping – Specification – Software tools – Requirements analysis – Parallel design

1 Introduction

This paper demonstrates how computer-aided prototyping from multiple viewpoints can clarify requirements for safety-critical embedded systems. The case study presented in this paper compares five different designs for the control software of a sophisticated medical device. All five designs were constructed from the same set of customer-supplied requirements documentation. We found that parallel redundancy in the prototyping and requirements analysis processes improved the quality of the results: different teams found different requirements defects and later cross comparisons identified defects that were not found by the individual teams.

The requirements for the Life Support for Trauma and Transport (LSTAT) system were given by a requirements definition document written using natural language expression [29–31]. This is a real system, and the requirements documents were written by domain experts who were independent of the prototyping teams. The purpose of the LSTAT stretcher is to sustain trauma patients in transit to a medical facility until they can receive emergency medical treatment. The LSTAT includes multiple blood pressure sensors and an intravenous (IV) infusion pump. The embedded Computer Assisted Resuscitation Algorithm (CARA) software automates the delivery of IV fluids to the trauma patient as needed by controlling the infusion pump based on the patient's blood pressure readings [1]. The algorithm calculates the drive voltage for the infusion pump. This determines the volume and rate of IV fluid administered. Proper operation of the CARA software should enable safe transport of critically injured patients without the need for continuous monitoring by medical professionals.

The situation and context of using CARA is complex and varied in terms of the level of the patient's blood pressure and the patient's different body effects after receiving IV fluid. The drive voltage used to control the IV fluid pump needs to be calculated dynamically. The requirements document gives detailed descriptions of the real-time constraints for the application process of the rescue strategies, which are crucial for satisfying CARA's inherent safety requirements. A delay in delivery or an improper amount of IV fluid to the patient can result in serious injury or loss of life.

[1] Computer Assisted Resuscitation Algorithm

CARA software is not only a control system but also an information system. It interacts with outside sensors and records all procedure data into a database for future reference. Also, to help the physician track the resuscitation process, CARA displays patient vital signs and system status information on a monitor. This requires seamless connections between the outside data monitoring, the inside software decision process, and the system hardware controlling the IV pump. To ensure that the CARA software satisfies its safety criteria, several specific requirements must be met:

(1) The monitoring and sampling of a patient's blood pressure should be accomplished in the required time to prevent losing data necessary for proper IV fluid control.
(2) The CARA software must choose between automatic or manual control of the infusion pump as determined by changes in a patient's blood pressure value and alert the physician to allow for timely execution of appropriate rescue efforts.
(3) The CARA software must be able to rapidly adjust the infusion pump's control values to maximize rescue effectiveness.

The design and implementation of the CARA software should fully consider the logical relationship between the control software, the peripheral hardware, the external inputs from the physician, and the ever-changing condition of the patient. Prototyping, as an economic way to build scale models and prototype versions of most systems, has proven an efficient and effective methodology for evaluation of proposed systems when acceptance by the customer or feasibility of development is in doubt [5, 12, 18, 27]. Computer-aided prototyping can build a scaled-down version of the software system and ensure that the requirements are satisfied before extensive effort is put into detailed implementation [9, 10, 14, 28].

Since the user requirements document describes the needs and boundaries of the software product and serves as a contractual agreement between the client and the developer, the completeness and accuracy of the user requirements document are essential for the success of the developed software. Early detection and correction of the faults in the user requirements document are the keys to keeping development costs down and building correct, reliable, and safe software. Past research shows that performing the operation in parallel can increase the design or inspection efficiency and improve the quality of the final product. Dual development uses two independent teams throughout the development, thereby improving the quality of the product at the end of each development step [25]. N-version programming uses the combined efforts of N independent designs and implementations to produce fault-tolerant code [2]. N-Fold inspection uses N independent teams with a single central "moderator" to inspect the user requirements for improving their reliability [11, 24, 26].

We used the Computer-Aided Prototyping System for Personal Computer (CAPS-PC) tool suite [7, 8] to prototype the CARA software. These tools are the result of our latest research on prototyping languages, real-time systems modeling and analysis, automatic code generation techniques, and rapid prototyping environments [20, 22]. The CAPS-PC tool suite helps create, modify, and maintain the requirements specification and architecture description documentation based on knowledge mapped from informal natural language descriptions. By building prototypes, we can check the reliability of the software attributes and monitor the characteristics of the software according to changes in the context environment. We can also explore the characteristics of high confidence embedded systems during the efforts of building and detailing the prototype models. We tested our tools and new user interface and explored the degree to which our design notations and models express the range of issues typical of high confidence embedded systems. A particular focus was to identify issues difficult to express in our current models and representations.

In prototyping the CARA software based on the requirements document, we explored five design alternatives by professional prototyping teams and analyzed the effectiveness of parallel conceptualization efforts to expose potential requirements issues. The teams used individual assumptions and interpretations of the requirements documents in the design alternatives. The effort demonstrated the effectiveness and efficiency of comparison and discussion of these different designs and viewpoints to find and fix faults in the requirements document.

2 Language and tools for analyzing, modeling, and prototyping complex systems

Department of Defense software systems fall somewhere within a continuous spectrum between pure information systems and pure control systems. All of these systems support the warfighter in one way or another, whether they are domestic warehouse inventory tracking systems or the embedded software in a smart projectile on the battlefield. These systems can be distributed, heterogeneous, and network-based, consisting of a set of components running on different platforms and working together via multiple communication links and protocols. These systems have many safety- and security-critical aspects and an associated need for high confidence [3, 22]. Hence we must develop models and languages to capture these requirements and attributes. We built upon our experience with specification and prototyping languages and refined the Prototype System Description Language (PSDL) for modeling and prototyping complex systems [13, 19, 22]. This section briefly summarizes our notations for high

confidence embedded systems, which we used for the CARA prototyping effort.

Formally, the external view computational model ζ' can be represented as follows:

$$\zeta' = (G, H),$$

where G is a functional emergent property vector that represents the functional requirements of an embedded system and H denotes a nonfunctional emergent property vector that reveals the nonfunctional requirements related to the high confidence of an embedded system [22]. By mapping nonfunctional and functional emergent properties presented in an external view of a model into local constraints, the internal view computational model can be derived. It is formally described as a 6-tuple:

$$\zeta = (S, E, C, D, F_1, F_2)$$

where:

$S = \{s_i | i \in [1, n]\}$, s_i is the component system, and n is the number of component systems;
$E = \{e_{jk} | j, k \in [1, n]\}$, e_{jk} is the set of interactions from component system s_j to component system s_k;
$C = \{c_i | i \in [1, n]\}$, c_i^p is the p-th constraint for s_i;
$D = \{d_{jk} | j, k \in [1, n]\}$, d_{jk}^q is the q-th constraint for e_{jk};
$C = F_1(G, H)$; $D = F_2(G, H)$, F_1 and F_2 are two maps that map emergent properties into local constraint sets on component systems and local constraint sets on interactions between component systems, respectively.

The main differences between the current and previous versions of PSDL are that constraints can be associated with interactions as well as with subsystems and that the open syntax for expressing an unbounded set of properties of the structural elements of the model has been given additional standard interpretations for expressing additional kinds of constraints.

2.1 Basic computation graph model

Pictorially, we can represent each s_i as a vertex of a computation graph and each e_{ij} as a set of directed edges from s_i to s_j in the graph [13, 16, 17, 19]. For example, the system shown in Fig. 1 corresponds to a system with

$$S = \{monitor\_bp, bp\_corroboration, pump\_control\}$$

and

$$E = \{\{last\_cuff\_value, bp\_source, bp\_ready\}, \{corrob\_ok\}, \{control\_mode\}\}.$$

The component `monitor_bp` can be associated with the set of constraints {`Maximum Execution Time = 100 ms`, `PERIOD = 500 ms`}, and the edge `last_cuff_value` is associated with the constraints {`LATENCY = 500 ms`}. The trigger condition of the component `bp_corrobration` is `BY SOME bp_ready`. That component generates output `control_mode` to the `pump_control` only under the condition `corrob_ok = true`.

To support the automated generation of glue and wrapper code, PSDL provides the capability to capture the attributes of target network systems. For example, Fig. 2 shows a PSDL model of a target network connecting three host machines.

2.2 Hierarchical computation graph model

The hierarchical graph model extends the basic computation graph model to support abstraction. A vertex s_i in a hierarchical computation graph may in turn be modeled in more detail as a subsystem $\zeta = (S, E, C, D, F_1, F_2)$, resulting in a hierarchical structure of nested computation graphs. For example, the `monitor_bp` vertex in Fig. 1 may be modeled as the graph shown in Fig. 3. Note that the children vertices (`monitor_cuff`, `monitor_pulse`, `monitor_a_line`, and `compare_bp`) inherit the constraints associated with the parent vertex (`monitor_bp`).

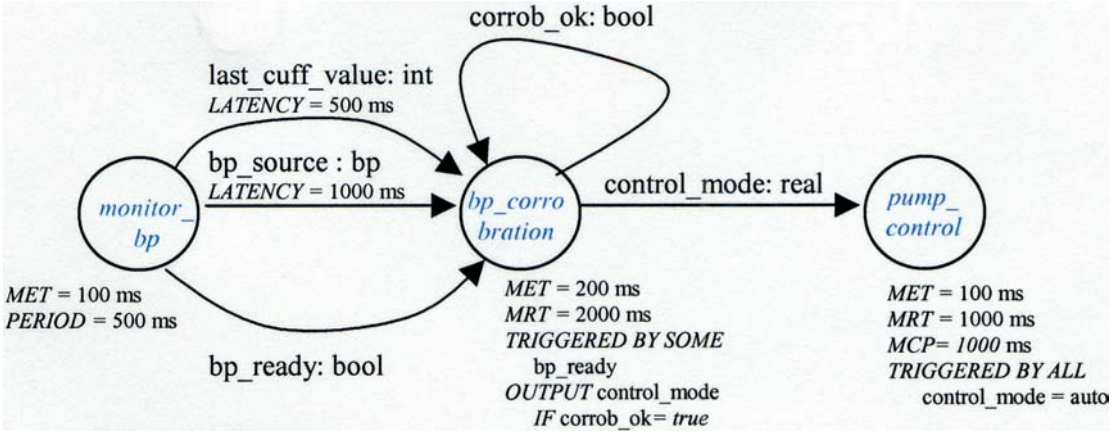

Fig. 1. The PSDL model for CARA system requirements

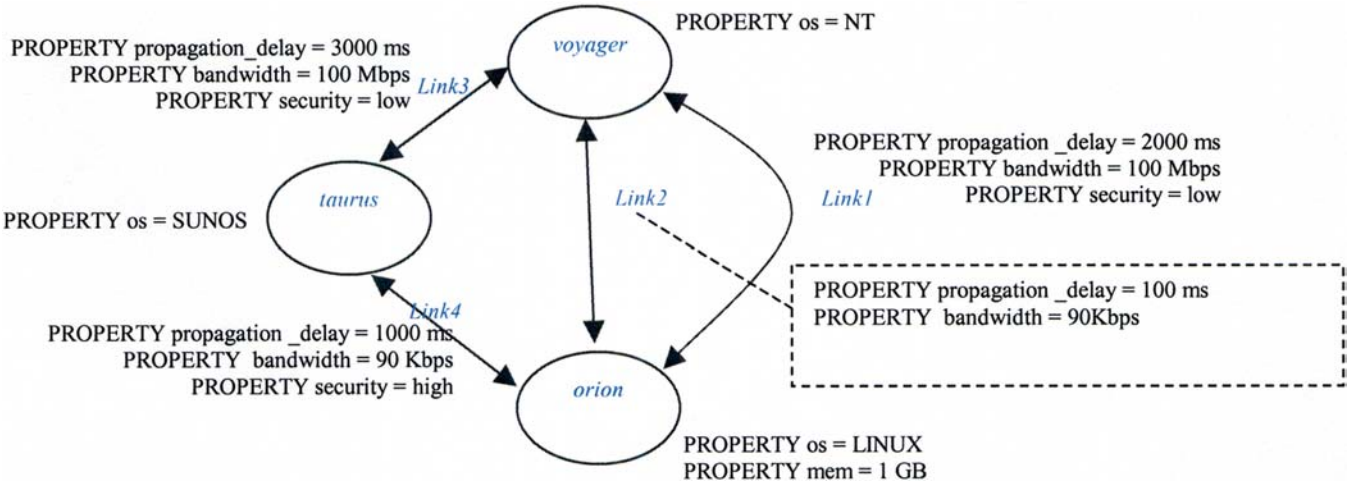

Fig. 2. The PSDL model for system network hardware

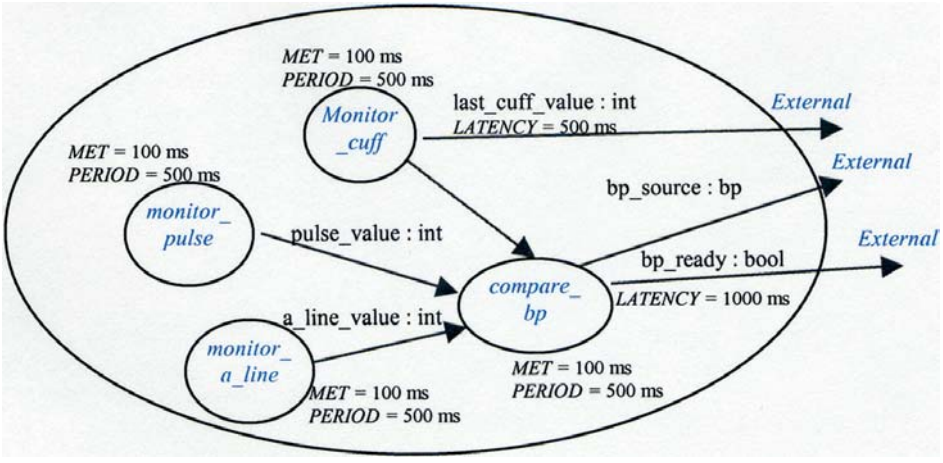

Fig. 3. Decomposition for the `monitor_bp` vertex

One can always represent a given static hierarchical computation graph by an equivalent basic static graph; however, care must be taken to ensure that such transformation does not introduce inconsistencies into the resultant constraints.

3 Tools for rapid automated prototyping of complex systems (CAPS-PC)

CAPS-PC is the result of our latest effort in a series of research on prototyping languages, real-time systems modeling and analysis, automatic code generation techniques, and rapid prototyping environments [7, 8, 20–23]. These tools provide a PC-based computer-aided environment to support the modeling, analysis, and prototyping of systems under development. The interface of the CAPS-PC is shown in Fig. 4.

The tool suite provides a system model editor for users to create and modify their system models defined by the PSDL prototyping language [19, 22], a translator to check the syntax/semantics of the system model and to generate glue and wrapper codes to realize the design for the target system architecture, and a scheduler to analyze the timing constraints and to generate code to realize these constraints in the target architecture [17]. The tool interface also provides menus for users to manage their projects and compile source code into an executable prototype [7, 8].

The knowledge mapping from informal natural language narration to formal language specification may result in some misunderstanding and incompleteness of the requirements documents. The graphical interface of the tool suite provides the user a simplified way of explicitly defining the model attributes to meet the requirements specifications.

The hierarchical design of models helps to organize the requirements specification in a way that can be tracked throughout the system's development according to abstraction level and responsible functionality. The clear and precise diagrams accompanying the documentation make it easy for a designer to check the consistency with the text. Each operator defined in the diagram refers to the requirement item number in narrative documents,

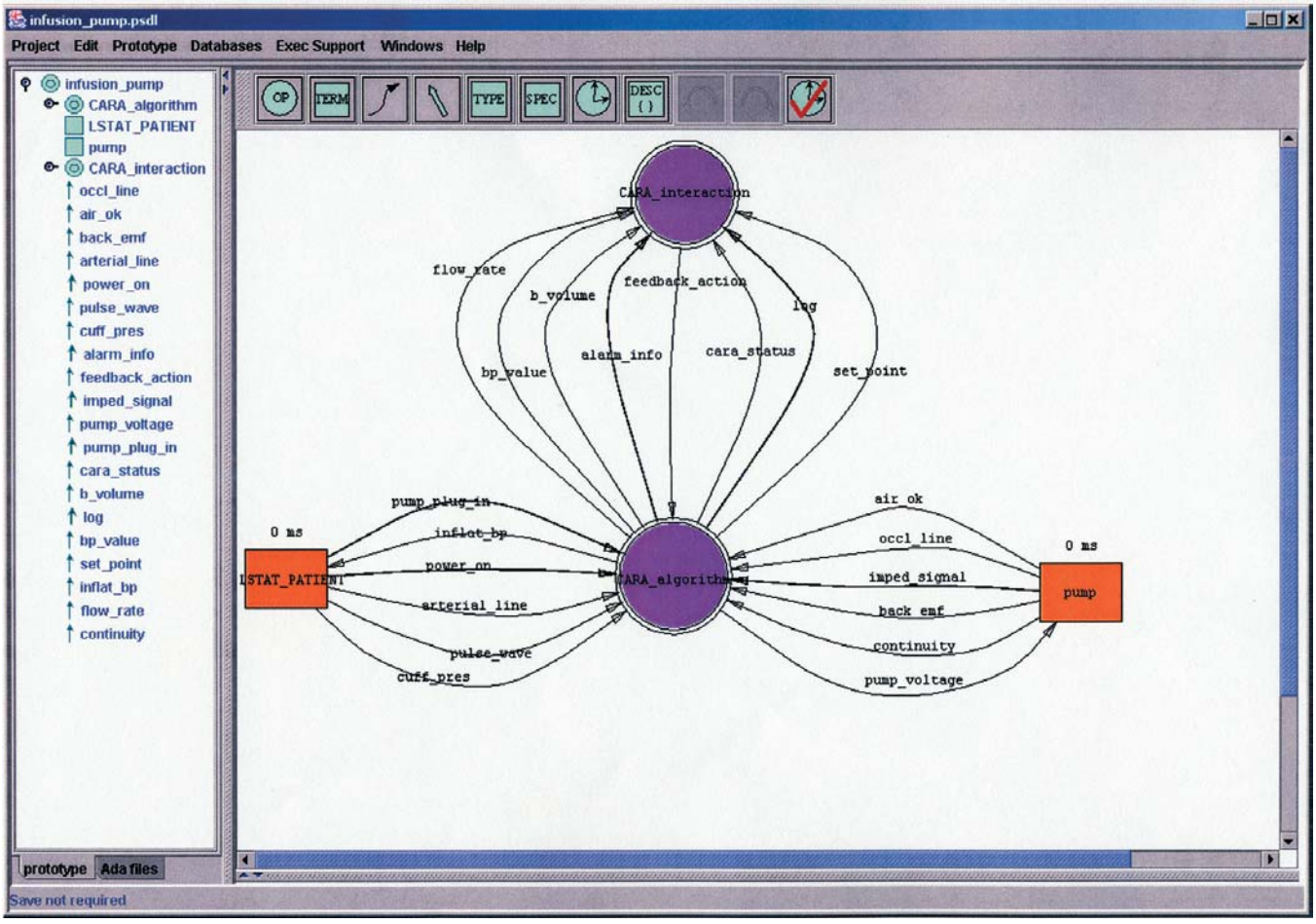

Fig. 4. Interface of the CAPS-PC environment

which makes it easy to find cross references in the whole design model.

Several supporting documents can be built by the designers or generated automatically by using CAPS-PC. As the central chain of development documentation, these documents drive the specification, design, implementation, and even testing of the system development. The consistency maintained by the tool suite between these documents provides a solid baseline throughout the development effort. The documentation generation function of the tool suite makes it easy for the customer, user, and sponsor to understand, handle, and review the system during development.

First, the specification of requirements can be generated with completeness and consistency checking, according to the system functionalities and constraints. The graphical design process maintains the syntax of the requirements specification, and a further translation process ensures the semantic consistency of the specification document.

Second, the narrative description of a designed model in natural language can be generated based on the definition of model functionalities and constraints, which tell the customers and users what the system will and will not do. For example, for CARA's periodic blood pressure corroboration, the model can generate the following narration: "if VALID_BP is CUFF_BP, and if MEAN_BP $>$ 90, the CUFF_BP will get data from CUFF_MONITOR every 10 minutes".

Third, by using technical terminology to describe the system's structure, data and function can be generated based on the model specification. Information about input, such as where data come from and how they are formatted; output, such as where data are sent and how they are formatted; general functional characteristics, such as periodic execution or sporadic execution; performance constraints, such as minimum execution time; and specific fault-handling approaches can be generated and described in the design documentation.

Fourth, during the generation of prototype code, the tool suite can generate partial implementation documentation based on the specification structure. The name, type, and purpose of major data structures and variables, simple description of logic flow, expected input, and possible output can be derived from the information defined in the model.

Furthermore, CAPS-PC can maintain the version control documentation for requirements specifications and

model design. If any changes are made to the requirements during the remaining phases of development, the changes can be tracked from the requirements document through the design process and all the way to the test procedures.

When we were designing the CARA software, the documentation generated or maintained by our tools greatly enhanced the effectiveness of the design procedure and the communication between the model designer, user, and reviewer. The specification documents were used by the tool suite to interlink the individual tools. The structured narrative descriptions of system design models are used for detailed discussions or for system reviews.

The use of our tool suite to prototype the CARA software based on its requirements documents showed that the development process and the communication between customer and the software developer can be improved by the integration of software development documentation and the enhanced information representation. Further research on key issues in software development driven by documentation, such as specification-based model consolidation, would make the development process even more efficient.

4 Parallel designs of the infusion pump control software

To find defects in the requirements, we formed five design teams and used PSDL and CAPS-PC to analyze the requirements of the Infusion Pump Computer Assisted Resuscitation Algorithm (CARA) software. The requirements for the CARA system total ten pages, with almost 70 main points listed. The descriptions of the requirements were listed logically based on the timing sequence of the infusion pump. For each of the points, the document gives a detailed description of event occurrence conditions and subsequent event effects. All of these requirements were depicted structurally and detailed in natural language.

In summary, the main objectives of the CARA software include:

(1) monitoring patients' blood pressure,
(2) controlling a high-output infusion pump for patient resuscitation,
(3) providing feedback on patient status to the physician,
(4) recording all information monitored by the system,
(5) activating an alarm for emergency situations.

All the requirements can be catalogued as two types – functional requirements and quality requirements. Functional requirements describe the expected action based on the satisfaction of identified conditions. The requirements documents also give some multicase descriptions for the conditional actions. Quality requirements describe the system properties constraints that will be satisfied during the execution of the action. The constraints include both control and timing constraints, some of which are safety constraints on the operation of the infusion pump. Control constraints tell the designer the required conditions or possibility of action occurrence. Timing constraints re-

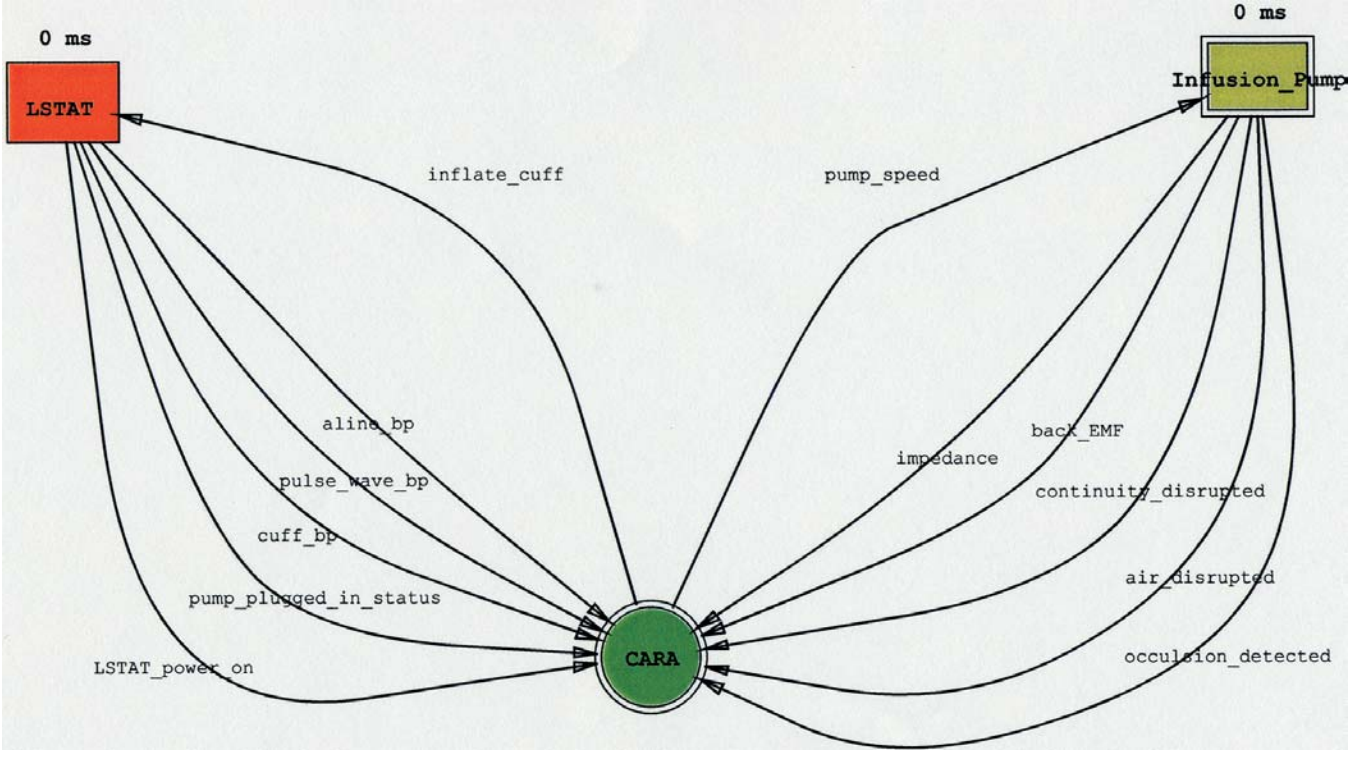

Fig. 5. Top level design of model 1

strict the time of the described event. These requirements can be treated also as case style descriptions.

To fully understand, analyze, and verify the requirements definition document, which contains a record of the requirements in the customer's terms, we set up five 2- to 3-person teams to prototype the CARA system according to their own interpretation of the requirements. The whole group held weekly meetings, highlighting and answering several questions during the intensive discussions among the five teams. The following subsections present the five design models for the LSTAT/CARA system.

All five design teams separated the CARA control subsystem from the physical data hardware sensors subsystem in their models. In each design, the control model can be decomposed into a blood pressure data component and an infusion pump control component. Some designs, such as models 3 and 4, specified the time constraints in more detail than others. Some designs, such as models 1 and 2, built extra quality assurance modules with detailed consideration of safety issues of the CARA system. Models 3 and 5 separated the control subsystem from the information subsystem. This allowed the designers to focus on issues specific to each subsystem. Models 4 and 5 incorporated several infrastructure tests to verify the completeness of the CARA requirements and the fault tolerance of the CARA system. They demonstrate the behavior of the system under simulated failure conditions.

We explored the effectiveness of parallel conceptualization efforts to expose potential requirements issues. Since the initial requirements document is inconsistent and inherently incomplete, each team made its own assumptions concerning requirement functions and system performance, which were embodied into their design models. The use of multiple independent teams to perform the inspection of the requirements revealed more requirements faults. Our parallel prototyping process demonstrated the effectiveness and efficiency of comparison and discussion of different designs and viewpoints in finding and fixing faults in the requirements specifications.

4.1 Design model 1

In its most abstract form, model 1 consists of just three main components: The LSTAT stretcher 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, as indicated in Fig. 5.

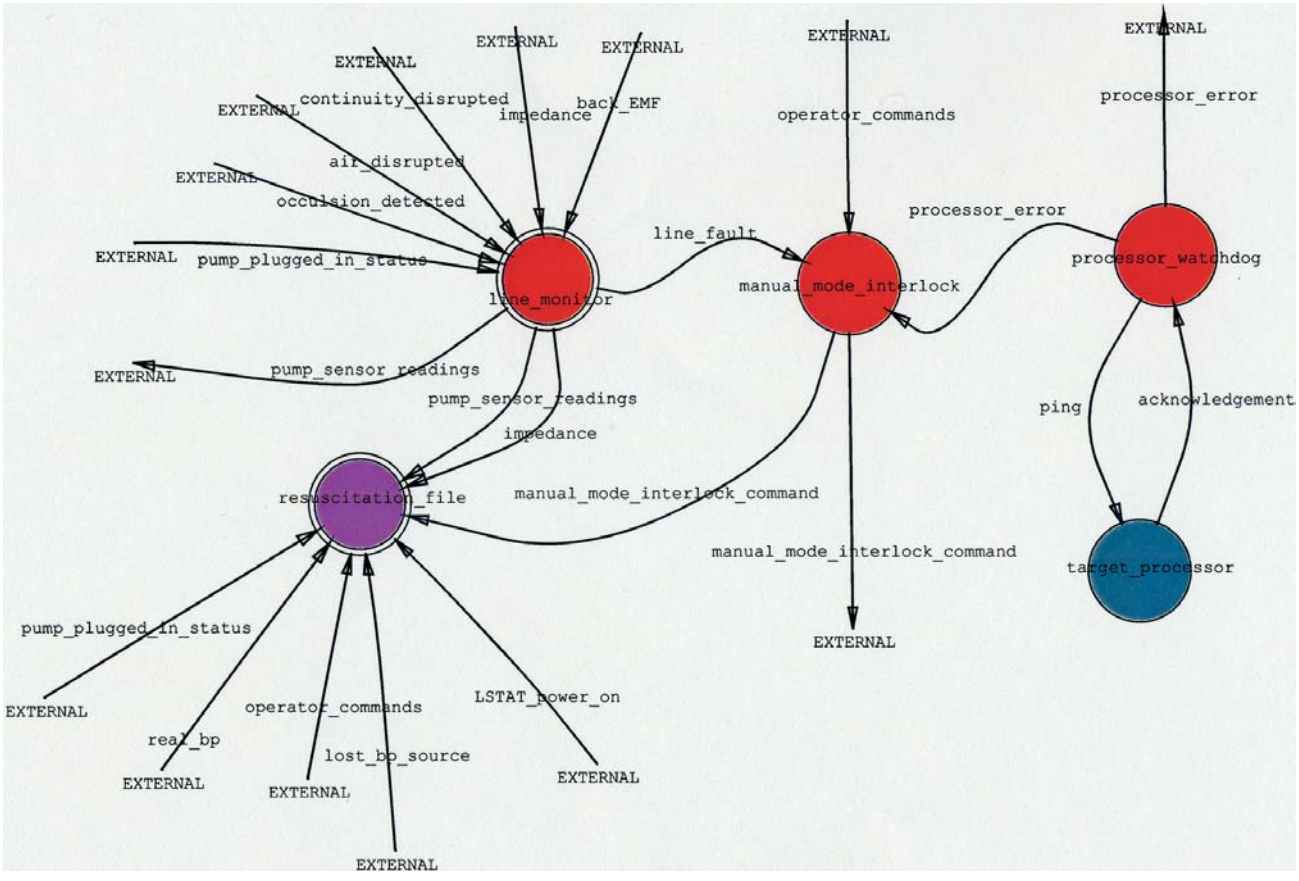

Fig. 6. Management module of model 1

This design model presents the most comprehensive consideration of the safety issues and deals directly with the potential confusion in control modes between manual control and autocontrol and the potential confusion in acquiring blood pressure sensor data.

The `manual_mode_interlock` operator in the decomposition of the `Management_module` shown in Fig. 6 is responsible for returning the system to a manual mode in case of any component failure. Feeding the `manual_mode_interlock` is a processor watchdog implemented on a separate processor. The Watchdog will sense any failure of the main processor and alert the operator via the display.

Figure 7 shows the `pump_control` module, which is the major safety critical module in the design. It is responsible for resolving an accurate blood pressure reading from the various input sources and determining the correct input to the pump when the system is in autocontrol mode. This process collects and resolves three types of blood pressure data: cuff, arterial line, and pulse wave. Once resolved, the correct blood pressure is used to calculate the pump drive voltage. To overcome the possibility of data confusion and ensure continuous voltage calculation, the team chose triple modular redundancy (TMR) to design the pump control module. The requirements document did not specifically call for this safety architecture; however, the safety-critical nature of the CARA 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. Such architecture is designed to address potential software faults. If implemented on separate hardware, the architecture could address some hardware faults as well.

4.2 Design model 2

Considering the different sources of blood pressure information, the top level design of model 2 includes the CARA, pump, LSTAT, patient, and `pressure_gauge`, as shown in Fig. 8. The `pressure_gauge` module in Fig. 9 consists of three atomic operators, each responsible for monitoring one of the three blood pressure sources. For each blood pressure source, the `gauge_impulse` signal is sent back to equipment that is attached to the patient to trigger additional data collection. The model embodies control of the blood pressure cuff inflation as `inflate_control`. Once the `pressure_gauge` module gets the signal from CARA, it starts the process of in-

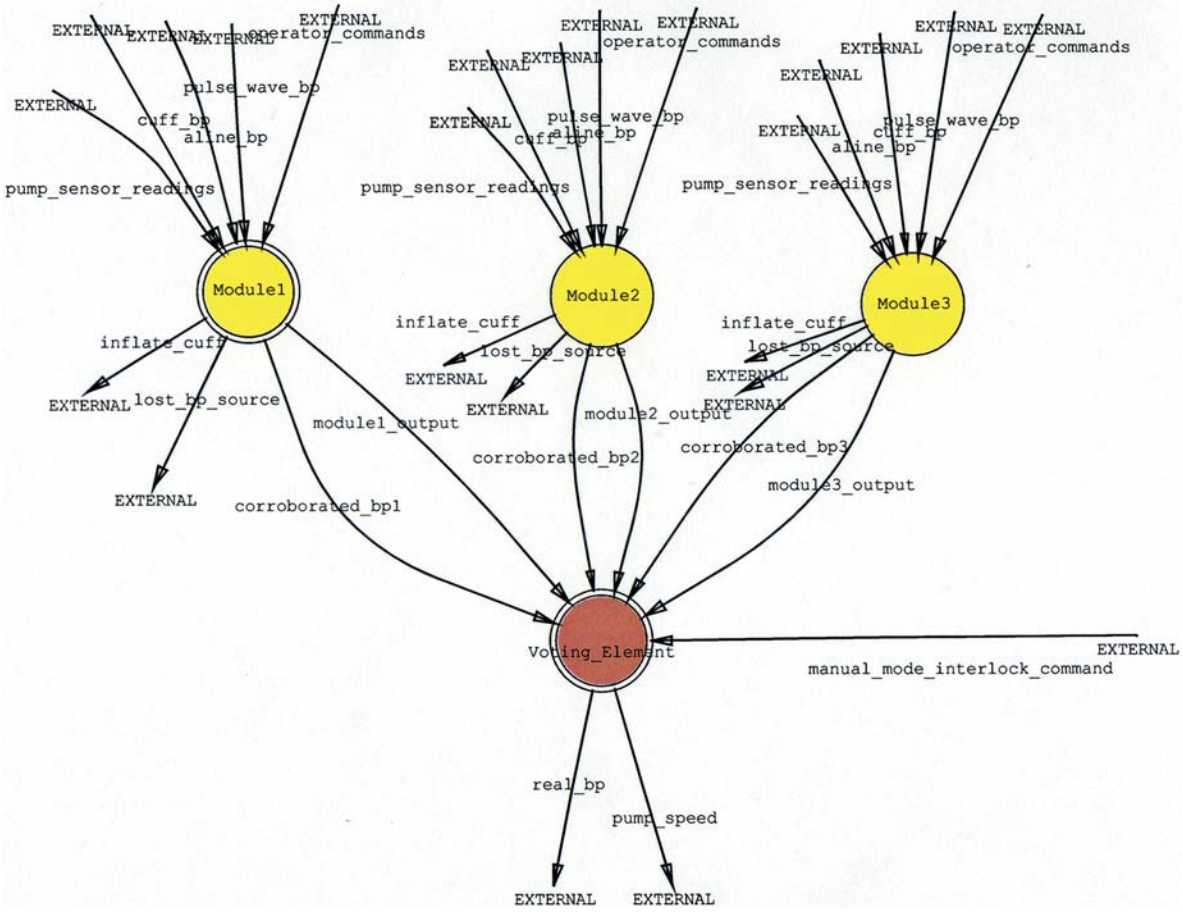

Fig. 7. The `pump_control` module of model 1

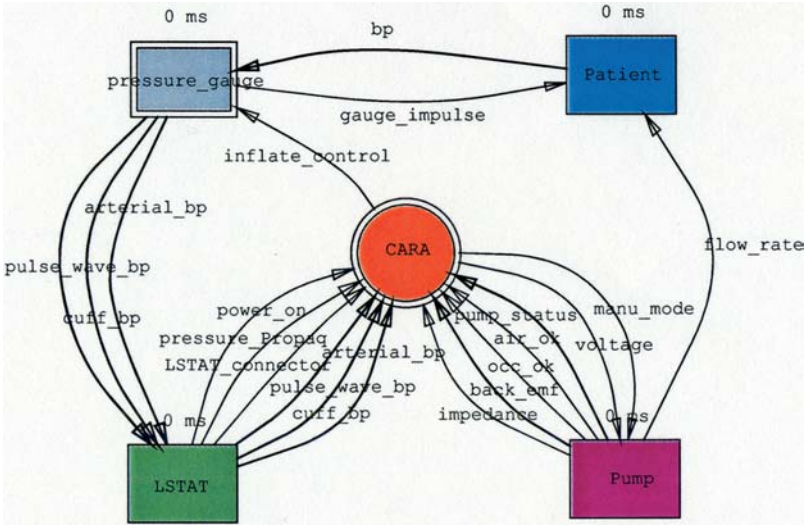

Fig. 8. Top level design of model 2

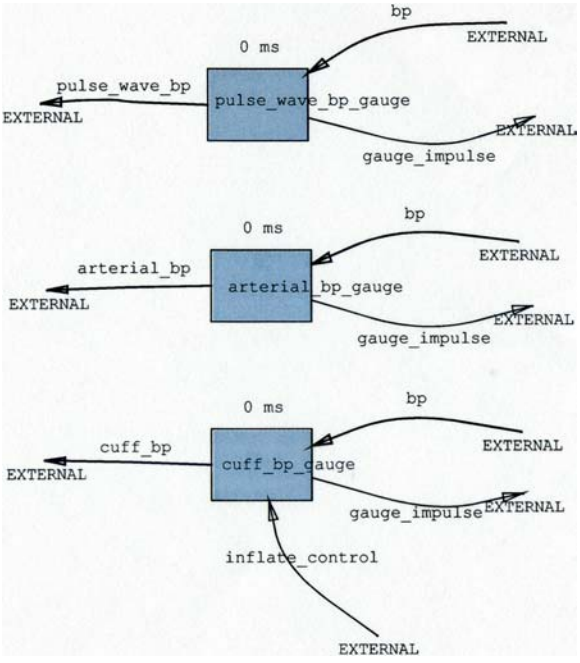

Fig. 9. The `Pressure_Gauge` module of model 2

flating the cuff to obtain the patient's blood pressure, as shown in Figs. 8 and 9.

The CARA module is the central part of the entire system. It can be subdivided into a monitor module, a control algorithm module, a display and alarm process module, and a log module. The monitor module monitors the signal from the pump and LSTAT to provide the CARA with pump, LSTAT, and blood pressure information.

The control algorithm module is the main part of CARA. It is responsible for calculating the flow rate and the cumulative volume of the pump according to the signal of `Back_EMF` (electromotive force) and providing the voltage to control the pump flow rate. The algorithm also performs several other functions on the monitored signals. The detailed process is included in the two lower layers. The display and alarm process module design in Fig. 10 can process an alarm and display a corresponding message based on the input data stream. Separating the information display and log record helps generate a well-structured rescue history and an easily retrievable data repository.

4.3 Design model 3

In model 3, the CARA software is split into two main subsystems, which work together with the `LSTAT_patient` and Pump, as shown in Fig. 11. CARA is not only an embedded control system but also an information system. These two essential characteristics result in the separation and encapsulation of blood pressure rescue control and information communication. The `CARA_algorithm` operator encapsulates most of the operational control and computation of blood pressure data and pump drive voltage. The `CARA_interaction` operator is responsible for information display control, alarm signal management, resuscitation file recording, and operator override handling. This isolation of information subsystem and control subsystem provides a good underlying condition for the designer to focus on different issues for different subsystems. Designing and prototyping the `CARA_algorithm` requires attention to source data control and time constraints definition. Comparatively more effort is spent on information display, completeness checking, and the interface design for the information representation subsystem.

When calculating the infusion pump drive voltage, selection of an appropriate source of blood pressure data for corroboration is one of the most important aspects of the CARA design. Instead of using three redundancy

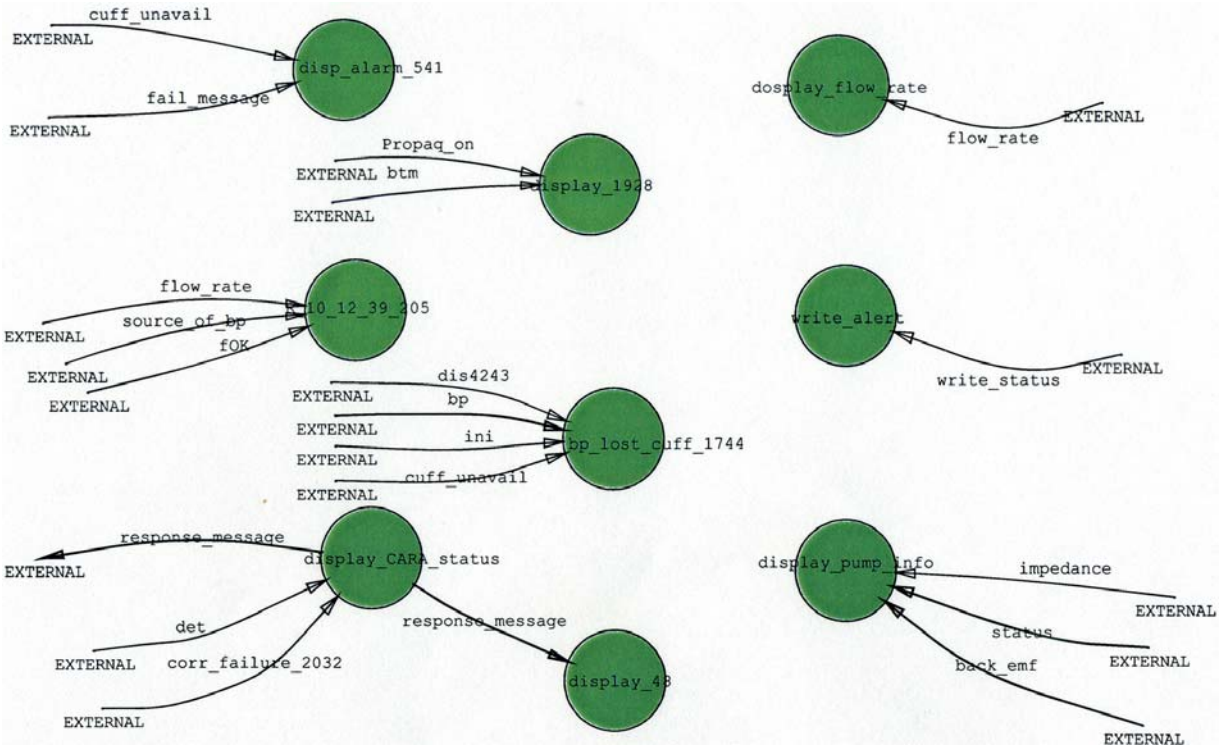

Fig. 10. Display and Alarm Process module of model 2

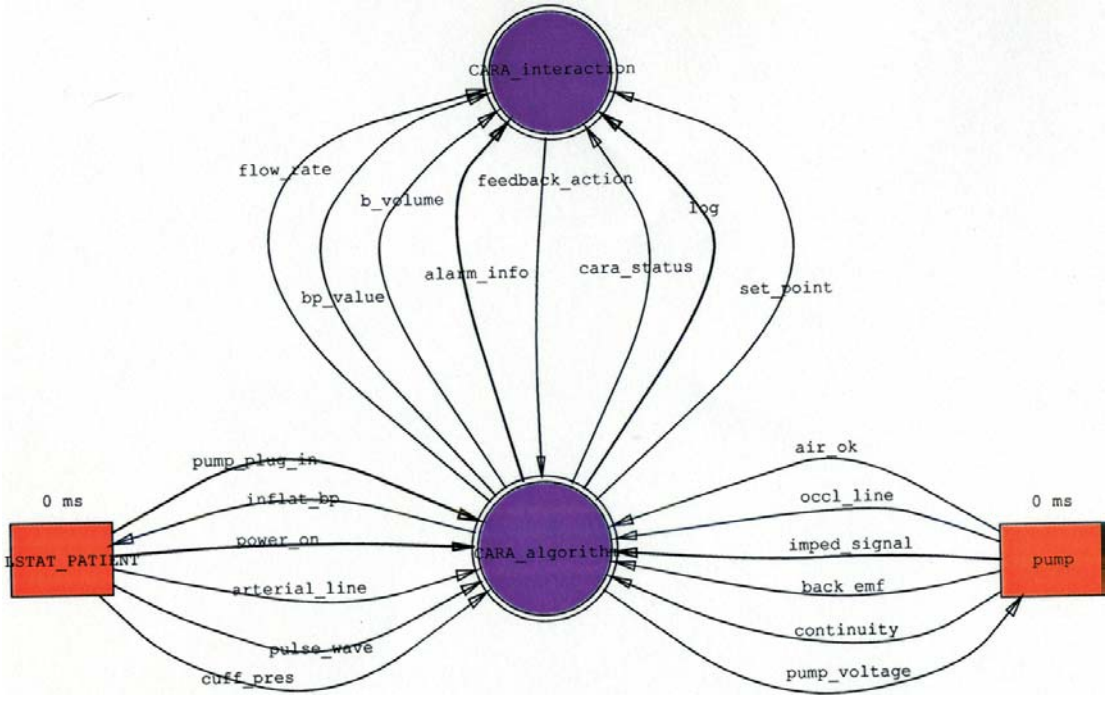

Fig. 11. Top level design of model 3

modules to calculate the data, this design model employs two preprocess modules to validate and rank the priority of the blood pressure. The first preprocess module acts as the valve for evaluation of the original blood pressure and ensuring that the valid source of blood pressure data is released for further computation. The second selects the highest priority blood pressure by considering continuity, validity, and stability of the source data, as shown in Fig. 13. This preprioritization of blood pressure data ensures the validation of the calculated pump drive voltage by controlling the validation of the source data.

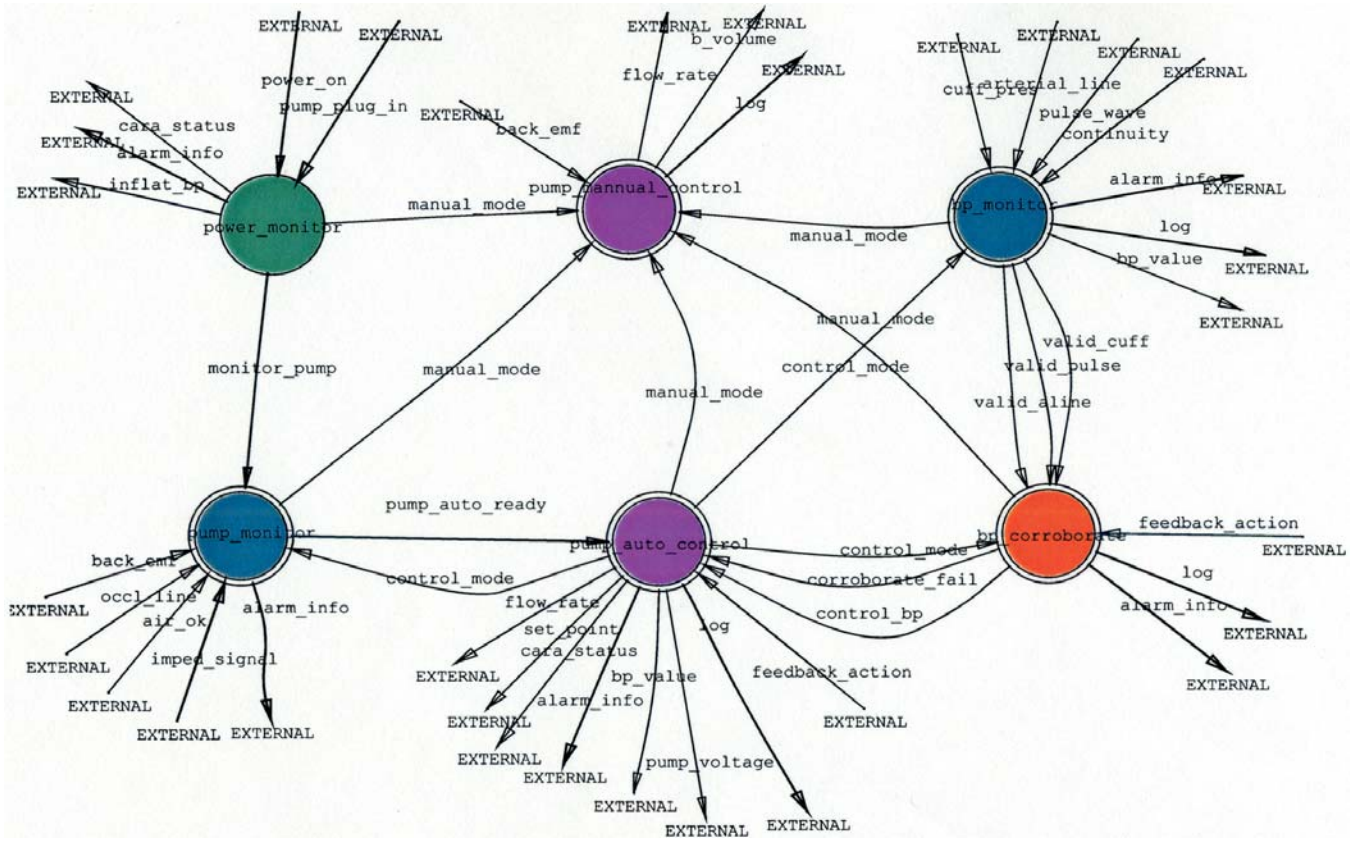

Fig. 12. The `CARA_Algorithm` module of model 3

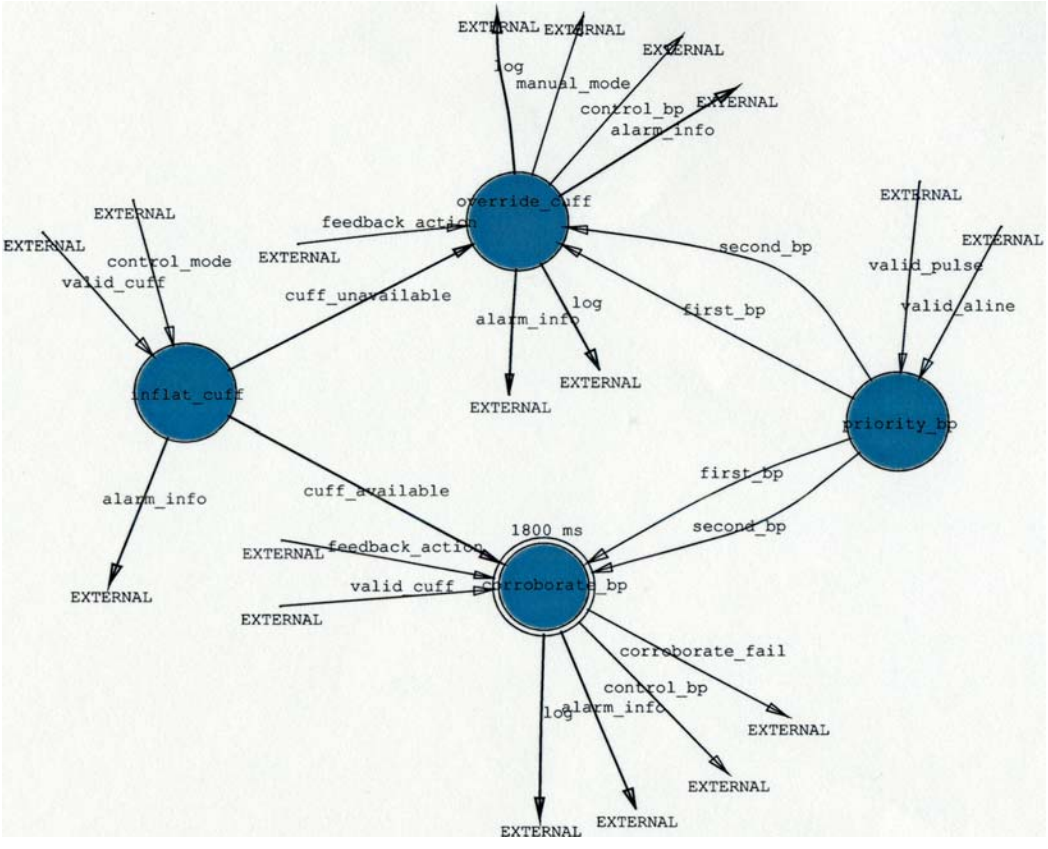

Fig. 13. The `bp_corroborate` module of model 3

4.4 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, as shown in Fig. 14. The `test_instrumentation` module provides a GUI (graphical user interface) for the user to control the status of the pump, patient, and LSTAT simulations during prototype execution.

The purpose of including `test_instrumentation` is to account for the possible failure of subcomponents. This infrastructure provides an additional verification capability based on prototype execution to demonstrate the behavior of the system under conditions where some of the subcomponents fail. The `test_instrumentation` module artificially induces failures for test purposes. For example, it can generate the exception status of LSTAT to test the emergency alarms of the CARA `manage_alarm` module. The `test_instrumentation` module is used during the prototyping period only. After finishing the prototype model and verifying the safety constraints, this module will be isolated from the generated evolutionary prototype. During actual system operation, the real source data collected from LSTAT, pump, and blood pressure will be used to feed the control signal to the CARA system.

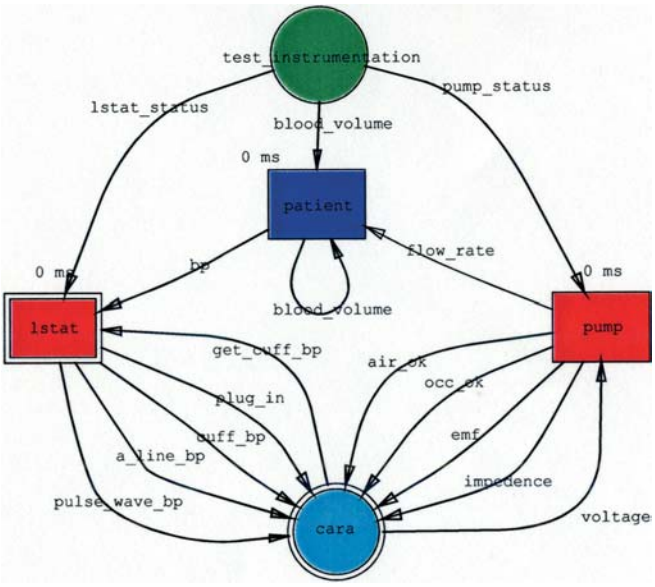

Fig. 14. Top level design of model 4

Central to model 4 is the CARA module, which is decomposed into the six submodules shown in Fig. 15. The `monitor_bp` and `monitor_pump` modules monitor and validate blood pressures from the LSTAT and moni-

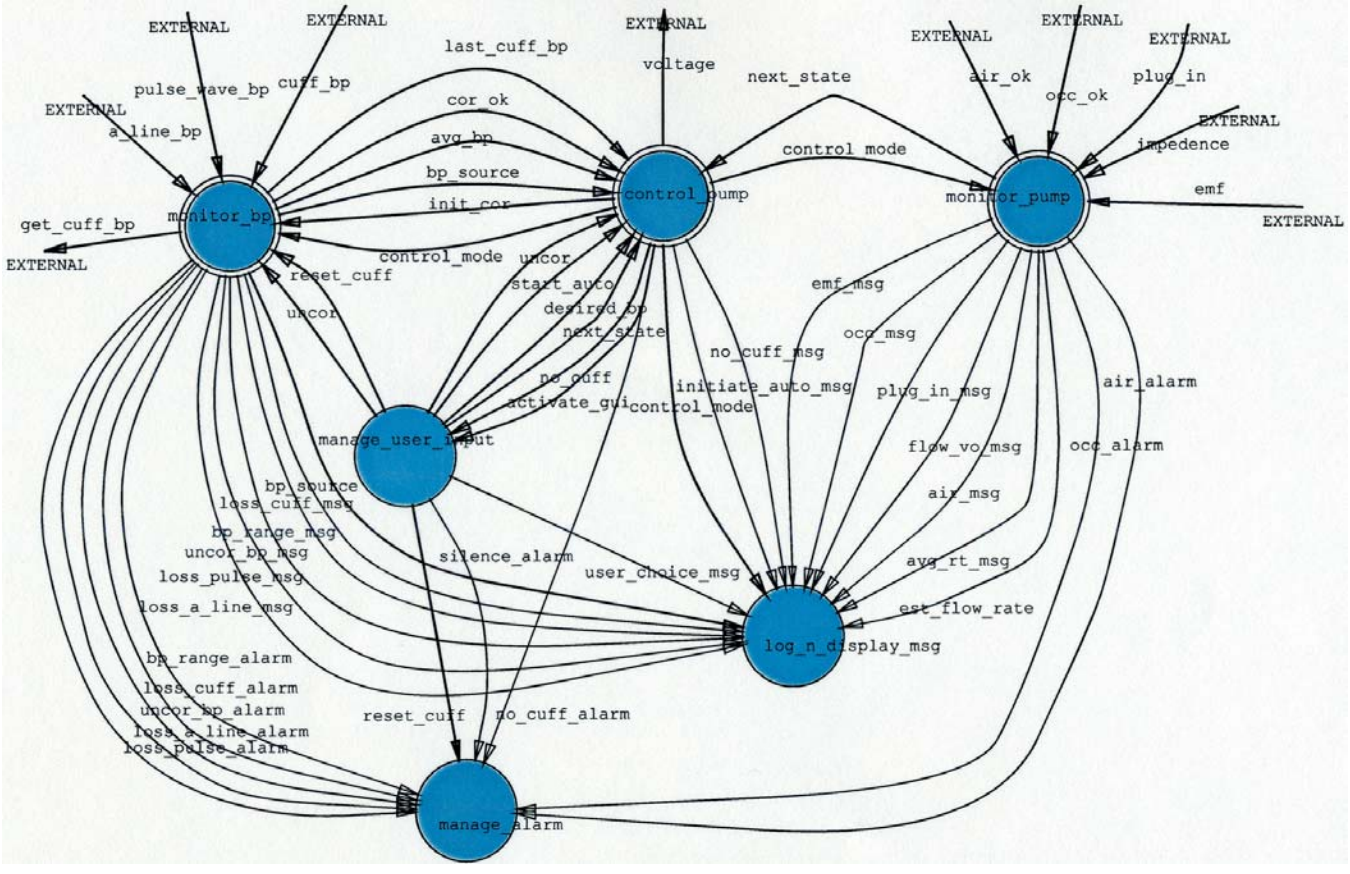

Fig. 15. CARA module of model 4

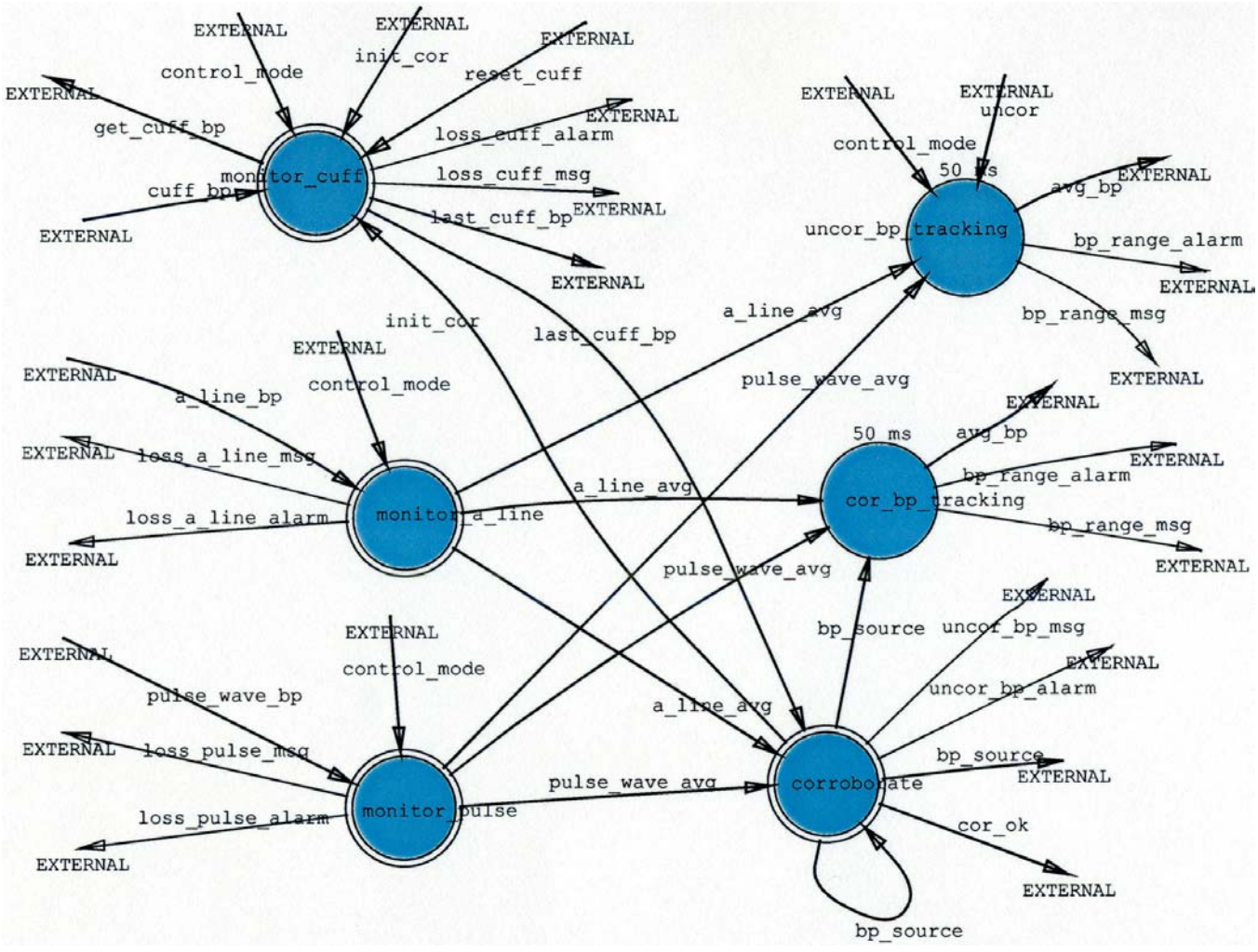

Fig. 16. The Monitor_bp module of model 4

tor signals from the infusion pump, respectively. Outputs from these modules feed the `control_pump` module to determine the infusion pump drive voltage. They also feed data to the `manage_alarm` and `log_n_display_msg` to alert the user as needed. The `manage_user_input` module processes inputs from the users and sends out the resultant events for further processing.

Figure 16 shows the internal composition of the `monitor_bp` module, which monitors the three different blood pressure sources (cuff, arterial line, and pulse wave). This module tracks the blood pressures when the CARA control software is in manual mode and performs blood pressure corroboration when the CARA control software is in the autocontrol mode. It also alerts the user to any disruption in the blood pressure sources and signals the `control_pump` module to switch the CARA control software from autocontrol mode back to manual mode if necessary.

Much like the `monitor_bp` module, the functions of the `monitor_pump` module in Fig. 17 are implemented by five submodules: `monitor_plugin`, `monitor_occ`, `monitor_air`, `monitor_impedence`, and `monitor_emf`. Outputs from these modules are used by the `decide_next_state` module to determine if the system is stable enough to switch into autocontrol mode or if the system is so unstable that it has to switch back to or remain in manual mode. The `control_pump` module is then signaled to act accordingly.

4.5 Design model 5

This model consists of several cohesive subsystems that are created based on a separation of concerns. The top layer of the model consists of the CARA software and four external subsystems: simulated patient, LSTAT, pump, and alarm, as shown in Fig. 18.

Figure 19 shows the internal composition of the CARA software, which consists of six submodules: `BP_monitor`, `Safety_monitor`, `Pump_monitor`, `Pump_controller`, `Logger`, and `Display`. The `Pump_controller` module includes a `Manual_pump_controller` and a `Software_pump_controller`, as shown in Fig. 20.

This model explores in more detail than do the other designs the requirements related to returning to the man-

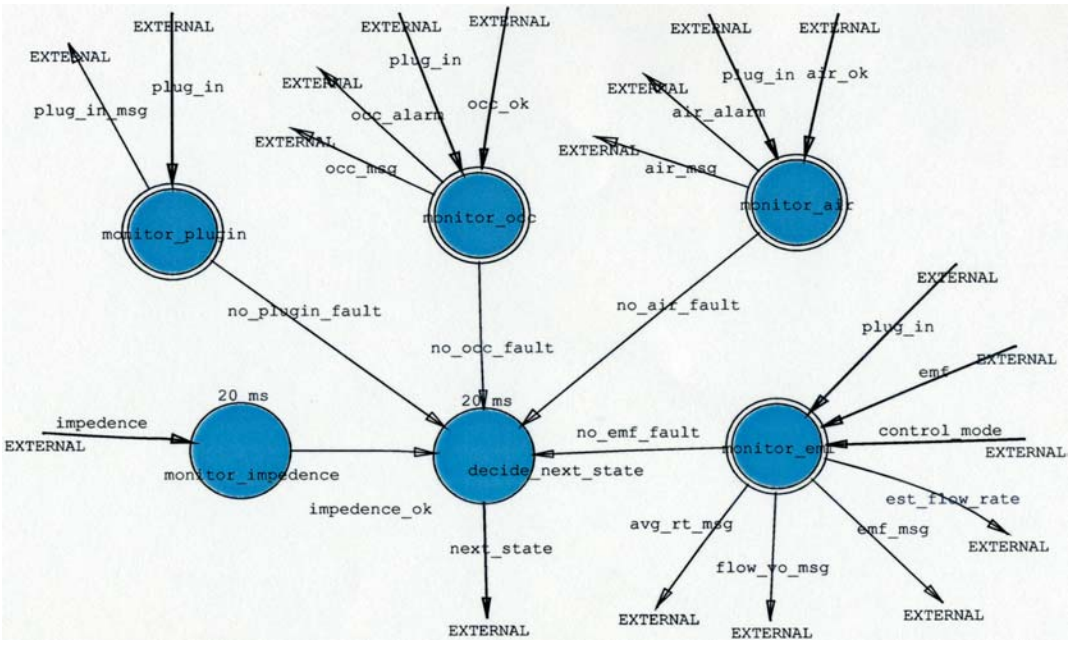

Fig. 17. The Monitor_pump module of model 4

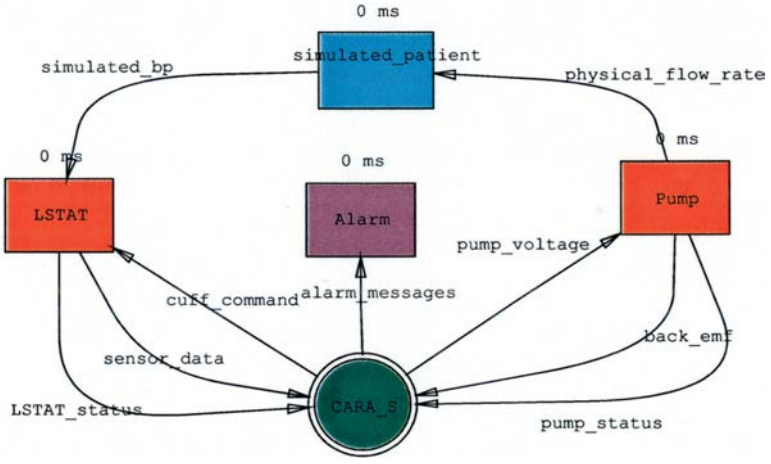

Fig. 18. Top level design of model 5

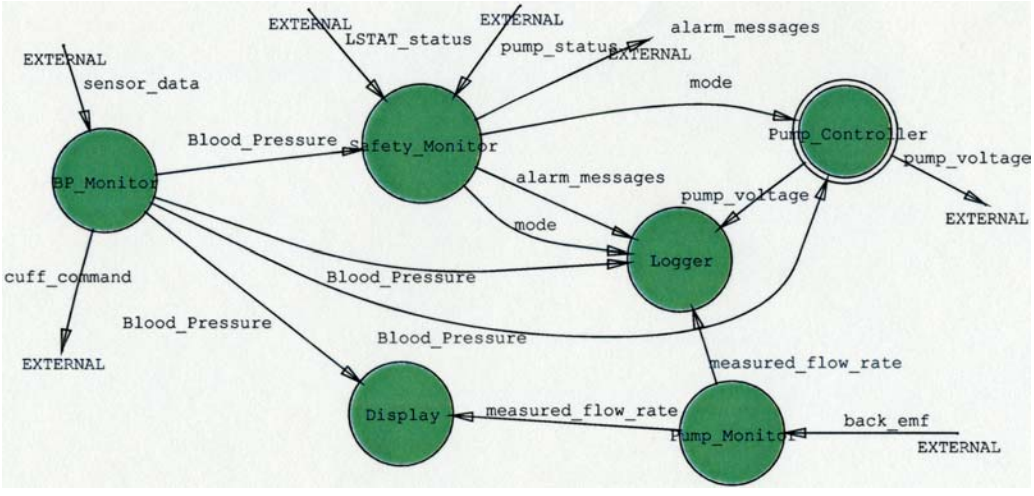

Fig. 19. CARA module of model 5

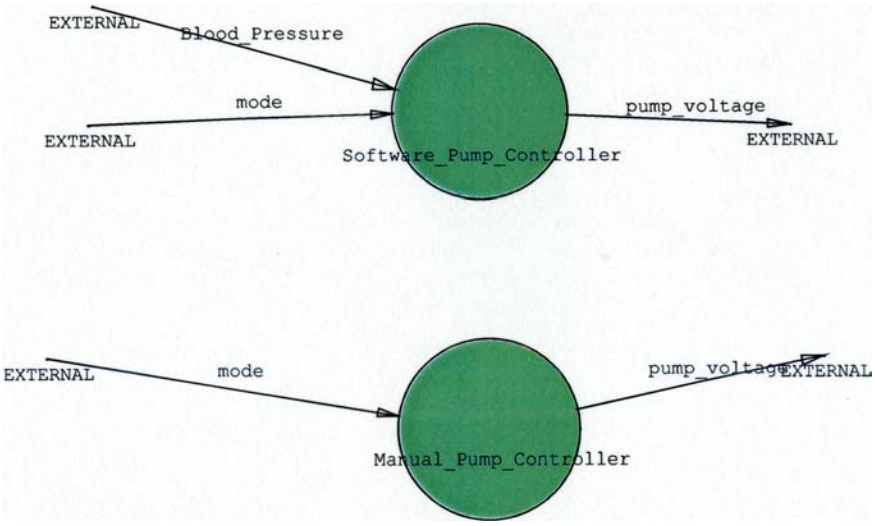

Fig. 20. The Pump_controller module of model 5

ual control mode when the safety of the automatic control is compromised. This design identifies the need for additional external interaction when unsafe conditions arise. When the patient's well-being is in jeopardy, alarms will alert human operators. The simulated patient tests the effectiveness of proposed pump control policies. Combining this design with the test instrumentation structure of model 4 would more effectively check the transitions between automatic control and manual control and back again, particularly with respect to safety issues and possible missing requirements.

5 Features of the design models

Using a suite of prototyping tools, five teams generated alternative design models based on the analysis of customer requirements documents. Because the experiments were conducted by five different teams independently, the five models had different features. The different features can be divided into four main categories: requirements coverage, simplicity of design, enabling architecture, and safety aspects. Analyzing features of the five models helped us uncover the defects of the requirements from different perspectives.

5.1 Requirements coverage

All of the models cover most of the stated requirements in the broad sense, but they vary in detail when it comes to capturing the logic of the procedures stated in the requirements. All of the models identify the major functions of the CARA software and group them into different modules. With the exception of model 5, each model covers $\sim 90\%$ of the high-level requirements and at least half of the detailed ones.

5.2 Simplicity of design

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 safety of the design. Models 2, 3, and 4 suffer from varying degrees of complexity at the lower levels of granularity.

Models 3 and 5 also attempt to segregate the safety-critical functions from the non-safety-critical ones in their design, but both need more work to complete the design. Model 2 attempts to divide the detailed activities of the control algorithms into six concurrent processes but fails to capture the event/response relationship among these processes. 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, the `pump_control` module models a state machine consisting of three states {`auto_off_state`, `auto_ready_state, and auto_on_state`}. Depending on the value of the `control_mode` state stream, one of the three processes in the `pump_control` module will be active at all times. Human input will trigger the `get_initial_cuff_bp` operator to request that the `monitor_bp` module corroborate the different blood pressure sources. The other four operators will remain idle whil waiting for the results from the `monitor_bp` module. When triggered by the arrival of the results, these four operators will decide whether it is safe to switch into `auto_control_state` and change the value of the `control_mode` state stream accordingly.

Simplicity of 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 to move the functions and form weaker coupling modules. For example, model 4 can greatly simplify the top layer of the CARA software shown in Fig. 15 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 signal sources and priorities and then process them accordingly significantly simplifies the alarm data stream design.

5.3 Understandability of the model architecture

Understandability of the model architecture was a common goal of all the designs. The design teams attempted to accomplish this goal via decomposition based on separation of concerns. This approach resulted in very similar, but nevertheless different, top level designs. The top level designs reflect two different assumptions of the intended prototype. Models 1 and 3, as indicated by Figs. 5 and 11, model the prototypes as open systems, where the pump and the LSTAT modules represent interfaces to the actual hardware. Models 2, 4, and 5, as displayed by Figs. 8, 14, and 18, model the prototypes as closed systems. These designs 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 runtime testing.

In an effort to decompose the CARA system, each design team identified the following functions of the CARA software – monitoring pump, monitoring blood pressure, controlling pump, logging and displaying messages, and managing user input. However, the five design teams varied in how they presented these functions in the hierarchical decompositions. Model 1 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 groups all monitoring functions, such as monitoring pump and monitoring blood pressure, into a single `monitor` module and buries the user input management in the `control_alg` module. On the other hand, model 3, as shown in Fig. 12, separates the message display and user input management from the CARA control software and places them in a `CARA_interaction` module in the top level of the prototype. Models 4 and 5, as shown in Figs. 15 and 19 respectively, have very similar designs. They both lay out all six functions explicitly in the top layer of the CARA software.

5.4 Safety aspects of the design

CARA is a software system that provides closed-loop control to a high-output infusion pump [6, 15]. It is important to quantitatively define the safety issues in its requirements documents. However, the design requirements, even with the additional questions and answers document [30] provided, are very incomplete, especially in the system safety aspect. The requirements documents do not provide any performance requirements – they provide primarily design requirements. In particular, the requirements documents do not identify or prioritize the functions, especially from a safety perspective. As a result, some models, such as models 2 and 4, do not differentiate between safety-critical and non-safety-critical functions.

The segregation of the identified safety-critical functions from non-safety-critical functions greatly enhances the design of models 1 and 5. In addition to those specified in the requirements documents, 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 in Fig. 7. This TMR architecture relies on three different algorithms for calculating the infusion rate when in autocontrol mode and a single voting mechanism to determine which rate to pass to the pump. Second, it implements a processor watchdog function on a separate processor to alert the operator in cases of main processor failure, as shown in Fig. 5. The redundant architecture on the blood pressure corroboration promises to substantially reduce the potential for a faulty monitor to drive the infusion pump.

Model 2 provides for ready identification of safety-related functions and data; however, the design does not segregate either from the non-safety-related functions and data. Therefore, the safety assessment and verification of this design is more difficult than the assessment of model 1 or 5.

The complexity of model 3 makes it the hardest to assess from a safety perspective. Most of the objects in the design have safety-related functions or information. Therefore, the design will require analysis and testing of a majority of the software. The preprocess modules of model 3 are safety critical in that they provide all of the safety-critical information used to control CARA.

The `test_instrumentation` module incorporated in model 4 provides a valuable tool for verifying safety-related functions in the design. In the current implementation, the design incorporates limited testing of safety-critical signals and functions; however, the design could expand to address additional safety-critical aspects. Like model 2, the complexity of the software and the lack of segregation of safety-related processing from non-safety-related processing make safety assessment and verification difficult.

The safety_monitor module in model 5 uses a different system architecture to achieve the necessary level of risk mitigation in the design. The use of this architecture can reduce the overall effort required to perform safety assessment; however, it requires that the safety_monitor

module be robust and highly reliable from a safety perspective and that it be capable of terminating other functions in the design when it detects unsafe conditions. Its design will require identification of all safety-related functions and data and potential failure modes that could pose a risk to the patient.

6 Elicitation and evaluation of the requirements document

6.1 Requirements inspection and evaluation

In our prototyping experiments, we tried to analyze the user requirements document and find the omissions, inconsistencies, ambiguities, and contradictions. Similar to the N-Fold inspection method, which uses N independent teams to conduct the requirements inspection [11, 24, 26], we utilized five independent teams to design the prototype for the CARA system and explored the effectiveness of parallel conceptualization efforts to expose potential requirements issues. The comparison and combination of the prototyping feedback revealed many omissions and discrepancies in the requirements documents. Most of them emerged during discussions of the different design models. Because of the inherent incompleteness and incorrectness of the requirements document, each group had its own assumptions about requirement functions, which they incorporated into their design models. For example, when considering how the `Back_EMF` affects the pump rate, requirements 11 and 12 state that the `Back_EMF` computation applies to the manual mode of the infusion pump. Some design models, such as models 3 and 4, asked whether the same computation logic applied to the automatic control mode or whether `Back_EMF` would affect pump rate in auto mode. The requirements documents do not answer these questions. Our discussions of design models and reunderstanding of requirements documents helped find and fix faults in the requirements documents.

In addition to the previously noted lack of performance requirements, the following list of omissions and discrepancies in the requirements documents resulted from our evaluation of the CARA system.

(1) Requirement 6 should explain how continuous "continuously" is: What is the maximum response time (MRT) for the system to detect and handle a discontinuity event?
(2) Similarly, there is no statement on how frequently the occlusion lines need to be monitored to satisfy requirement 7.
(3) Neither the `Back_EMF` units nor the algorithm for converting `Back_EMF` to pump rate is given in requirement 10.
(4) Do requirements 11 and 12 concerning `Back_EMF` computation apply only to the manual control mode or to both the manual and automatic control modes? Is `Back_EMF` irrelevant in automatic mode?
(5) In requirement 16, the term "impedance" is not defined. Some designers understood the term impedance as an electronics term instead of a fluid mechanics term. That type of confusion could result in an implementation that could lead to hazardous conditions for the patient under resuscitation.
(6) It is unclear from the requirements document what policies to use for control of the cuff blood pressure. Is it always under CARA's control, or is a hardware module in charge of taking the cuff blood pressure periodically? If the cuff is under CARA's control, then how long will it take to inflate the cuff to get a new blood pressure reading, and how long does CARA have to wait before it can inflate the cuff and take the next blood pressure reading?
(7) Requirement 20.8 does not define the maximum response time to perform the corroboration after the arrival of a higher priority blood pressure reading.
(8) While requirement 44 calls for the CARA software to check the validity of the cuff pressure, the requirements document does not provide any information regarding the valid range of blood pressures from different sources.
(9) There is no requirement related to what happens when fluid gets low. Shouldn't some kind of alarm go off? How does the pump sense low fluid level?
(10) There is no requirement for redundancy, with the exception of redundancy implied by blood pressure measurements. We modeled a TMR (triple modular redundancy) architecture in the CARA algorithm in model 1.
(11) A key insight from the requirements analysis is that the feedback available from the system enables many capabilities in CARA that were not included in the requirements. Specifically, the various forms of feedback would allow CARA to perform diagnostics of various components attached to the system on a real-time basis. While such capabilities may not be part of the intent of CARA, they would substantially improve its safety and functionality. An example is the `Back_EMF` from the infusion pump. The requirements simply state that CARA should monitor the `Back_EMF` to determine the flow rate. Coupled with the measurements of the impedance (resistance to flow of the infused solution), CARA can calculate the amount of fluid provided a patient as a function of flow rate and time. However, the `Back_EMF` provides additional information. A low value indicates that there is little resistance to pumping the fluid, which may indicate that the IV line is not inserted (i.e., pumping free) or there is no IV fluid. Conversely, a high `Back_EMF` with a constant impedance may indicate occlusion before the occlusion signal occurs, or it may indicate a failing pump.

6.2 Integration of the five design alternatives

Not only did the five parallel prototyping efforts reveal the defects of the requirements document, but the integration of the five design alternatives helped to find emergent supplements for the requirements and provide a more complete design model. The five design teams identified different requirements discrepancies. For example, one of the most important discrepancies was the lack of reference to safety constraints. The requirements documents also do not discuss exception handling mechanisms, which required the designers to make assumptions to complete the system design. Models 1, 4, and 5 characterize these assumptions.

The difference in design focus among the five teams results in a better coverage of the functionalities of the requirements document. The final postdesign discussion with the whole group unanimously concluded that a comparatively more complete, effective, and efficient design model of the CARA system would result if we:

(1) fully considered the separation of the CARA control module and the physical data source modules,
(2) explicitly specified the time constraints (as done in models 3 and 4),
(3) built an extra quality assurance module with detailed consideration of the safety issues associated with the CARA system (as done in models 1 and 2),
(4) built in prototype testing infrastructure to verify the completeness and consistency of the CARA requirements documents (as done in models 4 and 5).

Also, the group suggested that, following a prototyping effort such as ours, the requirements documentation be refined to include explicit system constraints identified and defined by this type of effort.

7 Evaluation of PSDL and CAPS-PC

This experiment shows that PSDL can effectively model complex embedded software. PSDL's triggering guards and execution guards provide a convenient means for users to specify state machines explicitly without being concerned with target code. The timer feature is useful in modeling complicated timing policies. For example, Fig. 21 shows a simple model with two operators and four timers for the policy that

"When the cuff pressure is being used for control:

if the mean blood pressure is 60 or below, cuff pressures will be taken once per minute;

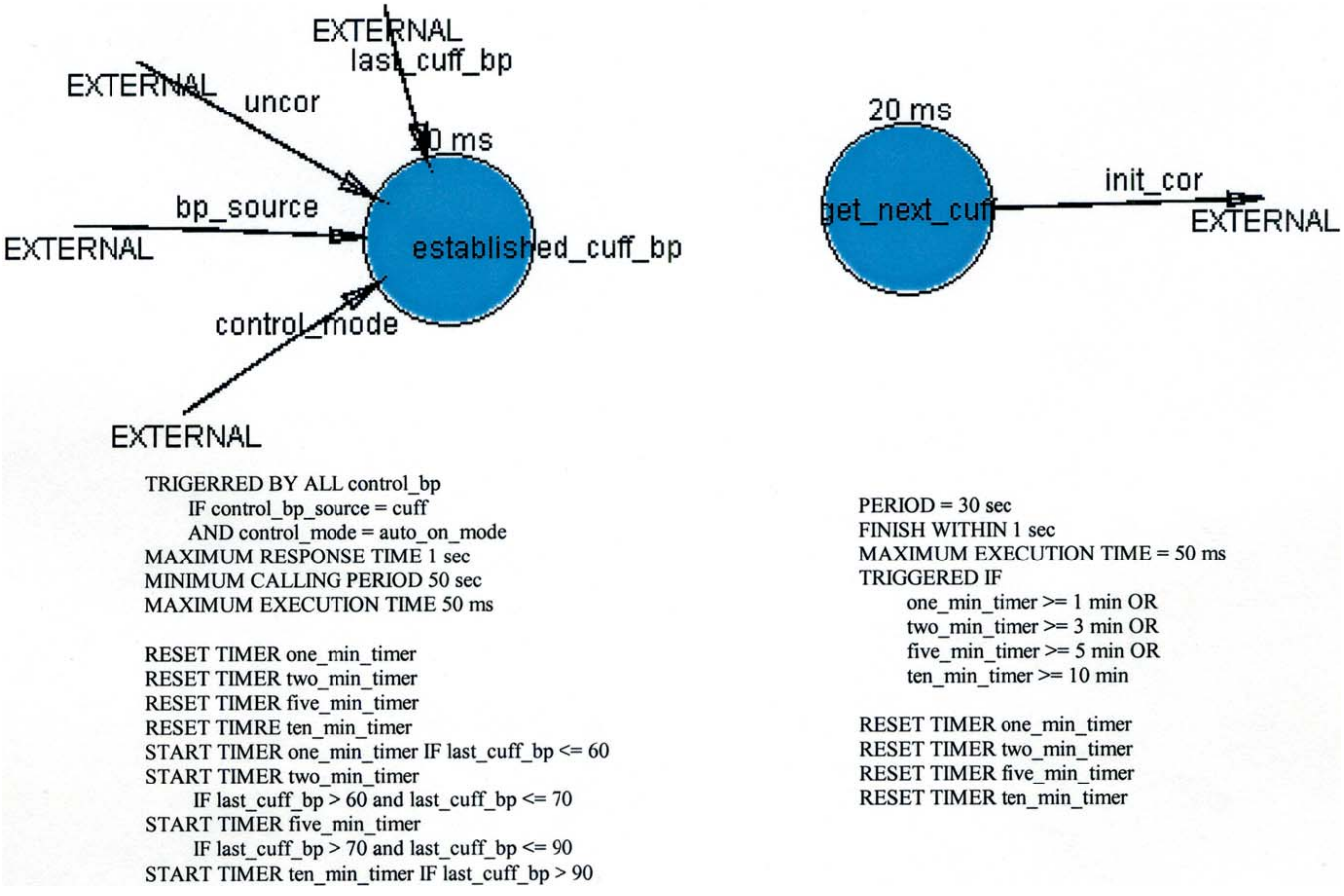

Fig. 21. Model for a cuff blood pressure monitor policy

if the mean blood pressure is (60–70], cuff pressures will be taken once every 2 min;
if the mean blood pressure is (70–90], cuff pressures will be taken once every 5 min;
if the mean blood pressure is above 90, cuff pressures will be taken once every 10 min."

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` operator, on the other hand, polls the timers once every 30 s and issues the `init_cor` command to the `monitor_bp` module in model 4 if any of the active timers reaches its preset trigger conditions.

Since the above design implements the policy via PSDL directly, users can accommodate policy changes easily without the need to modify any 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 in model 4, the above design avoids any conflict with such events since it will reset its timers automatically whenever it receives a new cuff blood pressure reading.

CAPS-PC provides the essential facilities for users to create and modify the models. It is easy to reason about complexity using CAPS-PC. When a level is determined to be too complex, it is easy to decompose. Also, when there are many data streams from one operator to another, it is easy to simplify, often by using user-defined types for the stream. By trying to fully represent each requirement in the model, it became clear which requirements were fully/consistently specified and which were not.

The tool provides an effective means of performing requirements consistency and understandability checking. It also provides some degree of computer-aided inconsistency checking and data entry propagation at the user interface level and a semantic check via the translator. Figure 22 shows a simple GUI of an executable prototype for model 4 that has 20 composite operators and 89 atomic operators. The executable prototype consists of 14.7 K lines of source code, 8.5 K of which are generated by the translator and the scheduler of CAPS-PC.

The prototype effort also revealed the need for future enhancements to CAPS-PC. Future enhancements include abstraction for data streams, visual queues for the declaration and use of timers, multiple views for requirements traces, and better facilities for constructing user-defined types.

8 Conclusions and future work

In the field of software engineering, requirements analysis is one of the most important stages in software develop-

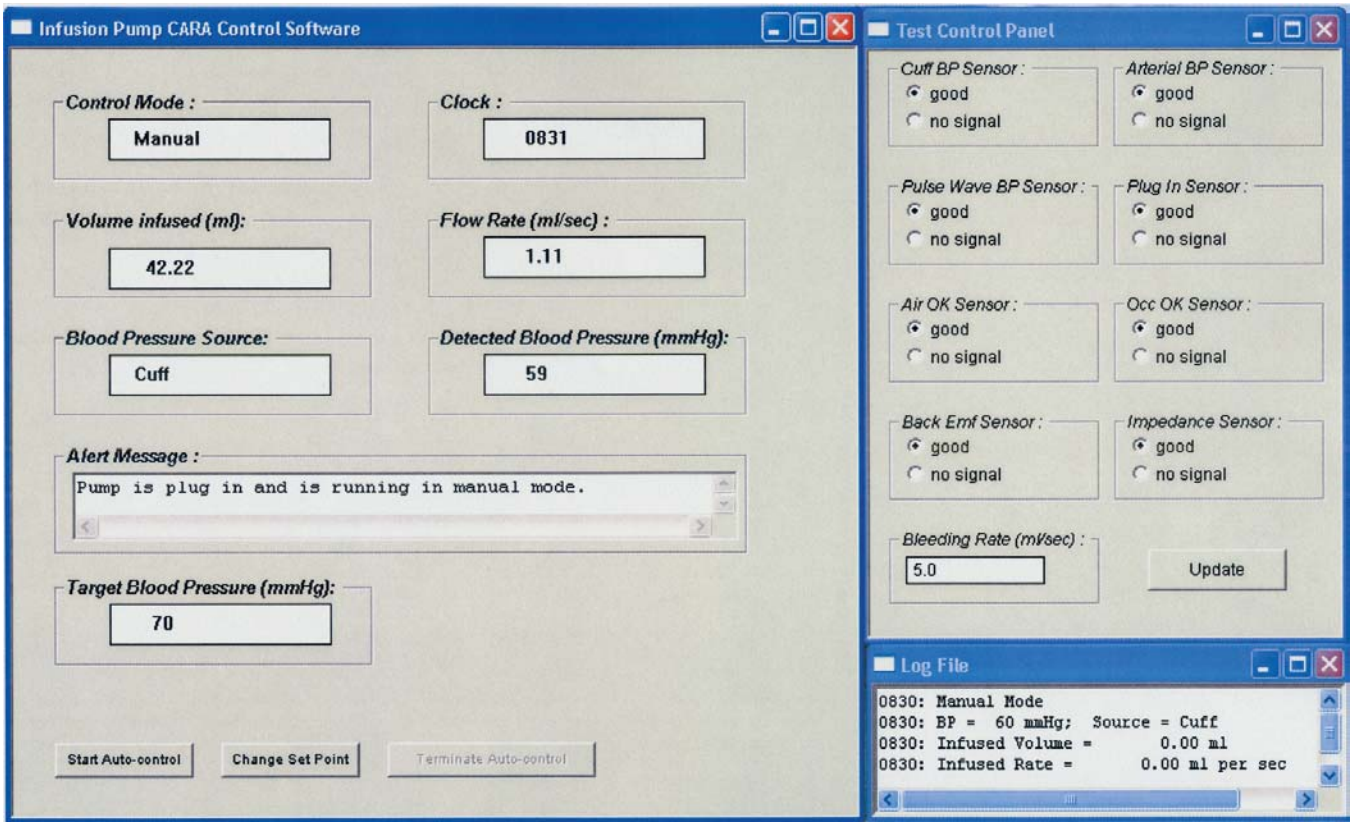

Fig. 22. Graphical user interface of the executable prototype for model 4

ment. It plays an essential role in achieving a reliable, robust, safe, and cost-effective software product. A defect found during requirements engineering costs two orders of magnitude less to fix than the same defect found after delivery [4]. Catching faults at the beginning of the development process can prevent high development cost, delays in product delivery, and loss of reputation. The CARA, as a safety-critical software system, especially requires a complete and correct requirements document for subsequent system development.

8.1 Parallel prototyping

Our parallel prototyping experiments based on the initial requirements document of the CARA system demonstrated that conceptual prototyping conducted by multiple teams can significantly help in revealing omissions and ambiguities in the requirements document. Each team designed its model independently with its individual evaluation focus, which made its design expose requirements problems not exposed by the other models. The unduplicated requirements problems exposed by each design model resulted in the high quality of the integrated prototype model. We also found that dividing concerns among team members and developing simple overlapping models independently by focusing on different issues was helpful in keeping us from getting lost in a complex problem. We believe this was largely due to the fact that focusing on a coherent but proper subset of the requirements using an informal slicing technique reduced the amount of relevant detail to a level that a single person could understand. This made it much easier for people to identify missing issues and to find the associated implications. We conjecture that the diversity of backgrounds and interests of the different teams helped to form natural focus areas and made the process even more effective.

In addition, the information exchanged between teams through meetings held at each stage led to the cross examination and evaluation of the reasonability of the findings and the feasibility of the design. Later cross comparisons helped expose issues that one person missed and another found in a different context. By explicitly asking whether an issue raised in one context was relevant to the others, we found errors of omission, such as the need for a test instrumentation interface to fully explore the issues of concern in model 5, as discussed in Sect. 4.5. Parallel independent efforts helped to expose ambiguities in natural language requirements and tacit assumptions of the domain experts that wrote the requirements. For example, some teams interpreted "impedance" in the document as "electrical impedance" and others as "fluid impedance". Both concepts are relevant to the application domain and both interpretations were plausible to software developers who were not experts on the LSTAT system. Cross comparison identified the ambiguity.

We also found that the process of developing a high-level architecture triggered many requirements questions and that parallel work by a team of analysts was effective in identifying them. A high-level prototyping language and tool support was needed to avoid getting bogged down in low-level details of the code and to enable the analysts to focus on requirements issues. A constructive model helped us find missing parts of the requirements and unanswered questions that had to be resolved before a system could be built. As the complexity of the embedded systems we develop increases, this process and the application of this and similar tools will prove extremely valuable in developing reliable, safe, and robust software.

8.2 Future work

In the future, we will further explore the efficiency and coverage of requirements inspection by using multiple specifications and joint inspection methods to conduct the prototype design. As mentioned earlier in this article, our 5 teams were able to uncover more than 12 different discrepancies in the requirements documentation, either as ambiguous requirements or as requirements omitted altogether. Some discrepancies were found by several teams, but others were found by only one. This raises the question of how many teams would be optimal for such a development effort. Obviously too many teams would be cumbersome to manage and the return on investment of the additional teams would be questionable. But in our case study, had we used fewer teams, some discrepancies may have been missed. The question of what the most effective mix of teams is for a given project deserves further attention. It could be useful to explore the use of multiple specifications to conduct the design and assess the impact of that approach on the refinement of requirements documentation.

We plan to refine our languages, models, and tools to respond to specific deficiencies identified in the initial CARA prototyping effort and to use the improved tools to establish a measurable basis for high confidence embedded systems. Future work on CARA will include completion of the prototype to the point where measurements can be made and exploration of ways in which such measurements can be related to the degree of confidence users can put on systems. We also plan to fix deficiencies in the models and tools that were exposed by this exercise. For example, we found that we needed to decompose data streams into finer generic data streams to keep complex architecture understandable. We also found that safety concerns required expressing new kinds of constraints. For instance, particular logical processes should be hosted on independent hardware to remove common causes for coincident failures (common mode failures).

Acknowledgements. We would like to thank the FDA for providing the case study and the requirements description and the ARO for their support. We would also like to thank the teams at University of Pennsylvania, SUNY at Stony Brook, Stanford University,

Software Engineering Automation Center at Naval Postgraduate School, and the rest of our research team. Special thanks to D. Hislop, P. Jones, S. Smolka, R. Cleaveland, I. Lee, H. Sipma, P. Iyer, M. Shing, W. Ray, L. Qiao, X. Liang, and N. Chaki.

References

1. Alur R, Arney D, Gunter E, Lee I, Nam W, Zhou J (2002) Formal specifications and analysis of the computer assisted resuscitation algorithm (CARA) infusion pump control system. In: Proceedings of Integrated Design and Process Technology (IDPT), Pasadena, CA, 23–28 June 2002
2. Avizienis A (1985) The N-version approach to fault-tolerant software. IEEE Trans Softw Eng 11(12):1491–1501
3. Bastani FB, I-Yen L, Linn J, Rao K, Winter VL (2001) Design for independent composition and evaluation of high-confidence embedded software systems. In: Proceedings of the Monterey workshop on engineering automation for software intensive system, Monterey, CA, 18–20 June 2001, pp 198–207
4. Boehm B (1987) Industrial software metrics top 10 list. IEEE Softw 4(5):84–85
5. Bernstein L (1996) Forward: importance of software prototyping. J Sys Integ (Special Issue on Computer Aided Prototyping) 6(1):9–14
6. Eugene SW (2002) CARA infusion pump project. SUNY at Stony Brook. Available at: `http://bsd7.starkhome.cs.sunysb.edu/ ~cara/`
7. Guan Z, Luqi (2003) A software prototyping framework and methods for supporting human's software development activities. In: Proceedings of the workshop on bridging the gaps between software engineering and human computer interaction. International conference on software engineering, Portland, OR, May 2003, pp 114–121
8. Guan Z, Luqi et al (2002) Computer aided prototyping for dependable interactive system development In: Proceedings of the 5th Asia-Pacific conference on computer human interaction, Beijing, China, 1–4 November 2002. Science Press, China, pp 480–490
9. Guler M, Kejriwal N, Wills L, Clements S, Heck B, Vachtsevanos G (2002) Rapid prototyping of transition management code for reconfigurable control systems. In: Proceedings of the 13th IEEE international workshop on rapid system prototyping, Darmstadt, Germany, July 2002, pp 76–83
10. Janka RS, Wills LM (2000) Combining virtual benchmarking with rapid system prototyping for real-time embedded multiprocessor signal processing system codesign. In: Proceedings of the 11th international workshop on rapid system prototyping – shortening the path from specification to prototype, Paris, 21–23 June 2000, pp 20–25
11. Kantorowitz E, Guttman A, Arzi L (1997) The performance of the N-Fold requirement inspection method. Require Eng J 2(3):152–164
12. Kordon F, Luqi (2002) An introduction to rapid system prototyping. IEEE Trans Softw Eng 28(9):817–821
13. Kraemer B, Luqi, Berzins V (1993) Compositional semantics of a real-time prototyping language. IEEE Trans Softw Eng 19(5):453–477
14. Kuhl M, Spitzer B, Muller-Glaser KD, Dambacher U (2001) Universal object-oriented modeling for rapid prototyping of embedded electronic systems. In: Proceedings of the 12th international workshop on rapid system prototyping, Monterey, CA, 25–27 June 2001, pp 149–154
15. Lee, I (2003) Advanced tool integration for embedded system assurance (HASTEN). University of Pennsylvania. Available at: `http://www-2.cs.cmu.edu/~weigand/aro/presentations/upenn_lee_1.pdf`
16. Luqi (1989) Handling timing constraints in rapid prototyping. In: Proceedings of the 22nd annual Hawaii international conference on system sciences. Kailua-Kona, HI, January 1989, pp 417–424
17. Luqi (1993) Real-time constraints in a rapid prototyping language. Comput Lang 18:77–103
18. Luqi (1996) System engineering and computer-aided prototyping. J Sys Integ (Special Issuse on Computer Aided Prototyping) 6(1):15–17
19. Luqi, Berzins V, Yeh R (1988) A prototyping language for real time software. IEEE Trans Softw Eng 14(10):1409–1423
20. Luqi, Chang C, Zhu H (1998) Specifications in software prototyping. J Sys Softw 42(2):150–177
21. Luqi, Berzins V, Shing M, Puett J, Guan Z, et al (2002) Infusion pump. Technical Report No. NPS-SW-02-004, Naval Postgraduate School, Monterey, CA, September 2002
22. Luqi, Qiao Y, Zhang L (2002) Computational model for high-confidence embedded system development. In: Proceedings of the Monterey workshop on radical innovations of software and systems engineering in the future, Venice, Italy, October 2002, pp 7–11
23. Luqi, Shing M, Berzins V, Puett J, Guan Z, et al (2003) Comparative rapid prototyping: a case study. In: Proceedings of the 13th international workshop on rapid system prototyping, San Diego, 9–11 June 2003, pp 210–217
24. Martin J, Tsai WT (1990) N-Fold inspection: a requirements analysis technique. Commun ACM 33(2):225–232
25. Ramamoorthy CV et al (1981) Application of a methodology for the development and validation of reliable process control software. IEEE Trans Softw Eng 7(6):537–555
26. Schneider GM, Martin J, Tsai WT (1992) An experimental study of fault detection in user requirements documents. ACM Trans Softw Eng Methodol 1(2):188–204
27. Siewiorek DP, Smailagic A, Salber D (2001) Rapid prototyping of computer systems: experiences and lessons. In: Proceedings of the 12th international workshop on rapid system prototyping, Monterey, CA, 25–27 June 2001, pp 2–8
28. Spitzer B, Kuhl M, Muller-Glaser K (2001) A methodology for architecture-oriented rapid prototyping. In: Proceedings of the 12th IEEE international workshop on rapid system prototyping, Monterey, CA, 25–27 June 2001, pp 200–205
29. WRAIR Department of Resuscitative Medicine (2001) Narrative description of the CARA software. Proprietary Document, WRAIR, Silver Spring, MD, January 2001
30. WRAIR Department of Resuscitative Medicine (2001) CARA pump control software questions, version 6.1. Proprietary Document, WRAIR, Silver Spring, MD, January 2001
31. WRAIR Department of Resuscitative Medicine (2001 CARA tagged requirements, increment 3, version 1.2. Proprietary Document, WRAIR, Silver Spring, MD, March 2001