Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Formally Specifying CARA in Java*

Eugene W. Stark

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794 USA, e-mail: stark@cs.sunysb.edu

Received: / Revised version:

Abstract. A restricted dialect of Java is proposed as a language for writing formal specifications for reactive systems. Specifications written in this dialect have one Java class per system module. Each class uses static fields to record module state, uses synchronized static methods as entry points for services provided by the module, and communicates with other modules by method calls. Specifications written in this form are directly executable, can serve as a reference model for subsequent implementations, and can also be used as a target for formal verification techniques. Application of the method to construct an executable specification of the CARA (Computer-Assisted Resuscitation Algorithm) system is described.

Key words: specification – verification – reactive systems – Java – medical devices

1 Introduction

CARA (Computer-Assisted Resuscitation Algorithm) is a software system that provides closed-loop control to a high-output intravenous infusion pump. The system is intended to infuse fluids to resuscitate medical patients who are in danger of developing severe hypotension; for example, soldiers who are bleeding from injuries sustained in a battlefield situation. The CARA software is designed to operate automatically, thereby minimizing the need for attention by medical personnel. During automatic operation, the system takes periodic blood pressure readings from multiple sensors, applies a "corroboration" procedure to determine the value to be used as a basis for control, and then computes an analog output voltage to be used to control the flow rate of the infusion pump. The system must detect and respond appropriately to a variety of anomalous and error situations that can arise during operation. The actual CARA system implementation is being developed by Walter Reed Army Institute of Research (WRAIR) and their contractors. For a more detailed overview of the CARA system, the reader is referred to other papers in the present volume.

Our interest in CARA arises out of our research into the application of formal verification techniques, such as model checking, to the development of safety-critical embedded systems, of which CARA is a good example. The CARA system is sufficiently complex as to provide a challenge to current formal verification techniques, but sufficiently simple that it is reasonable to set as a goal the verification of important properties that encompass most or all of the system. In short, CARA can serve as a good driving example for our research in formal verification.

In order to apply formal verification techniques to CARA, it is necessary to have a formal description of the system to be verified. In late January 2001 we were provided by Walter Reed Army Institute of Research (WRAIR) some requirements documents (unpublished: 1999, 2001) they had developed for CARA. Some of these documents consisted of stylized lists of numbered items detailing various behavioral requirements on the CARA system. Another document consisted of informal English prose giving an overview of the system and its intended mode of use (i.e. its "SOP"). Although there were a variety of ambiguities and inconsistencies in these requirements documents, because a CARA prototype did not

* This research was supported in part by the National Science Foundation under Grant CCR-9988155 and the Army Research Office under Grants DAAD190110003 and DAAD190110019. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the Army Research Office, or other sponsors.

yet exist it was necessary for us to accept these documents as the authoritative description of CARA.

From February, 2001 to early June, 2001, two members of our group (Prof. Arne Skou from Aalborg University in Denmark who was visiting Stony Brook during the period, and Stony Brook Ph.D. student Arnab Ray) worked on formalizing portions of the CARA requirements using the UPPAAL [9] and the Concurrency Workbench of the New Century (CWB-NC) [3] verification tools. The status of that work as of June, 2001 is described in an unpublished report [13]. Ray has subsequently continued his work on CARA using the Concurrency Workbench, and the results are described in another paper in the present volume.

By June, 2001, certain limitations of the approaches that had been taken by Skou and Ray had become apparent to me. Specifically, in the process of trying to construct formal models of the CARA system in the rather low-level languages provided by the formal verification tools, quite a number of abstractions and simplifications had been made, with the result that it seemed very difficult to make a strong case that the formalizations in any sense faithfully captured "the CARA system." In my own mind, I had significant difficulty imagining any way to argue, before the engineers constructing the actual CARA prototype, that the formal models constructed by Skou and Ray had any particularly close relationship with the real system, or that properties they had formally verified about their models had any particular implications for the real system. Thus, I began to consider how to construct a formal description of the system that would correspond as closely to the requirements documents as I could manage. I was hoping that such a formal specification could serve as a "master reference model" that could be readily compared with the concrete prototype being constructed by WRAIR as well as used as a basis for more abstract models to which formal verification techniques could be applied.

To develop a more comprehensive model of CARA that fully and faithfully captured the content of the requirements documents, it seemed necessary to me that the model be coded in a higher-level language than that provided by the UPPAAL or the CWB-NC verification tools Ray and Skou were using. In addition, it seemed necessary to have computerized support for checking the consistency of the model in various ways; for example, to make sure that every service required by some module in the system was actually implemented somewhere. Finally, I felt that it would be very difficult to remove from the model the large number of obvious errors that were bound to be present, unless we had some capability of executing or simulating the model to permit the use of traditional debugging techniques. Only once the model had been cleared of the most blatant errors would it make sense to look for the kind of subtle problems for which formal verification would be most useful.

This last point, concerning "blatant errors" versus "subtle problems," perhaps deserves some further clarification. As is well known to anyone who has used any type of formal language such as a programming language or a formal logic, it is essentially impossible in all but the most trivial situations for a human being to produce an error-free expression in a formal language on the first attempt. Instead, the formal artifact must be checked in various ways to reveal the existence and location of errors, which are then corrected. For a purely declarative, non-executable specification in a formal logic, the checking would typically be performed by deducing logical consequences to determine, for example, whether the specification itself is consistent, whether desired "good" behavior satisfies the specification, and whether undesired "bad" behavior is ruled out by the specification. This deduction of logical consequences is generally too tedious to be carried out by a human being; an automated tool such as a theorem prover or model checker is required. On the other hand, for an executable specification, a significant amount of checking can be carried out using traditional debugging and testing techniques: various executions are generated which are then examined to see if any errors are manifest. Although it is well-known that human beings are not very reliable when it comes to certifying the absence of errors in an execution, in fact they can be quite effective at noticing the presence of many kinds of errors, especially errors resulting from "trivial" typographical or coding mistakes.

We can thus crudely classify the the kinds of errors that can occur in a specification either as "obvious" errors, which those that manifest themselves in a large fraction of "normal" executions and which are easily spotted by a human being, or as "subtle" errors, which are those that are evident either only in a small fraction of "unusual" executions, or which by their nature require tedious analysis of an execution in order to be detected. An example of an obvious error that we actually made in developing the CARA specification was a misplaced statement bracket whose effect was to cause the system frequently to enter auto-control mode without the operator having pushed any button to request it. An example of a subtle error we actually made was an input dialog that triggered an inappropriate action, due to the fact that the system state had changed between the time the dialog was displayed and the time the user's response was entered, in such a way that the dialog itself was no longer relevant. Whereas obvious errors are readily found and removed by normal debugging techniques, it can require a significant amount of time (both human and computer) to construct and run tests to identify such mistakes with current formal verification tools. So traditional debugging techniques will likely be more cost-effective for removing obvious errors. Formal verification techniques, however, can identify subtle errors that persist even after extensive debugging and testing,

and for such errors the cost of preparing the required tests is justified.

2 Formal Specifications in Java

To address the issues laid out above, I decided that it would be useful to develop a formalization of the CARA requirements using the Java programming language. By coding the model in Java, the Java compiler could be used to perform various consistency checks. Since Java is a full-fledged programming language, it would be possible to execute the model and eliminate major errors using traditional debugging. Finally Java also comes with the *Javadoc* documentation generator, which generates browsable HTML documentation from specially formatted comments included with the source code. Documentation generated in this way would be a great help in keeping track of details of the model as it was developed.

It is important to note that even though a formal model of CARA constructed in Java would be executable, I did not intend for this model to be the same thing as an implementation of the system. In general, the formal model would be more abstract than an actual implementation: I would attempt not to introduce details into the model that were neither mentioned explicitly in the requirements nor implicit from the manner in which they were stated. Also, the formal model would avoid any platform-dependent considerations, such as whether the final system would be event-driven or interrupt driven, except insofar as seemed relevant and necessary to an accurate rendering of the requirements.

Although the use of Java to code the CARA model would free me to a great extent from distracting constraints imposed by low-level modeling languages, I realized that if arbitrary features of Java were to be used in the model it would be difficult for the model to serve as a basis for formal verification. So, to keep the distance from becoming too great between the Java reference model and low-level versions used for formal verification, I imposed a number of restrictions on the way in which the model was coded. These restrictions are described in more detail below. Overall, the effect of these restrictions is to ensure that the CARA model can be viewed as a large state machine that is defined as the parallel composition of a number of components, in which the components communicate with each other in a fixed pattern by method calls. Such a view is close to that required for coding the model in low-level process-algebraic languages such as those supported by the CWB-NC.

Process-algebra-based architectural design languages (ADLs, see *e.g.* [1,2,6,11]) also represent an attempt at providing a method for formal system description at an abstract level, and could possibly be seen as candidate formalisms for the CARA specification. However, the focus in such languages, such as Wright [1] and PADL [2], is on the formal description of "component types" and the verification that instances of these types will always interoperate in a compatible fashion. The protocols by which instances of component types interact constitute a simple, separable part of the overall system behavior, and low-level process algebra makes a good tool for specifying them. For the CARA specification though, the point is not to describe a general class of component *types* and their interactions, but rather to formalize *specific* system and component behavior described in the informal requirements documents. For this purpose, an ADL that provides only process algebra as its behavioral specification language would not be much easier to use than would the input language for a process-algebra-based verification tool. So ADLs do not seem to be a solution to the problems posed by the CARA specification.

The first decision I made in formalizing the model was to adopt a decomposition of the CARA system into a small, fixed set of modules having a static communication pattern. CARA is simple enough that an adequate modularization is obtained in this way, and Skou and Ray had made a similar decision at an early stage in their work. The modularization I chose, which is similar to those devised previously by Skou and Ray, comprised seventeen modules, of which five represent entities external to the CARA system, and the remaining twelve represent internal modules of the CARA system. The modules representing external entities are as follows:

- `Patient`: Models the patient.
- `Pump`: Models the infusion pump.
- `ArterialSource`: Models the arterial line blood pressure sensor.
- `PulseWaveSource`: Models the pulse wave velocity blood pressure sensor.
- `CuffSource`: Models the cuff blood pressure sensor.

The modules internal to the CARA system are as follows:

- `PumpControl`: Uses blood pressure and set point information to calculate control voltage to be supplied to pump. Also monitors for falling blood pressure.
- `PumpMonitor`: Tracks the status information supplied by the pump and estimates flow rate and total infused volume.
- `ArterialBP`: Obtains blood pressure readings from the arterial line sensor.
- `PulseWaveBP`: Obtains blood pressure readings from the pulse wave velocity sensor.
- `CuffBP`: Obtains blood pressure readings from the cuff sensor.
- `Corroborate`: Performs the corroboration function for the blood pressure sensors, thereby determining which sensor will be used for control.
- `Mode`: Keeps track of the current operating mode and manages transitions between modes.

- `Dialog`: Handles caregiver/operator input via buttons and dialog boxes.
- `Display`: Represents the caregiver/operator information display.
- `Logging`: Handles the writing of messages to the resuscitation log.
- `AlarmControl`: Handles the raising and clearing of the various alarms.
- `Timer`: Provides timing services for other modules.

Figure 1 contains a schematic diagram that shows the various modules and their primary communication relationships. (To avoid clutter, no relationships are shown explicitly for `Timer` and `Logging` because each of these modules communicates with most of the other modules in the system.) Although one can probably argue that the the five modules representing external entities are implicit in the requirements documents and would thus have to be present in any reasonable formalization of the system, one probably cannot make the same argument about the particular modularization I chose for the system internals. A different, independently developed formalization of the CARA system (for example, an actual implementation) would likely disagree to some extent with my choices. Such a discrepancy would make it more difficult to draw conclusions about an actual system implementation from results of any validation or verification performed on the reference model. Though I would rather not have introduced right at the outset an *ad hoc* assumption about the structure of the system, some kind of modularization was necessary at this stage in order to end up with an understandable and manageable system description.

One Class Per Module. After having settled upon a modularization, I next decided to model the system in Java by representing each system module as a single Java class, whose methods would correspond to services that it could provide to the other modules. In more detail, the basic idea was to think of each class as representing a kind of state machine whose actions are partitioned into *input*, *output*, and *internal actions*. An input action represents the occurrence of a call to one of the exported methods of the module. An output action represents a call made by code within the module to a method of some other module. Internal actions represent internal computation steps performed by the module as a result of a call to one of its methods. This point of view corresponds closely to that supported by the *I/O automaton* model [10] developed by Nancy Lynch and her students, and it also supports my eventual goal of formalizing some aspects of the system in terms of *probabilistic I/O automata* [16,15].

All Fields and Methods Static. Since the number of modules in the CARA system is fixed, the classes representing them would not have any dynamic instances (*i.e.* objects), but would instead simply serve as a way of packaging a fixed collection of state variables together with methods to operate on those state variables. I therefore stipulated that all fields and methods of the classes representing system modules should be declared `static`, and the Java `new` construct would not be used with these classes.

Simple State Variables. I required that the state variables (fields) of each module always be simple boolean, integer, or floating point variables; in particular, no data structures would be used. This restriction was designed to retain some possibility of deriving from the CARA specification various abstractions that could be analyzed by formal verification tools designed to handle finite-state models. Even though integer and floating point variables have very large sets of possible values, a significant amount of formal verification with finite-state tools can still be achieved if one first abstracts the large value spaces to small finite sets. For example, for CARA it might make sense to abstract a floating point value for blood pressure to one of the seven values: "ridiculously low," "well below set point," "below set point," "at set point," "above set point," "well above set point," and "ridiculously high." With simple integer and floating point variables, it is relatively easy to devise such abstractions. If arbitrarily complex data structures are used, it is not.

I should note that in the actual code, I did not usually use the Java type `double` to represent floating-point values such as blood pressure or flow rate. Instead, I introduced auxiliary classes as "wrappers" for floating point values, so that each variable in the specification could be given an appropriate type. I introduced five such classes altogether:

- `FlowRate`: Models a rate of fluid flow.
- `Pressure`: Models a blood pressure.
- `Time`: Models a time interval.
- `Voltage`: Models a voltage.
- `Volume`: Models a volume of fluid.

These classes were slightly more than just wrappers, since they also had methods for constructing and decomposing values according to specific units of measurement. For example, the `Time` class has methods `inSeconds()` and `inMinutes()`, both of which take a value of type `double` and return a value of type `Time`. Thus, these auxiliary classes facilitate the free use of dimensional quantities in the code, without having to keep explicit track of the associated units.

I found the "no data structures" restriction to be too limiting in one particular module: `AlarmControl`. This module keeps track of a large number of different alarm conditions that each have to be handled in a similar way. To avoid extremely long and repetitive code, I organized the state of the many alarms into a few fixed-size arrays that could be accessed by loops. However, except for the fact that the code was significantly shortened

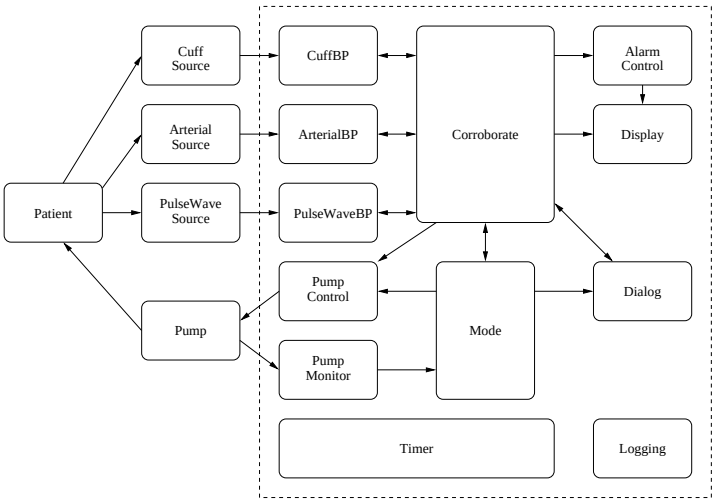

Fig. 1. Schematic Diagram Showing CARA Modularization

and redundancy eliminated, the use of arrays and loops was inessential in the sense that they can be regarded as macros that expand to straight-line code accessing simple variables.

No Exported Fields. In process-algebraic models of the type I hoped eventually to derive from the Java-based reference model, the various processes in a system typically do not access directly each others' state variables, but instead communicate with each other by way of synchronized events. In my model, synchronization occurs when one module makes a call (output) to an exported method (input) of another module. To guard against direct outside access to state variables, I imposed the requirement that all fields of the classes representing system modules be declared `private`. (Later, I relaxed this restriction to permit `protected` declarations, as a concession to facilitate the construction of a system simulator that could display the state of a module graphically without requiring that explicit support code for this purpose be incorporated into the formal specification. This is described in more detail in a later section.)

No Returned Values. The synchronized-event communication mechanism supported by process-algebraic models is in a sense a very low-level communication primitive. Higher-level communication patterns, such as call/return, are not supported directly by such a primitive, but instead have to be simulated. To maintain a close correspondence with what can be directly represented in process algebra, I decided that the exported methods of the classes representing system modules would all return `void`. Thus, although the invocation of a service provided by a module could include parameters, if a value were to be returned as the result of such an invocation, this would have to be accomplished by a "callback," rather than by the direct return of results.

One Caller per Method. Although some process-algebraic models, such as CCS [12], support as primitive the possibility that an input action may synchronize with more than one output action over the course of system execution, other models do not provide such support. I/O automaton models in particular are based on the notion that synchronization occurs whenever an action to be performed by one process is also a member of a statically determined alphabet of actions of another process. Since I was interested in extracting I/O-automaton-like models from the Java specifications, I required that each method exported by a system module be called by just one other module in the system. If a module was to provide a similar service to several other modules, a separate method would have to be exported for each possible caller. Another purpose of this "single-caller" restriction is to support values returned via callback: the module to which a callback should be made has to be uniquely determined by the context.

"Synchronized" Methods, with Straight-line Bodies. I imposed the requirement that all exported methods should be declared `synchronized`, and that that the bodies of all methods (exported or not) should be "straight-line" code that does not contain loops, time delays, or blocking (sleeping or yielding). Also, it should be impossible for a method to call itself recursively, either locally via a chain of calls within the same class

or globally via calls to other classes. These restrictions serve to simplify the concurrent execution semantics of the model, making it possible to perform a faithful simulation of the model and to understand the implications that verification results obtained for the model might have for a real system.

The execution model that I had in mind was as follows. Each of the classes representing an external entity would export an `update` method, which would be called periodically to mark the passage of time and to trigger the occurrence of spontaneous state changes in the module. Initially I envisioned a multithreaded execution model in which the system would be driven by a collection of threads, one associated with each external module. Each thread would contain a simple `run` loop in which it would sleep for a fixed amount of time, invoke the `update` method of its associated module, then go back to sleep. The fact that the exported methods of each module are all declared as `synchronized` implies that when a thread makes a call to such a method, it obtains exclusive access to that module until such time as the call has returned. Thus, a call made by a thread to an exported method of a class runs to completion without the possibility of interruption by a call to an exported method of that class made by any other thread. However, it is possible for a thread executing in a method of one class, say C_1 to make a call to a method of another class C_2, which in turn makes a call back to a (different) method of C_1, because once a thread holds the monitor lock on a class it is permitted to make calls to any `static` methods of that class without blocking.

The simulation facility I eventually implemented for the CARA specification supports a related but somewhat more restricted model that what I originally envisioned. In particular, just one top-level thread was used, rather than a single thread for each external module. This was done to eliminate the unknown effect of the thread scheduling mechanism on the simulation timing, thereby bringing all variables affecting the choice of execution trajectory under the explicit control of the simulator. Note, however, that the single-threaded model has a strictly smaller set of possible execution sequences than the multi-threaded model. Thus, if we observe "bad" execution sequences in the simulator we can conclude that the same sequence is possible under the multithreaded model, but failing to observe a particular "bad" behavior (such as deadlock) in the simulator does not necessarily imply that this behavior is impossible under the multithreaded model. It is necessary to maintain such considerations well in mind when attempting to determine the implications for a real system of validation or verification results performed on the formal model.

In the end, I found it necessary to violate the "straight-line code" restriction and introduce loops in a few places in the CARA specification. In each case, though, the loops that were introduced are bounded loops that can be regarded as macros that could be expanded to straight-line code. The alternative to introducing these loops would have been highly repetitive code, which would have been less transparent and more difficult to maintain than the version containing the loops.

Limited use of Private Methods. I allowed myself to use private methods in classes, subject to the same "straight-line code" restrictions required of exported methods. I found the use of private methods to be essential for avoiding extensive repetition of code in the specification. In contrast to exported methods, I did *not* require that private methods always return `void`. Note that the "no return values" restriction is not essential for private methods, which are only called by other methods within the same class, since such methods can always be regarded as macros and expanded in-line. This cannot be done for calls that cross class boundaries.

3 Executing the Specifications

As I stated at the outset, a major reason for using Java to code the CARA specifications was so that major errors could be found and removed by executing the specifications and applying traditional debugging techniques. To this end, I implemented a top-level class, called `Simulation`, to drive the execution by periodically invoking the `update` methods of the various external modules. In addition, I implemented a graphical user interface (GUI) to make it possible to control the simulation, to view and modify internal state variables of the various modules, and to emulate the caregiver display functions described in the CARA requirements documents. The result was a CARA simulator that could be run as a stand-alone Java application, or in a browser as an applet. Figure 2 shows a screen shot of the simulator. The frame is divided into three panels, the top two of which display information specifically mentioned in the CARA requirements documents, and the bottom of which provides access to simulation internals. The top panel of the frame is a mock-up of the caregiver display. The buttons mentioned in the requirements documents are shown on the left and an active "Change Set Point" dialog is shown on the right. The middle panel of the frame shows various alarm indicators and messages that are also mentioned in the requirements documents. The bottom panel of the frame contains a collection of tabs that permit the user to control the progress of the simulation and to display and modify internal state variables of the various modules.

The construction of the simulator GUI posed an interesting design puzzle. I wished to maintain a very clear separation between the very stylized code that was supposed to constitute the formal specification of the CARA system and the unrestricted code that implemented the GUI. The question was, how could I arrange to display

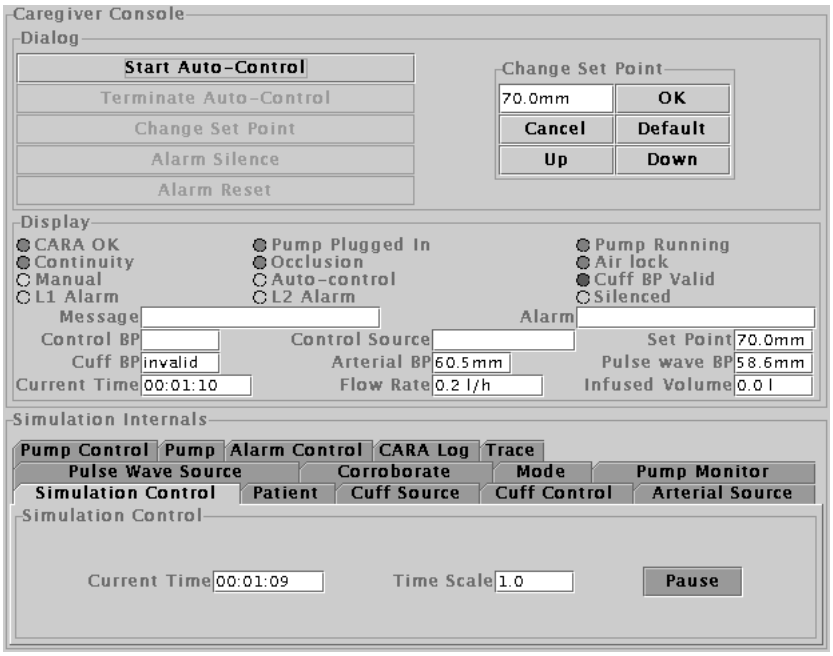

Fig. 2. CARA Simulator Application Screenshot

and modify internal state variables of the various system modules, without cluttering the formal CARA specification with substantial amounts of GUI-related code having little or nothing to do with the behavior described in the CARA requirements documents? In fact, I was indeed able to introduce the GUI code in such a way that it has very limited coupling with the formal model, and calls to the GUI do not appear at all in the formal model portion of the code.

The decoupling of the GUI and the formal model was achieved by arranging for each of the classes representing a system module to have an associated subclass that implements a GUI panel for that module. As a subclass, the GUI panel has access to all the internal state variables of the system module, assuming that we relax slightly the "no exported fields" condition and permit fields to be declared `protected` rather than `private`. When the simulation is initialized, an instance of the GUI panel for each module is created, and arrangements are made for the GUI display to be updated periodically (*e.g.* once a second). When the GUI display needs to be updated, the GUI subclass "peeks" at the current values of the state variables in the main class and updates the display appropriately. In this way, the GUI display is kept up-to-date without the necessity of making explicit calls from the formal model to the GUI. If the user wishes to change the value of some state variable, the GUI subclass simply assigns directly to that variable. The GUI subclasses are kept completely separate from their associated main classes and their code appears in separate files. It is thus possible to work on the formal model without any chance of confusion between which code constitutes the formal model and which constitutes the GUI.

I feel obliged to comment that that for each system module to have a GUI display panel as a subclass is somewhat dubious from an object-oriented point of view, since the GUI panel does not satisfy the usual "is a" relationship with the system module. In fact, it probably makes more sense for the system module to be a subclass of the GUI panel, and the same separation of GUI code and formal specification can be achieved within this alternative structure. Nevertheless, the "GUI-as-subclass" organization was the one I originally conceived of, and was therefore the one I implemented. I have recently found additional reasons to have the GUI panel as the parent class, rather than the other way around, and plan to implement this alternative organization in the next revision of the code.

The `Dialog` class has a more interesting relationship with its GUI subclass than do the other modules, which merits some further comment. The GUI panel associated with the `Dialog` class not only has the responsibility of simulating the display of the CARA caregiver console, but also of notifying the `Dialog` class when any of the input controls on the caregiver console are activated. The CARA specification includes some rather complicated requirements that concern when and how various elements (buttons and dialogs) of the caregiver console are to be displayed. For example, the requirements state that certain buttons are to be displayed under some conditions and not under others, and that buttons that are displayed may be "grayed out" or not. Static priorities are to be consulted to determine which one of multiple active dialogs will actually be displayed at any given instant. Our `Dialog` specification does not actually include any code for constructing or painting user interface

elements on the caregiver console. Instead, it just maintains a collection of boolean state variables that indicate which of the elements are currently active and which are currently displayed. The priorities and other constraints are taken into account when `Dialog` updates these state variables in response to a method call. It is the GUI subclass associated with the `Dialog` module that actually renders the caregiver console, after reading the boolean state variables from the parent class and displaying the interface elements accordingly. In this way, I was able to formalize the interesting parts of the CARA requirements on the caregiver console without actually having to include GUI code as part of the formal specification.

When one of the controls (*e.g.* a button) on the caregiver console is activated, it is necessary for the GUI panel associated with the `Dialog` class to send a notification that something has occurred. The GUI panel does this by invoking a method of the parent `Dialog` class to indicate that the control has been activated. This is the only situation in which a GUI subclass invokes any methods of its associated main class — all other communication between GUI and main classes is through direct inspection and modification of state variables.

There is one other class in the formal CARA specification that deserves some comment, and that is the `Timer` class. This class was introduced in order to encapsulate all the timing services required by other modules in the system. For example, when a blood pressure reading is required from the blood pressure cuff, the (internal) `CuffBP` module calls the (external) `CuffSource` module to initiate the reading. However, taking a reading with a blood pressure cuff is not instantaneous, but requires first inflating the cuff and then listening to the sounds made by the blood as the cuff is slowly deflated. It is also possible that an attempt to read the cuff could fail if the cuff or sensor was not properly positioned. To handle this situation, the `CuffBP` module uses a timeout as a method of bounding the amount of time taken to obtain a cuff reading. Because of the "no delays or blocking" restriction we have imposed, the timeout cannot be achieved by sleeping within one of the methods of the `CuffBP` module. Instead, when a reading attempt is initiated, `CuffBP` calls `Timer` and requests that a callback be made after a certain amount of time has elapsed. If the reading succeeds before the timeout, `CuffBP` calls `Timer` to cancel the "wakeup call." On the other hand, if a reading is not obtained by the time the timeout action occurs, the callback from `Timer` will initiate an error-handling sequence within `CuffBP`.

Although I have described the `Timer` class as an internal module of the CARA system, unlike the other internal modules it provides an `update` method that is driven by `Simulation`. Callbacks from `Timer` to the other modules occur as part of executing this `update` method. In retrospect, it seems that the `Timer` class ought actually to be two separate classes: a `TimeBase` class which is external to the CARA system and which provides an `update` method, and the `Timer` class proper, which is an internal module that provides timing services to the rest of the CARA system.

4 Overview of the CARA Specification

In this section, I give a brief overview of each of the modules in the CARA specification. In order to provide a more concrete feel for what the individual module specifications end up looking like, Appendix A lists detailed code for one of the modules: `Pump`. In the code for `Pump`, the reader may note a number of calls to the method `Trace.log`. These calls are not part of the formal specification, but rather inform the simulator of the occurrence of significant events. The simulator logs these events, thereby producing a simulation trace that can be used for post-mortem debugging and validation. I felt that the importance of having such logs far outweighed the disadvantage of the additional clutter added to the formal model by the tracing calls.

4.1 External Modules

4.1.1 Patient

The `Patient` module models the response of the human patient to fluid infusion. We use a very simple-minded model in which the blood pressure of the patient is directly proportional to the current volume of blood in the patient's circulatory system. The various blood pressure sensors periodically query `Patient` to obtain the "true" current blood pressure by calling the `Patient.getBP` method. The infusion of fluid is modeled by the `Pump` module periodically invoking the `Patient.addInfusedVolume` method to add a small increment of fluid volume. The `update` method of the `Patient` module is invoked once per second to remove a small amount of fluid volume, to model bleeding and excretion.

4.1.2 Pump

The `Pump` module models the M100 infusion pump. The pump provides logic-level outputs `CONT` (continuity), `OCC` (occlusion), and `AirOK` (air-free IV line). When these lines change state, the pump generates interrupts, which are simulated by calls to the `PumpMonitor.setCont`, `PumpMonitor.setOcc`, and `PumpMonitor.setAirOK` methods. There are also analog outputs `EMF` (back EMF) and `IMP` (impedance). These lines have to be polled by calls to the `pollEMF` and `pollIMP` methods, which usually result in immediate callbacks to the `PumpMonitor.setEMF` and `PumpMonitor.setIMP` methods, respectively. There is a certain probability that the corresponding callback will not occur for a particular polling request; this models a

failure of the A/D converter that samples the EMF and IMP lines.

The pump is controlled by an analog control voltage that determines the infusion rate. The setting of this control voltage is simulated by calling the `setControlVoltage` method. In addition, there is a logic-level input to the pump that determines whether the pump will pay attention to the analog control voltage or alternatively run at a predetermined hardware setting. Changes to the state of this line are simulated by calling the `setAnalogControl` method.

The `update` method of the pump runs once per second and sends a certain amount of infused fluid volume to the `Patient` module. In addition, the value of the `EMF` output is updated based on the current infusion rate. This is not currently modeled realistically: in the real M100 pump, the EMF value varies constantly as a rocker arm (serving as the impeller) contacts the pump tubing. This constantly varying signal has to be processed in order to estimate the infusion rate. In our model, we simply treat the EMF line as giving a direct indication of the current instantaneous infusion rate.

Also incorporated into the `Pump` module are simple two-state Markov failure models associated with each of the logic-level status lines `CONT`, `OCC`, and `AirOK`. At any time, each of these lines is either in the "failed" or "not failed" state. In the "not failed" state, the line has the logic value "true", indicating normal operational status. In the "failed" state, the line has the logic value "false", which indicates an error condition. Once per second, there is a possible transition between states. Transitions from "not failed" to "failed" occur with a probability chosen to produce a specified mean time to failure (MTTF). Transitions from "failed" to "not failed" occur with a probability chosen to produce a specified mean time to repair (MTTR).

4.1.3 **ArterialSource**

The arterial line sensor is a "beat-to-beat" blood pressure sensor that provides a direct measurement of blood pressure via a transducer inserted into an artery. The `update` method of the `ArterialSource` is invoked every 15 seconds, though it might be more realistic if it occurred in synchrony with a simulated patient heartbeat. When `update` is called, `ArterialSource` queries `Patient` to request the "true" blood pressure. This blood pressure reading is then modified by adding some Gaussian noise whose standard deviation is specified as a percentage of the true pressure. The resulting "noisy" pressure reading is supplied to the (internal) `ArterialBP` module via a call to the `ArterialBP.setBP` method.

Also incorporated into the `ArterialSource` module is a simple two-state Markov failure model, similar to that used in `Pump`. At any time, the module is either in the "failed" or "not failed" state. When the module is in the "not failed" state, it supplies blood pressure readings periodically as described above. When the module is in the "failed" state, it does not supply any readings. Every 15 seconds, there is a possible transition between states, which is taken with a probability designed to produce a specified MTTF or MTTR.

4.1.4 **PulseWaveSource**

The pulse wave velocity sensor is a "beat-to-beat" blood pressure sensor that infers blood pressure from propagation characteristics of the pressure wave resulting from each heartbeat. In our model, `PulseWaveSource` is essentially identical to `ArterialSource`, except that as separate modules they can fail independently and they can produce different blood pressure readings due to independent noise.

4.1.5 **CuffSource**

The cuff sensor uses a traditional sphygnomanometric method that involves inflating a cuff to a pressure somewhat over the systolic blood pressure of the patient and observing the waveforms associated with heartbeats as the cuff pressure is slowly reduced. In contrast to the "beat-to-beat" methods, the cuff sensor produces a blood pressure reading only when activated, and substantial time (on the order of one minute) may elapse between the time the reading is requested and the time it is obtained.

In our model, when a cuff pressure is required, the `CuffBP` module initiates the reading by invoking the `CuffSource.pollBP` method. In contrast to the `ArterialSource` and `PulseWaveSource` modules, the `CuffSource` module does not immediately return a pressure. Rather, there is a random delay which we model using a geometric distribution having a specified mean (the default is 40 seconds). When the delay time expires, the current blood pressure is returned via a callback to the `CuffBP.setBP` method.

Also incorporated into the `CuffSource` module is a Markov failure model similar to that used in other external modules. At any time, the `CuffSource` module is either in the "failed" or "not failed" state. When the module is in the "not failed" state, it responds to a `pollBP` request by returning a blood pressure as described above. When the module is in the "failed" state, it does not return any blood pressure.

4.2 Internal Modules

4.2.1 **PumpControl**

The `PumpControl` module uses blood pressure and set point information to calculate the control voltage to be supplied to the pump. It also monitors for falling blood pressure. The `PumpControl` module is informed

by Mode via the startAutoControl method when auto-control begins. The set point is initialized to the default of 70mmHg, and the pump control voltage is set so as to yield the initial flow rate of 4 liters per hour. Information about the controlling blood pressure is updated when the Corroborate module calls the setControlBP method supplied by PumpControl. Each time this occurs, PumpControl calculates a new control voltage and supplies it to Pump by calling the Pump.setControlVoltage method. Periodically (every 15 seconds), PumpControl checks whether the set point has been reached and whether it appears that the blood pressure is falling. If the caregiver modifies the set point via the change set point dialog, the Dialog module supplies the new set point to PumpControl via the setSetPoint method. When auto-control terminates, Mode uses the stopAutoControl method provided by PumpControl to convey this information.

We currently use a very simplistic control algorithm that applies the maximum control voltage to the pump when the control blood pressure is below 60mmHg, applies the minimum, "keep vein open" (KVO) control voltage to the pump when the control blood pressure is at or above the set point, and which varies the control voltage linearly with the control blood pressure when the latter is between 60mmHg and the set point.

4.2.2 PumpMonitor

The PumpMonitor module tracks the status information supplied by the pump. It also estimates the infusion rate, which it integrates over time to determine the total infused fluid volume. The PumpControl module uses the setTimeAtSetPoint method to keep the PumpMonitor informed about how long it has been since the desired blood pressure set point was attained. After the set point has been held for ten minutes, the PumpMonitor calculates a baseline "steady-state" infusion rate and monitors for increases in the infusion rate that signficantly exceed this level (such increases might indicate a substantial increase in patient bleeding).

The setAirOK, setCONT, and setOCC methods provided by PumpMonitor are called by the Pump module to simulate interrupts that occur when the corresponding logic outputs of the pump change their state. The setEMF and setIMP methods are called by the Pump module in response to calls made periodically (once every five seconds) by PumpMonitor to the Pump.pollEMF and Pump.pollIMP methods. The information obtained in this way is combined to infer some derived status information, such as whether the pump is plugged in (we currently equate this with the CONT line being at a logic "true" level) whether the pump is "OK," and to estimate the current infusion rate. Each time there is a change in pump status, the Mode module is informed by a call to the Mode.setPumpStatus method.

4.2.3 ArterialBP

The ArterialBP module obtains blood pressure readings from ArterialSource, checking the readings for validity, and delivering them on demand to Corroborate. Each time ArterialBP receives a blood pressure reading from ArterialSource, it checks this reading for validity and saves it. When Corroborate requires a blood pressure reading, it calls ArterialBP.pollBP, which results in an immediate callback to Corroborate.setArterialBP.

The ArterialBP.intrT15 method is called by Timer every fifteen seconds, to enable ArterialBP to detect the loss of BP data from ArterialSource. If fifteen seconds go by without a new reading, then the current reading is flagged as "stale" and invalidated.

4.2.4 PulseWaveBP

The PulseWaveBP module obtains blood pressure readings from PulseWaveSource, checking them for validity, and delivering them on demand to Corroborate. It is essentially identical to ArterialBP.

4.2.5 CuffBP

The CuffBP module obtains blood pressure readings from CuffSource and delivering them on demand to Corroborate. Corroborate requests a reading by calling the pollBP method provided by CuffBP, which responds by calling the setCuffBP callback method provided by Corroborate. CuffBP in turn requests a reading from CuffSource by calling its pollBP method, and CuffSource responds by calling the setBP method of CuffBP. The CuffBP module checks each reading it obtains for validity before passing it along to Corroborate.

The blood pressure cuff differs from the "beat-to-beat" sensors in that it may take much longer (a minute or more) to deliver a reading in response to a request, and in some cases it might not deliver a reading at all. To bound the time that it might take to deliver a response when a cuff reading is requested by Corroborate, the CuffBP module sets a timer when a reading is first requested. If no response is forthcoming by the time the timer expires, an invalid blood pressure is given as a response to Corroborate.

CuffBP also handles the taking of periodic readings that are required when the cuff is the only blood pressure sensor available for controlling the infusion pump. Corroborate sets the time interval between periodic readings by calling the setBPInterval method of CuffBP.

4.2.6 Corroborate

The Corroborate module is the most complex of all the modules in our specification. Corroborate is responsible for combining information from the various blood

pressure sources to determine the control blood pressure value supplied to `PumpControl`. This function involves the manipulation of the cuff blood pressure sensor to obtain readings at appropriate times to corroborate the readings provided by the other sources used for control, to monitor for lost blood pressure sources and issue alarms, and to cause auto-control mode to be terminated in case no suitable control source is available. `Corroborate` is provided with blood pressure data via calls to the `setArterialBP`, `setPulseWaveBP`, and `setCuffBP` methods. Calls to these occur in response to calls made by `Corroborate` to the `pollBP` methods supplied by the `ArterialBP`, `PulseWaveBP`, and `CuffBP` modules.

`Corroborate` supplies the current control pressure to `PumpControl` every 15 seconds. The control pressure is only updated upon receipt of new valid blood pressure data from the current control source, or upon the occurrence of a timeout indicating that the current source has been lost. If no new data arrives from the current source, then the most recent control pressure is what is supplied to `PumpControl`.

If no blood pressure source other than the cuff is available, `Corroborate` exhibits some special behavior. It calls `CuffBP.setBPInterval` to initiate periodic automatic readings of the cuff. The period of cuff readings ranges from 1 minute apart to 10 minutes apart, depending on the current blood pressure. In addition, a failure of the cuff blood pressure source to provide data in a timely fashion will cause more serious alarms when the cuff is the only source available than it would otherwise.

The core function of the `Corroborate` module is "corroboration", which involves comparing readings obtained from the so-called "beat-to-beat" sensors, the arterial line sensor and the pulse wave velocity sensor, to the readings obtained from the cuff sensor. This corroboration function is modeled as a state machine, which becomes active every 30 minutes during auto-control or whenever a change in the available blood pressure sources makes re-corroboration necessary. The transitions of this state machine are driven by the responses from the cuff source and from caregiver interaction via "override dialogs".

`Corroborate` also interprets the responses from override dialogs issued for the user. An override dialog is issued when a beat-to-beat source being used for control does not corroborate with, or match, the current cuff reading. The user is asked to respond "YES" or "NO" as to whether the current control source should be used anyway. In case of a "YES" response, `Corroborate` checks to make sure that the selected source is still valid (there are scenarios under which the source can become invalid while the dialog is pending), then remembers that the current source was selected by override. In case of a "NO" response, `Corroborate` either terminates auto-control or tries to corroborate a lower-priority source, depending on the circumstances.

4.2.7 Mode

The `Mode` module keeps track of the current operating mode and manages transitions between modes. The possible modes are "waiting", which is the mode just after the CARA system has been initialized but before the pump has been detected for the first time, "manual", in which the pump ignores the analog control voltage and pumps at its hardware setting of 0.2 liters per hour, and "auto-control", in which the pump is under active control of the CARA software.

A transition from "waiting" to "manual" mode occurs the first time the pump is detected. `Mode` is informed about the status of the pump via calls made by `PumpMonitor` to the `setPumpStatus` method. A transition from "manual" to "auto-control" mode occurs when the caregiver presses the "start auto-control" button, resulting in a call by the `Dialog` module to the `startAutoControl` method. One way a transition from "auto-control" to manual mode can occur is when the caregiver issues a YES response to a "terminate auto-control" dialog, which results in a call by the `Dialog` module to the `stopAutoControl` method. Another way auto-control can be terminated is when a call by `PumpMonitor` to the `setPumpStatus` method indicates that the pump is no longer "OK". A third way auto-control can be terminated is when a call by `Corroborate` to the `setBPStatus` method indicates that there is no longer any valid control BP. When auto-control initiates or terminates, `Mode` calls methods of various other modules, including `Corroborate`, to orchestrate the change.

4.2.8 Dialog

The `Dialog` module handles user input via buttons and dialogs. As stated earlier, `Dialog` does not actually contain any GUI code, but simply maintains the current state of the user interface as a collection of boolean variables that describe which buttons and dialogs are currently active and which are currently displayed. Each time the state of anything changes, `Dialog` recomputes the display state appropriately.

4.2.9 Display

The `Display` module handles the display of various kinds of status and alarm indicators and other data. It provides various methods to the other modules by which the state of the items to be displayed can be changed when necessary.

4.2.10 Logging

The `Logging` module handles the writing of messages to the resuscitation log. A number of methods are provided for making log entries in various situations. These methods are invoked by other modules when necessary.

4.2.11 AlarmControl

The `AlarmControl` module handles the raising and clearing of the various alarms. `AlarmControl` maintains a notion of the "current" alarm, and displays the name of the current alarm in an alarm message field when any alarm is active. Pressing the alarm reset button resets only the current alarm; if there are other active alarms, another one becomes the current alarm.

When there are active alarms, a "Silence Alarm" button is also available. Pressing this button silences the audible alarm signal for all alarms for a period of time that is different for each alarm. When the silence time for an alarm has expired, the audible alarm signal again becomes active for that alarm. Pressing the silence button again will again silence that alarm, but will not affect the remaining silence times associated with other already-silenced alarms.

Separate methods are provided by `AlarmControl` for the activating and deactivating of each different alarm. Most of these methods take a single boolean parameter that is true if the alarm is to be activated, and false if the alarm is to be deactivated. There is also a `cancelAllAlarms` method which is used by `Mode` to deactivate any active alarms when leaving auto-control mode.

4.2.12 Timer

The `Timer` module provides timing services required by the other modules. There are two types of timing services provided by `Timer`. One kind of service is settable "countdown" timers which other modules use to implement timeouts. These timers are set by supplying `Timer` with the amount of time to wait. When this amount of time expires, `Timer` performs a callback to the appropriate method of the invoking module. The other type of timing service provided by `Timer` is periodic interrupts at various frequencies. Currently, the other modules require interrupts at one-second, five-second, fifteen-second, and one-minute intervals. Not all modules require all frequencies, and some modules do not require any.

`Timer` itself contains a `update` method that is invoked by `Simulation` once per second. Counters are used to generate all the other timings from this basic one-second period.

5 Validation and Verification using the Java-based Specifications

As stated in the introduction, my objective in writing a Java-based specification for CARA was to create a "master reference model" that could serve both as a standard against which concrete implementations could be compared, and as a basis for applying formal verification techniques. So far, the primary outcome of developing the Java-based specification has been that it forced us to analyze the WRAIR requirements documents in detail; thereby identifying situations in which these documents do not adequately describe the behavior that the CARA system should exhibit.

In the course of our analysis, we found `Corroborate` to be the most difficult (and interesting) part of the specification. In creating a detailed specification for `Corroborate` that adhered closely to the behavior described in the requirements documents we found it necessary to introduce rather more in the way of concrete detail than we would have preferred to do. However, we did not see any way of constructing an executable specification that matched the requirements closely without descending to this level of detail. Having to work out the details of `Corroborate` turned out to be a worthwhile exercise, though. While exchanging E-mail messages with a WRAIR engineer (Steve Van Albert, private communication, January 2002) in an attempt to clarify aspects of the corroboration procedure that we found vague, the engineer identified a situation in the code for their own prototype where a "null response" would occur in a situation where the appropriate response was for an action to be performed. Besides this, a few other situations were identified that were not necessarily errors, but seemed to merit a closer look.

We are currently pursuing two approaches to applying formal verification techniques to the Java-based CARA specification. The first approach combines simulation and model checking to check the specifications against formalized versions of the requirements. To do this, we have extended the simulation code to provide the capability of driving the simulation with predefined scripts or "scenarios." Each scenario specifies a sequence of controls to be applied to the states of the external modules of the CARA system. The controls are designed to drive the system into a regime of operation covered by the requirements. As the system is simulated under the control of the scenario, an execution trace is recorded. It will then be possible to check this execution trace against formal assertions about what behavior should be observed. For example, a scenario might specify that the "start auto-control" button should be pressed, and that when the system has been under auto-control mode for a certain period of time, then the pump should be unplugged. The resulting execution trace could be checked for whether the appropriate alarms are triggered and whether auto-control mode is terminated within a certain period of time. We plan to use a real-time version of linear-time temporal logic to formalize the correctness assertions, and to use a model-checking procedure to automatically check whether the assertions are satisfied by the execution traces generated by simulation. This approach will enable a substantial library of tests to be constructed that will likely expose mismatches between our reference model and the requirements documents.

The tests could also be applied to traces generated by an actual CARA prototype.

Our second approach to applying formal verification to the CARA specification involves extracting lower-level, process-algebraic models from the specification, and then applying model checking, equivalence checking, and performance analysis algorithms to these lower-level models. An advantage of extracting the lower-level models automatically from the higher-level, Java-based specification is that one can then be sure that the lower-level specifications correspond exactly to the higher-level ones, and thus that analysis results obtained for the lower-level specifications are relevant to the higher-level versions. We have implemented a prototype tool capable of automatically translating Java-based specifications into a language for describing "probabilistic I/O automata" (PIOA) [14]. The tool is also able to produce input for the PRISM probabilistic model-checker [7,8]. Ideally, specifications extracted from the Java code would be input into the CWB-NC using a PIOA-language front end we have already implemented, or into the PRISM tool using its associated input language. Once compiled by the CWB-NC or PRISM, we would be able to bring to bear the suite of analysis tools they support to check correctness properties of the CARA specification. As a practical matter, however, there is still a significant gap between the size of the specifications compiled from the Java code for CARA and the size of the specifications that can be handled by the CWB-NC or PRISM.

An approach that might reduce the size of the "verification gap" would be to use the Bandera system [4,5] to perform the model extraction and verification. Bandera was designed to support model extraction and verification of realistic Java programs and consequently implements some sophisticated techniques for producing manageable-size models and verification problems. First of all, the extraction of a formal model from Java source code is handled in Bandera in much the same way as an optimizing compiler would handle the translation of Java to object code. The optimizations applied during the translation process serve to significantly reduce the size of the model that is generated. Second, model extraction is performed by Bandera only in the context of a particular formal specification that is to be verified. This permits Bandera to make use "program slicing" techniques to automatically strip away portions of the program that cannot possibly affect the truth of the given specification, thereby further reducing the size of the state space. Third, Bandera explicitly supports the use of abstract interpretation to replace data domains having a large or infinite number of values with a much smaller, finite number of values.

A tool like Bandera should be viewed as playing a supporting role for the Java-based approach to specification proposed here, rather than as an alternative to it. In writing the CARA specification, we imposed restrictions on Java not because we wanted to make things difficult in general, but because we wanted to make it somewhat difficult to write a specification that included too much in the way of concrete implementation details and that diverged too much from the kind of model natively supported by current model-checking tools. Although Bandera is targeted at the verification of realistic (and therefore unrestricted) Java programs, it can certainly be applied to Java code satisfying the restrictions we have used for the CARA specification. Indeed, its suite of optimizations ought to make it work even better in this case. To date, however, we have had only minimal experience with Bandera and we hope to use the CARA model as the basis for more in-depth experiments with it in the future.

6 Conclusions

A Java-based specification for a system of the complexity of CARA does not require an enormous amount of effort to create. The CARA model and simulator described here was constructed by graduate student Liqiang Wang and me working part-time over the period from mid-June, 2001 to mid-January, 2002. The CARA model proper comprises about 6400 lines of code, including numerous comments that key parts of the code to specific items in the requirements documents, flag ambiguities that were uncovered during the process of developing the model, and record specific decisions made that did not appear to be explicitly covered by the requirements documents. The simulator and GUI comprise about 3000 lines of additional code. As a model like this is developed, and especially after much of it has become fairly stable, it is very important to be able to track changes that were made, together with the reasons for making them. For this reason, we imported the specification into the Concurrent Versions System (CVS) revision control tool as soon as we had completed a coherent first draft. Although CVS was very helpful, it does not provide explicit support for tracking associations between lines of code and specific items in the requirements documents. Such a tool would be essential in a production setting.

I expect that the Java-based approach described here would be useful for other systems similar in complexity to CARA. The approach would most fruitful if applied at an early stage during requirements analysis, so that the model could actually serve as a formalization of the requirements as they are developed. The specification resulting from this approach would serve multiple purposes: as a kind of rapid prototype to show the behavior that is a consequence of the specification, as a reference model against which to compare subsequent implementations, and as a target on which to apply formal verification techniques.

Our simulation code, including the simulation-based validation tool we are currently working on, is not CARA-specific and could be re-used for other specifi-

cations. The overall framework for the GUI is likewise not tied to CARA, however the individual GUI panels are. The application of our approach to other systems would probably best be facilited by a tool that would permit a designer to enter the classes that make up the formal model, and then use a "visual" editor to lay out a GUI panel for each module and declare the connection between the GUI elements and the state variables for the module. The code to actually construct the GUI could then be generated automatically and combined with the generic simulation and validation code to produce a complete execution environment. It is perhaps not realistic to expect that system designers will be as disciplined as I was in adhering to a restricted dialect of Java and in keeping the system specification separate from the simulation and GUI code. For this reason, any tool to support the approach described here should explicitly enforce the necessary discipline.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
2. M. Bernardo, P. Ciancarini, and L. Donatiello. On the formalization of architectural types with process algebras. In D. Rosenblum, editor, *Proc. ACM/IEEE Int. Conf. on Fundamentals of Software Engineering (FSE-8)*, pages 140–148, San Diego, CA, 2000. ACM Press.
3. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, 2000. IEEE Computer Society.
5. J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *Proc. of the 12th Int. Conf. on Concurrency Theory (CONCUR '01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer-Verlag, 2001.
6. P. Inverardi, A. L. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, July 2000.
7. M. Kwiatkowska. Model checking for probability and time: From theory to practice. In *Proc. 18th IEEE Symposium on Logic in Computer Science (LICS'03)*, June 2003.
8. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.
9. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1:134–152, 1997.
10. N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.
11. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.
12. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
13. A. Ray, A. Skou, R. Cleaveland, S. A. Smolka, and E. W. Stark. Formal modeling and analysis of the control software for the CARA infusion pump. Unpublished prose document describing the CARA system and its intended mode of use, June 2001.
14. E. Stark, R. Cleaveland, and S. Smolka. A process-algebraic language for probabilistic I/O automata. In *Proc. of the 14th Int. Conf. on Concurrency Theory (CONCUR '03)*, Marseille, France, September 2003.
15. E. W. Stark and G. Pemmasani. Implementation of a compositional performance analysis algorithm for probabilistic I/O automata. In *Proceedings of 7th International Workshop on Process Algebra and Performance Modelling (PAPM'99)*, Zaragoza, Spain, September 1999. (to appear).
16. S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1-2):1–38, 1997.

Appendix A: Java-based Specification of the `Pump` Module

```
/**
 * Models the M100 infusion pump.
 *
 * @author Eugene W. Stark
 * @author Liqiang Wang
 */
class Pump implements Periodic {
    /**
     * The maximum infusion rate of the pump.
     * We assume that this is the infusion rate that would obtain
     * when the maximum control voltage is applied.
     */
    final static FlowRate INFUSION_RATE_MAX = FlowRate.inLitersPerHour(6.0);

    /**
     * The minimum infusion rate of the pump.
     * We assume that this is the infusion rate that would obtain
     * when the minimum control voltage is applied.
     * The M100 home page gives the minimum flow rate as 0.2lph.
     */
    final static FlowRate INFUSION_RATE_MIN = FlowRate.inLitersPerHour(0.2);

    /**
     * The "KVO" infusion rate of the pump.
     * We assume this is the flow rate that is obtained when the
     * KVO control voltage is applied to the pump.
     *
     * NOTE: The M100 infusion pump specs page gives the minimum
     * flow rate of the pump as 0.2 lph.  However, in UPenn Q167,
     * WRAIR indicates that KVO setting is 0.02 lph.
     * This discrepancy needs to be looked into.
     */
    final static FlowRate INFUSION_RATE_KVO = INFUSION_RATE_MIN;

    /**
     * Maximum control voltage input to pump.
     * NOTE: The M100 infusion pump home page does not have detailed
     * electrical specifications.  However, it does say that the
     * pump operates on 6 AAA batteries, which would suggest that it
     * is working with max signal levels of 6x1.5V = 9V.
     * So, we use 9V as the maximum voltages below.
     */
    final static Voltage CONTROL_VOLTAGE_MAX = Voltage.inVolts(9.0);

    /**
     * Minimum control voltage input to pump.
     * This has not been checked against actual pump specs.
     */
    final static Voltage CONTROL_VOLTAGE_MIN = Voltage.inVolts(0.0);

    /**
     * "Keep vein open" control voltage input to pump.
     * This may well be the same as CONTROL_VOLTAGE_MIN.
     * It has to be checked against actual pump specs.
     * In any case, it is the control voltage required to produce
     * a KVO flow rate.
     */
    final static Voltage CONTROL_VOLTAGE_KVO =
        calculateControlVoltage(INFUSION_RATE_KVO);
```

```
/**
 * Control voltage input to pump required to produce a flow
 * rate of 4 liters per hour, as per (R17.1).
 */
final static Voltage CONTROL_VOLTAGE_4LPH =
    calculateControlVoltage(FlowRate.inLitersPerHour(4.0));

/**
 * Maximum "back EMF" voltage output from pump.
 * WRAIR in UPenn Q167 seem to indicate that the back EMF
 * signal ranges from 0V to 1V.
 * Steve Van Albert of WRAIR confirmed this in E-mail.
 * The back EMF signal is not constant at all, but rather
 * varies as the pump rocker arm contacts the tubing.
 * This signal would presumably require filtering to
 * estimate the flow rate.  Here we just assume unrealistically
 * that the back EMF signal directly reflects the current
 * infusion rate.
 */
final static Voltage EMF_VOLTAGE_MAX = Voltage.inVolts(1.0);

/**
 * Minimum "back EMF" voltage output from pump.
 * This has not been checked against actual pump specs.
 */
final static Voltage EMF_VOLTAGE_MIN =
    calculateBackEMF(INFUSION_RATE_MIN);

/**
 * Probability that EMF polling will fail.
 */
final static double EMF_POLL_FAILURE_PROB = 0.1;

/**
 * The current control voltage being applied to the pump.
 */
protected static Voltage controlVoltage = CONTROL_VOLTAGE_KVO;

/**
 * Maximum "IMP" voltage output from pump.
 * E-mail from Steve Van Albert at WRAIR indicates that impedance
 * goes from 0V to 5V with a typical value of 1.5V for saline.
 * He also says if this line is cut, the IMP signal will go
 * negative for 8 to 10 seconds, and this is what they use
 * for their loss of continuity check.
 */
final static Voltage IMP_VOLTAGE_MAX = Voltage.inVolts(5.0);

/**
 * Minimum "IMP" voltage output from pump.
 */
final static Voltage IMP_VOLTAGE_MIN = Voltage.inVolts(0.0);

/**
 * Probability that IMP polling will fail.
 */
final static double IMP_POLL_FAILURE_PROB = 0.1;
```

```
/**
 * Pump mean time to failure.
 * Used for generating random failures.
 */
private final static Time MTTF = Time.inHours(24.0);

/**
 * Pump mean time to repair.
 * Used for generating random repairs.
 */
private final static Time MTTR = Time.inMinutes(5.0);

/** Whether or not random failures and repairs will take place. */
protected static boolean randomFailuresEnabled = true;

/**
 * Given a control voltage, compute the corresponding infusion
 * rate that would obtain if that control voltage were applied.
 * We assume the infusion rate varies linearly as the control
 * voltage goes from its minimum to its maximum value.
 * In reality, the flow rate will likely also depend on the
 * impedance of the fluid being pumped, but we ignore this
 * for now.
 *
 * @param controlVoltage The control voltage applied to the pump.
 * @return The infusion rate of the pump.
 */
private static FlowRate calculateInfusionRate(Voltage controlVoltage) {
    double fraction =
        (controlVoltage.getVolts() - CONTROL_VOLTAGE_MIN.getVolts())
        /
        (CONTROL_VOLTAGE_MAX.getVolts() - CONTROL_VOLTAGE_MIN.getVolts());
    FlowRate rate =
        FlowRate.inLitersPerHour
          (fraction *
           (INFUSION_RATE_MAX.getLitersPerHour()
            - INFUSION_RATE_MIN.getLitersPerHour())
           + INFUSION_RATE_MIN.getLitersPerHour());
    return(rate);
}

/**
 * Given a flow rate, compute the corresponding control voltage
 * required to produce that flow rate.  This is not used in actual
 * operation, just to initialize control voltage constants above.
 *
 * @param flowRate The flow rate to be produced.
 * @return The required control voltage.
 */
private static Voltage calculateControlVoltage(FlowRate flowRate) {
    double fraction =
        (flowRate.getLitersPerHour()
         - INFUSION_RATE_MIN.getLitersPerHour())
        /
        (INFUSION_RATE_MAX.getLitersPerHour()
         - INFUSION_RATE_MIN.getLitersPerHour());
    Voltage voltage =
        Voltage.inVolts
          (fraction *
           (CONTROL_VOLTAGE_MAX.getVolts()
            - CONTROL_VOLTAGE_MIN.getVolts())
           + CONTROL_VOLTAGE_MIN.getVolts());
```

```
    return(voltage);
}

/**
 * Given a flow rate, calculate the back EMF that would be produced.
 * We currently use a simple model for this signal in which
 * the value of the back EMF signal is linearly related to
 * the infusion rate.
 */
private static Voltage calculateBackEMF(FlowRate flowRate) {
    return(Voltage.inVolts
              (flowRate.getLitersPerHour()
               * (EMF_VOLTAGE_MAX.getVolts()
                  / INFUSION_RATE_MAX.getLitersPerHour())));
}

/**
 * The rate at which the pump is infusing fluids.
 */
protected static FlowRate infusionRate = INFUSION_RATE_KVO;

/**
 * The current back EMF signal level from the pump.
 */
protected static Voltage EMF = calculateBackEMF(INFUSION_RATE_KVO);

/**
 * The current IMP signal level from the pump.
 * A typical value is 1.5 volts for saline, per Steve Van Albert of WRAIR.
 */
protected static Voltage IMP = Voltage.inVolts(1.5);

/**
 * The current AirOK status signal value from the pump.
 * True means OK, false means bad.
 */
protected static boolean AirOK = true;

/**
 * The current OCC status signal value from the pump.
 * True means OK, false means bad.
 */
protected static boolean OCC = true;

/**
 * The current CONT status signal value from the pump.
 * True means OK, false means bad.
 */
protected static boolean CONT = true;

/**
 * The current value of the signal that tells the pump whether
 * it is under analog control.
 * True means under analog control, false means manual control.
 */
protected static boolean useAnalogControl = false;

/**
 * Called by PumpMonitor to request current EMF value.
 * The value is returned by a callback to PumpMonitor.setEMF().
 */
synchronized static void pollEMF() {
```

```
    Trace.log("Pump", "pollEMF", "EMF: " + EMF.formatVolts());
    /*
     * There is a certain probability that no value will be returned.
     */
    if(Simulation.randomChoice(1.0 - EMF_POLL_FAILURE_PROB))
        PumpMonitor.setEMF(EMF);
}

/**
 * Called by PumpMonitor to request current IMP value.
 * The value is returned by a callback to PumpMonitor.setIMP().
 */
synchronized static void pollIMP() {
    Trace.log("Pump", "pollIMP", "IMP: " + IMP.formatVolts());
    /*
     * There is a certain probability that no value will be returned.
     */
    if(Simulation.randomChoice(1.0 - IMP_POLL_FAILURE_PROB))
        PumpMonitor.setIMP(IMP);
}

/**
 * Called by PumpControl to inform the Pump of a new control voltage.
 *
 * @param volts The new control voltage.
 */
synchronized static void setControlVoltage(Voltage volts) {
    controlVoltage = volts;
    if(useAnalogControl)
        infusionRate = calculateInfusionRate(volts);
    Trace.log("Pump", "setControlVoltage",
              "Control voltage: " + volts.formatVolts());
}

/**
 * Called by PumpControl to tell the Pump to use or not to use the
 * analog control voltage.
 *
 * @param on true if the analog control voltage is to be used.
 */
synchronized static void setAnalogControl(boolean on) {
  useAnalogControl = on;
  if(on) {
      infusionRate = calculateInfusionRate(controlVoltage);
      Trace.log("Pump", "setAnalogControl", "Analog control");
  } else {
      infusionRate = INFUSION_RATE_KVO;
      Trace.log("Pump", "setAnalogControl", "Manual control");
  }
}

/** The update interval of the pump. */
public static final Time UPDATE_INTERVAL = Time.inMilliseconds(1000);

/**
 * Simulate the autonomous behavior of the pump.
 * When the pump status changes, it will inform PumpMonitor.
 * (R57, R59, R60, Q62)
 */
public synchronized void update() {
    EMF = calculateBackEMF(infusionRate);
```

```
// We assume the impedance output remains constant
// at its initial value.
//IMP = ??;

// The logic output lines toggle their state in a fashion
// governed by a two-state Markov chain, so that the probability
// of failure can be different than that of repair.
// Note that that the expected time to failure (and to repair)
// is then a geometric distribution with mean 1/p
// if p is the probability of failure (or repair) at each
// time step.  Thus, given a desired MTTF (or MTTR) t,
// the probability of failure (or repair) at each step should
// be 1/t.

if(randomFailuresEnabled) {
    if(AirOK) {
        if(Simulation.randomChoice
           (1.0/(MTTF.getSeconds()/UPDATE_INTERVAL.getSeconds()))) {
            AirOK = !AirOK;
            Trace.log("Pump", "run", "Random AirOK failure");
        }
    } else {
        if(Simulation.randomChoice
           (1.0/(MTTR.getSeconds()/UPDATE_INTERVAL.getSeconds()))) {
            AirOK = !AirOK;
            Trace.log("Pump", "run", "Random AirOK recovery");
        }
    }
    PumpMonitor.setAirOK(AirOK);

    if(OCC) {
        if(Simulation.randomChoice
           (1.0/(MTTF.getSeconds()/UPDATE_INTERVAL.getSeconds()))) {
            OCC = !OCC;
            Trace.log("Pump", "run", "Random OCC failure");
        }
    } else {
        if(Simulation.randomChoice
           (1.0/(MTTR.getSeconds()/UPDATE_INTERVAL.getSeconds()))) {
            OCC = !OCC;
            Trace.log("Pump", "run", "Random OCC recovery");
        }
    }
    PumpMonitor.setOcc(OCC);

    if(CONT) {
        if(Simulation.randomChoice
           (1.0/(MTTF.getSeconds()/UPDATE_INTERVAL.getSeconds()))) {
            CONT = !CONT;
            Trace.log("Pump", "run", "Random CONT failure");
        }
    } else {
        if(Simulation.randomChoice
           (1.0/(MTTR.getSeconds()/UPDATE_INTERVAL.getSeconds()))) {
            CONT = !CONT;
            Trace.log("Pump", "run", "Random CONT recovery");
        }
    }
    PumpMonitor.setCont(CONT);
}
```

```
        /*
         * Inform the patient about the volume of fluid that
         * has been infused since the last update.
         * This is just the current infusion rate in liters per hour
         * times the update interval in hours.
         */
        Patient.addInfusedVolume
            (Volume.inLiters(infusionRate.getLitersPerHour() / 3600.0));
    }
}
```