Linking abstract analysis to concrete design: A hierarchical approach to verify medical CPS safety*

Anitha Murugesan
anitha@cs.umn.edu

Oleg Sokolsky
sokolsky@cis.upenn.edu

Sanjai Rayadurgam
sanjai@cs.umn.edu

Michael Whalen
whalen@cs.umn.edu

Mats Heimdahl
heimdahl@cs.umn.edu

Insup Lee
lee@cis.upenn.edu

1Department of Computer Science and Engineering
University of Minnesota
200 Union St., Minneapolis, MN 55455, USA

2Department of Computer and Information Science
University of Pennsylvania
3330 Walnut St., Philadelphia, PA 19104, USA

ABSTRACT

To manage design complexity and provide verification tractability, models of complex cyber-physical systems are typically hierarchically organized into multiple abstraction layers. Formal reasoning about such systems, therefore, usually involves multiple modeling formalisms, verification paradigms, and associated tools. System properties verified using an abstract component specification in one paradigm must be shown to logically follow from properties verified—possibly using a different paradigm—on a more concrete component description. As component specifications at one layer of abstraction get elaborated into more concrete component descriptions at the next lower level, abstraction induced differences come to the fore; differences that have to be reconciled. In this paper, we present an approach to tie together distinct verification paradigms and reconcile these abstraction induced differences using a medical device cyber-physical system as an example. While the specifics are particular to the example at hand, we believe the techniques are applicable in similar situations for verifying cyber-physical system properties.

Keywords

Model-based development, Verification, Medical CPS

*This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715 and the DARPA HACMS program (contract FA8750-12-9-0179).

1. INTRODUCTION

Advanced capabilities being developed for the next generation of commercial and military aircraft will be based on complex new software. These aircraft will incorporate adaptive control algorithms and sophisticated mission software providing enhanced functionality and robustness in the presence of failures and adverse flight conditions. Unmanned aircraft have already displaced manned aircraft in most surveillance missions and are performing many combat missions with increasing levels of autonomy. Manned and unmanned aircraft will be required to coordinate their activities safely and efficiently in both military and commercial airspace. The cyber-physical systems that provide these capabilities are so complex that software development and verification is one of the most costly development tasks and therefore poses the greatest risk to program schedule and budget.

Without significant changes in current development processes, the cost and time of software development is a barrier to the deployment of the advanced capabilities needed for the next generation of military aircraft. DARPA’s HACMS project

DARPA’s META program was undertaken to significantly improve the design, manufacture, and verification process for complex cyber-physical systems. The work described in this paper directly addresses this goal by allowing the system architecture to be composed from libraries of complexity-reducing design patterns with formally guaranteed properties. This allows new system designs to be developed rapidly using patterns that have been shown to reduce unnecessary complexity and coupling between components. This work also deeply embeds formal verification into the design process to enable correct-by-construction development of systems that work the first time. The use of components with formally specified contracts, design patterns that
provide formally guaranteed properties, and an architectural modeling language with well-defined semantics ensures that the system design is known to meet its requirements even before it is implemented. Further details can be found in [1].

Such a top-down compositional approach to verification is quite appealing – allocation of verification burden parallels allocation of development burden. But, as we descend down the system hierarchy, certain practical concerns surface that complicate the picture. First, different specification formalisms, modeling notations, and verification tools – the ones most appropriate for the component being verified – are likely to be employed, which necessitates some formal semantic mapping to attach the verification result of a lower-level component to its abstract higher-level representation. Secondly, as architectural layers and model boundaries are traversed, abstraction-induced differences are discovered and must be reconciled.

In this paper, we describe such a compositional approach to verify safety properties of a closed-loop generic patient-controlled analgesia infusion system. This prototypical system includes a patient blood oxygen level monitor whose output is constantly monitored by a supervisory controller that can automatically command the pump to terminate (pain medication) infusion, should the oxygen-level fall below a critical threshold.

In prior works [24, 22], we have separately and independently described two approaches to respectively verify (i) safety properties of the closed-loop system using timed automata models in UPPAAL that included a physiological model of the patient in the loop, and (ii) critical requirements of the infusion pump control software using a compositional assume-guarantee reasoning approach on an AADL system architecture with component behaviors elaborated in Simulink/Stateflow models. A natural question then is whether these two analyses can be combined in some meaningful way such that the particular infusion pump when used as part of the closed-loop system can be guaranteed to uphold critical safety properties. In this paper, we demonstrate an approach to soundly combine these analysis results. We highlight the key aspects of our approach for providing a semi-formal semantic mapping to tie distinct verification paradigms and for reconciling the abstraction induced differences between the closed-loop system and the infusion pump system. While the specifics are particular to the case example at hand, we believe the approach is broadly applicable in similar situations for verifying complex CPS system properties.

The rest of the paper is organized as follows. In Section 4 provides a brief overview of the the medical system case study. In Section 3 the hierarchical system modeling using different tools and notations is explained. Section 4 describes the verification of the models using tools appropriate for that modeling notation. Section 5 discusses our approach of mapping abstractions to demonstrate a formal hierarchical system verification. Following a brief discussion of related work in Section 6, Section 7 concludes by summarizing our efforts, learning and our next steps in this research direction.

2. SYSTEM OVERVIEW

2.1 Closed Loop System

We considered the clinical scenario of Patient Controlled Analgesia (PCA). PCA is an approach to pain control in post-operative patients, widely used in modern hospitals. A liquid pain medication, typically an opiate such as morphine, is delivered by an infusion pump that is equipped with a request button. The pump delivers medication at a pre-defined low rate of infusion, called basal rate, which is often insufficient to control pain. Hence, the patient can request additional medication by pressing the request button, that delivers drug at a higher infusion rate, called bolus.

Figure 1: Overview of the closed-loop system

A serious side effect of opioid medication is that an overdose can lead to a respiratory failure that can result in the death of a patient. Sensitivity to the medication varies widely within the patient population, and can also be affected by the condition of the patient. Therefore, the right balance between achieving pain control and avoiding overdose is hard to achieve. For this reason, PCA guidelines require the use of a sensor, typically a pulse oximeter, that allows the clinician to detect the onset of respiratory problems. Pulse oximeters measure \( \text{SpO}_{2} \), the blood oxygen saturation. Low oxygen saturation indicates that the patient is not breathing strongly enough to supply adequate oxygen to the lungs. In current practice, the caregiver manually Adjusts settings the pump after the pulse oximeter raises an alarm.
In a clinical setting, a caregiver is typically responsible for a number of patients. Thus, there may be a delay before he or she responds to an alarm. For this reason, the quality of care may be improved by a closed-loop system illustrated in Figure 1 where a safety interlock device continuously monitors the pulse oximeter readings and stops the pump once a pre-set threshold is crossed.

2.2 Generic Patient Controlled Analgesic Infusion Pump

Infusion pumps have been involved in many incidents that have resulted in harm to the patient. The US Food and Drug Administration (FDA), through its Infusion Pump Improvement Initiative has sought to proactively promote the safe use of these devices by establishing additional requirements for infusion pump manufacturers.

To contribute to this initiative, we have been developing an archetype of system development artifacts for a Generic Patient Controlled Analgesia Pump (GPCA). One of the artifacts is an industrial-scale software controller for a GPCA pump modeled in Simulink/Stateflow. Unlike the abstract PCA pump in the closed loop system, the GPCA we considered, as shown in Figure 2, is richer in functionality, such as multiple modes of drug infusion, prescription validation, drug library, variety of system response to hazards depending upon their severity, visual and aural notifications, logging capability, routine system checks etc. At a high level, the GPCA primarily has three functions: (1) deliver the drug based on the prescribed schedule and patient requests, (2) prevent hazards that may arise during its usage by monitoring and notifying the clinician of any hazardous conditions encountered and (3) respond to the commands received.

3. MODELING

Systems are naturally constructed in hierarchies. Formal models can capture these hierarchies and help maintain intellectual control over the behavior of the system. At higher levels of abstraction the purpose of modeling is to specify and analyze the overall system functions and so the component details are heavily abstracted. As the system is refined, the focus shifts from the overall system functions to the particular component functions. The notations used to represent the system at the higher levels of abstraction may not be appropriate at lower levels. Also some components of the system may be readily available to be reused. Hence it is natural to choose different notations to model and analyze the system at different layers of abstraction. However, in many cases the modeling and analysis efforts are kept isolated within the layers of abstraction and and correspondence between the results, if there is any, is informal. In this section, we show the closed loop system modeled at several levels of abstraction using three different modeling notations: UPPAAL, AADL/AGREE, and Simulink/Stateflow. We then describe how the analyses performed at the different levels are semantically connected to establish the system level guarantees when using the Simulink/Stateflow GPCA implementation.

3.1 Closed Loop System Modeling

The key patient safety questions that need to be answered for the closed loop system are whether the pump will always be stopped, and whether it will be stopped quickly enough. There are several delay factors in the control loop. First, human physiology imposes a delay from the time the drug concentration reaches a dangerous level to the time the effects of an overdose becomes apparent in the oxygen saturation levels. In modeling the physiology, we utilize a simplified model of pharmacokinetics of intravenous delivery of anesthetic drugs presented in [4]. Second, there is a delay in the pulse oximeter, which averages a number of samples during a variable-size time window to produce a reading. Sensed values and pump commands are transmitted over the network, adding to the delay. Finally, the controller itself may add a processing delay before issuing a command to the pump.

In order to capture the essential functionality of the closed-loop system and perform analysis of timing properties of the system, we modeled the system components and interactions between them using UPPAAL. An UPPAAL model is a collection of timed automata that can communicate using synchronous channels and shared variables. The architecture of the UPPAAL model closely follows that of the system in Figure 1. The PCA infusion pump is represented by three automata, which we discuss in more detail below. The model also includes an automaton representing patient physiology, an automaton for the pulse oximeter, an automaton capturing the logic of the safety interface, and
a network automaton. For details of the model, we refer the reader to [24].

The three automata used to model the pump correspond to the three essential properties of the pump. The first property states that \textit{Infusion cannot begin until parameters of the infusion are configured and pump is started by the caregiver}. The second property specifies basal rate delivery. It states that \textit{if the pump is not stopped, it delivers medication with at least the basal rate and otherwise the infusion rate is zero}. Finally, the third property specifies bolus delivery. It states that, \textit{once a bolus is requested, medication is delivered at the bolus rate for the prescribed duration}. The timed automata corresponding to the latter two properties are shown in Figure 3.

### 3.1.1 GPCA Architectural Modeling

The generic PCA (GPCA) pump considered in [24] contains the bare minimum functionality necessary to reason about the system property, but in no way corresponds to the level of functionality and safety features expected in a commercial PCA device. As part of a related project, we have created a rich model of the GPCA that contains functionality similar to a commercial PCA device. This model is hierarchically organized, having leaf-level components written in Simulink/Stateflow that are connected using the architecture description language AADL [26].

The AADL notation supports descriptions of both hardware and software components and their interactions, and supports hierarchical composition through \textit{systems}, each of which describes a layer of a hierarchically composed architecture. AADL supports many of the constructs needed to model embedded systems such as processes, threads, devices, processors, buses, and memory. Furthermore, it contains an extension mechanism (called an \textit{annex}) that can be used to extend the language to support additional features, such as requirements modeling. The graphical representation of the architecture model is shown in Figure 4. AADL distinguishes between a \textit{system}, which describe the input/output interface of an AADL aggregate, and \textit{system implementations}, which describe the internal structure of the system.

In our model, the top most system, \textit{PCA_Pump}, has interfaces (inputs and outputs) that match the PCA interfaces of the closed loop system modeled in UPPAAL. This was done so that the level of abstraction of the \textit{PCA_Pump} in AADL is same as the PCA of the closed loop system in UPPAAL and the properties of the \textit{PCA_Pump} in AADL are always specified at the same level of detail as in the UPPAAL model.

While modeling the interfaces we grouped the individual inputs and outputs based on their sources: for the \textit{PCA_Pump} inputs, all infusion values from caregiver/clinician are grouped as \textit{CAREGIVER_IN}; all commands to the pump are grouped as \textit{PUMP_CMDS_IN} and patient bolus inputs are grouped as \textit{PATIENT_IN}. This was done in order to simplify signal routing throughout the model. This grouping of inputs was consistently done for all components at all levels of system abstraction.

The architecture, which is an implementation of the \textit{PCA_Pump} component, consists of typical infusion device components such as sensors (\textit{GPCA_HW_Sensors}), a controller which is a software component (\textit{GPCA_SW}) and actuators (\textit{GPCA_HW_Actuators}). The sensor component represents the device input hardware such as keypads and sensors such as flow rate sensor. The actuator component represents the device hardware that delivers drug to the patient and the device display. The software component is responsible for the system control. For the software, we reused the \textit{GPCA_SW} model from an earlier related work described in [22]. In addition to the typical device components, we also created interface components (\textit{GPCA_IP_Interface} and \textit{GPCA_OP_Interface}) to match and route the inputs and outputs between the abstract \textit{PCA_Pump} and concrete GPCA device.

From this point onwards, we refer to the high level

![Figure 3: Infusion pump model in UPPAAL: (a) Basal infusion control (b) Bolus control](image-url)
PCA_Pump system as the abstract model and its implementation that is the detailed architecture of the device as the concrete model.

The concrete model has more inputs than the abstract PCA_Pump model; hence the interface component matches some straightforward inputs, such as Pump_Stop of the abstract model to Infusion_Cancel of the concrete model, as well as suppress certain concrete model inputs so that the behaviours of the concrete model not required by the abstract model are excluded. For example, the concrete model implements a functionality of suspending infusion in response to an Infusion_Pause command from the clinician. However the abstract model does not specify this functionality. Hence the interface suppresses the Infusion_Pause command to the device. Similarly other inputs and system conditions of the concrete model that are not specified by the abstract model are abstracted by the interfaces.

3.1.2 GPCA Software Architecture

As mentioned earlier, we reused an existing model of the software (GPCA_SW). The software component (GPCA_SW) was by far the most complicated. To support intellectual control of the software, it was further decomposed into subcomponents as shown in Figure 5. To remove clutter from the figure, we do not show connections from the sub-components to the component boundary: the inputs and outputs of the subsystems are connected to the import and output of the system with the same name. This decomposition of the software was specified as an implementation of the GPCA_SW component similar to the decomposition of PCA_Pump. For details of the model, we refer the reader to [22].

3.2 Behavioral Modeling

The AADL architecture is structural: it captures the components and their connections, but does not specify functional behavior. To capture the controller behavior, we used the popular MathWorks MATLAB tool with the Simulink [18]/Stateflow [20] notations. This tool is widely used for critical software development in avionics and automotive software and is increasingly used for medical device software. Simulink is a graphical data flow language for modeling and simulating dynamic systems (both the language and the tool are referred to as Simulink). Stateflow is a state-based notation used to model state machines and flow charts. These individual behavioral models allowed us to analyze and verify the behaviors in a manageable fashion. For details of the model, we again refer the reader to [22].

4. VERIFICATION

Our strategy for verifying properties over models that span multiple abstraction levels is depicted in Figure 6 using the closed-loop infusion system as the example. The main idea is to use the notation and tools appropriate for each abstraction level, decompose the single verification problem into distinct verification tasks for each level of abstraction, and finally tie together the results in a logical fashion that is amenable to manual review.

4.1 Verification Approach and Tools
For the verification effort of the GPCA model, we used three different model checking tools. For the timed automata model verification, we used the UP-PAAL model checker. For the architectural verification, we used the recently developed JKind tool\footnote{Available at: https://github.com/agacek/jkind} from Rockwell Collins. For the behavioral model verification, we used the Simulink Design Verifier (SLDV)\cite{19}. Both the JKind and SLDV tools use k-induction\cite{27} algorithms implemented on top of a Satisfiability Modulo Theories (SMT) solver\cite{12} to reason about infinite-state models involving real (rational) numbers and bounded or unbounded integers.

We proved two essential closed-loop safety properties. The first one states that, once the patient’s blood oxygen saturation level becomes lower than 90%, the pump will eventually be stopped, expressed as an UP-PAAL query as $\text{samplebuffer} < 90 \rightarrow \text{PCAbasal.stopped}$. Here, $\text{samplebuffer}$ is a shared variable that represents the true oxygen saturation. Note that the invariant of the $\text{stopped}$ state in the $\text{PCAbasal}$ automaton is $\text{pca_rate}==0$, implying that no drug is entering the patient. The second property takes into account the fact that the stopped pump can be restarted manually by the caregiver. We show that if the caregiver never restarts the pump when the patient is in danger, the patient eventually recovers, with the oxygen saturation levels returning to normal. This property is expressed as an UP-PAAL query as $\text{samplebuffer} < 90 \rightarrow \text{samplebuffer} >= 91$.

\subsection{Architectural Model Verification}

AADL is supported by a growing number of tools, including tools that support editing and import/export of AADL models, as well as tools that allow one to analyze different aspects of the model—correctness of the connections, component resource usage within limits, etc. However, AADL does not have a built-in means of associating requirements with different components within the architecture, nor does it have support for reasoning about requirements.

To formally argue that the system satisfies its requirements, assume-guarantee contracts\cite{21} provide an appropriate mechanism for capturing the information needed from other modeling domains to reason about system-level properties. In this formulation, guarantees correspond to component requirements, and assumptions correspond to the environmental constraints.
that are used in verifying the component requirements. A contract specifies precisely the information that is needed to reason about the component’s interaction with other parts of the system. Furthermore, the contract mechanism supports a hierarchical decomposition of the verification process that follows the natural hierarchy in the system model.

In our work, we use AGREE [8] – a compositional reasoning framework based on assume-guarantee contracts. AGREE is a plugin to the OSATE AADL tool and adds support for requirements capture and formal verification of the architectural models. In AGREE, we use the past-time operator subset of *past-time linear temporal logic* (PLTL) [15] to specify contracts.

AADL distinguishes between a system, which describes the input/output interface of an AADL aggregate, and *system implementations*, which describe the internal structure of the system. Each system type may have several implementations. We define requirements contracts in a system because requirements are defined over the input/output interface of the component and should not be defined in terms of implementation details. However, we perform proofs at the *system implementation* level, where we can use the contracts of subcomponents and their architectural relationship to establish system level properties.

For each layer of the architecture, we establish, for each implementation of a system, that the implementation meets the requirements of the system defined in the layer. Transitivity, we thus establish that the requirements of the top-level system are proved given that the properties of the lowest layer leaf-level components are true. The structure of contracts is the same for the subcomponents, though of course the interfaces and properties are specialized to the functionality of each subcomponent.

In order to formally link the closed loop PCA to the architectural model, we recaptured the closed loop PCA properties as AGREE contracts at the PCA_Pump level in AADL. Let us take an example to explain the hierarchical steps of verification. A closed loop level property for the PCA stated informally is, *Infusion cannot begin until parameters of the infusion are configured and pump is started by the caregiver*. This requirement recaptured as AGREE contract is,

\[
\text{property no_infusion_start } = \\
\text{true } \implies \text{pre(PUMP\_DISPLAY\_OUT.Current\_System\_Mode)=1} \\
\text{and (not(CAREGIVER\_IN.Infusion\_Programmed)) } \implies \\
\text{(PUMP\_DISPLAY\_OUT.Current\_System\_Mode = 1)};
\]

In the above formulation, the *pre* of the variable *Current\_System\_Mode* represents its value from the previous step. The *Current\_System\_Mode = 1* represents the pump state (abstractly represented as mode 1) in which flow rate is zero. The astute reader might have noticed the "true" at the beginning of the property, which effectively makes the property true in the first step. This is intentional, for otherwise the previous value of the variable at the very first step could be an arbitrary value that causes some avoidable complications. To keep the specifications straightforward, we defined properties such that, we specify that the pump always starts in a state at which flow is zero and then onwards the above mentioned property holds.

In order for the above property of PCA_Pump to hold, its components should compositionally guarantee using their properties. The GPCA_SW being a component of PCA_Pump has much richer functionality than that specified by PCA_Pump. Hence state mapping between them was required to write GPCA_SW properties that would guarantee PCA_Pump properties. Figure 7 shows a high level representation of the states and its mapping between the systems. The state representation of PCA_Pump was derived from the UPPAAL model and that of GPCA_SW was derived from its behavioral specifications. Similar to the state mapping, the transitions from and to these states were also mapped. We discuss the details in a subsequent section.

In order to satisfy the PCA_Pump property, certain assumptions were required at the PCA_Pump level to match the abstractions. First of all, CAREGIVER_IN.Infusion_Programmed - specifies if a valid prescription is already programmed in the pump - was mapped to the concrete model variable (PUMPgetDisplay.OUT.Configured > 0) that means the device holds a valid prescription. At the PCA_Pump level, these were still abstract concepts since the caregiver decides if the infusion is programmed by looking at the pumps’s display and determining if it is configured. Hence we state this as an assumption at the PCA_Pump level.

Secondly, we mapped the (ON->Programmed) and Stopped of the abstract model to IDLE state of the concrete model hence the transitions (ON->Programmed)->Running triggered by (CAREGIVER_IN.Infusion_Start and CAREGIVER_IN.Infusion_Programmed) and the transition Stopped -> Running triggered by (CAREGIVER_IN.Clear_Command) were assumed to imply each other.

Thirdly, GPCA_IP_Interface maps the abstract model’s Infusion_Initiate input signal with the concrete model’s Infusion_Initiate command. Finally the GPCA_SW guarantees that,

\[
\text{property no_enter_therapy } = \\
\text{true } \implies \text{pre(GPCA\_SW.OUT.Current\_System\_Mode) = 1} \\
\text{and not(OP\_CMD\_IN.Infusion_Initiate)} \\
\text{and GPCA\_SW.OUT.Configured > 0 } \implies \\
\text{(GPCA\_SW.OUT.Current\_System\_Mode = 1)};
\]

There were a few other component guarantees that
deal with mapping inputs and outputs that are not discussed here since its not a significant point of discussion. Hence when we compositionally verified the PCA_Pump property in AGREE, it was guaranteed using their component guarantees and system assumptions.

At this point, the satisfaction of the properties were only until the GPCA Device components level. As discussed in section 3.1.2 the GPCA_SW was further decomposed into components. Hence the properties guaranteed by the GPCA_SW for the PCA_Pump has to be guaranteed by its sub-components. We performed another level of AGREE reasoning in which now the GPCA_SW is the top level system that has properties to be guaranteed (similar to the PCA_Pump) by its components guarantees such as TLM, ALARM, IM. For example, the property of the GPCA_SW (property no_enter_therapy), is guaranteed by Infusion Manager(IM), Configuration Manager (CONFIG), and Output (OUTPUT) component guarantees. There were not as many assumptions and abstraction mapping required for the property guarantee between the GPCA_SW and its components since they were already done as a part of another initiative.

4.4 Behavioral Model Verification

The required properties of the software components captured in AGREE used to guarantee the GPCA_SW, which in turn were used to guarantee the PCA_Pump. However, in order to verify if the software components captured in AGREE indeed hold, we used the detailed behavioral model of each of these components built using Simulink/Stateflow and verified using Simulink Design verifier that they satisfy their properties. The Simulink Design Verifier requires all properties to be specified as Boolean expressions in one of the available MATLAB notations. Hence in our work, we recapture the required component properties in embedded MATLAB - a subset of the MATLAB computing language that supports efficient code generation for deployment in embedded systems. For more details on the behavioral modeling verification, we refer the reader to [22].

5. DISCUSSION

In total, 7 PCA_Pump properties were proved in AGREE that correspond to the abstract timed automata model of the PCA. In order to guarantee these 7 system properties, there were 7 GPCA_IP_Interface properties and 11 GPCA_SW properties along with some auxiliary properties of the GPCA_HW_Sensors, GPCA_HW_Actuators and GPCA_OP_Interface to route input and output signals. Similarly the 11 GPCA_SW properties were in turn guaranteed by 4 ALARM, 11 IM and 6 CONFIG component properties.

5.1 Matching Abstractions

To ensure that the concrete GPCA works correctly in the closed-loop instance, it is necessary to show that the infusion pump in the closed loop model simulates the concrete GPCA. We are currently extending AGREE tools to support a machine-checked proof of simulation. In this section, we discuss how we relate the abstract and concrete models and our (currently informal) check that the relation is a simulation.
5.1.1 Defining Mappings

To create a simulation relation, one must match the variables and states in the abstract model with those in the concrete model. The concrete GPCA model has many inputs, states and transitions that are not represented in the abstract model. We addressed this by associating concrete infusion modes with abstract states of the infusion pump, introducing a non-deterministic input in the abstract model, and finally adding environmental assumptions in the concrete model to prevent it from entering states that do not preserve the mapping.

First we established a mapping between states in the abstract model of the infusion pump and the infusion modes of the concrete GPCA model, as shown in Figure 7. This enables restating properties specified in terms of flow-rate constraints at the abstract level as properties in terms of the infusion modes. The different GPCA infusion modes have specific associated flow-rates that can be immediately verified for conformance to flow-rate constraints. In this mapping, the abstract states Stopped, ON and Programmed are associated with the IDLE mode represented by the predicate Current_System_Mode = 1; Running is associated with BASAL and OTHER BOLUS modes represented by Current_System_Mode $\in \{2,3\}$; and BOLUSING is associated with PATIENT BOLUS represented by Current_System_Mode = 4. Note that there is no abstract state associated with the PAUSED mode of the GPCA, which denotes a temporary halt to infusion due to certain exceptional conditions identified by the alarm sub-system. Environmental assumptions, to be discussed shortly, guarantee that this mode is not reached for the scenarios considered.

Second, the abstract closed-loop system modeled the “stop” command to the pump as an exclusive input from the supervisory control for oxygenation-level based safety-interlock. However, realistically the operator may command the GPCA to stop at any time – one may view the operator as an external monitor for exceptional conditions, analogous to the internal monitor and not stopped, the system infuses at BASAL flow rate.

5.1.2 Establishing Simulation

The GPCA monitors various exceptional conditions as modeled by the ALARMS component and responds appropriately, some of which would include a temporary pause to infusion. These alarms may be triggered by sensor inputs such as air-in-line or internally computed conditions such as empty-reservoir. Thus, even a simple abstract property, Once the pump is started, unless it is stopped, the drug flows at least at basal flow rate, does not hold for the concrete model, because, the property presupposes the absence of infusion-ending triggers other than the stop command. In the closed-loop system, the abstract infusion pump model rightly disregards such details: at that level, one is interested in establishing essential system properties using a simplified notion of an infusion pump. The responsibility for stopping infusion for any reason can be pushed out to the environment – in this case the pump operator can be assumed to command “stop” as necessary. Therefore, when relating abstract infusion pump properties to a concrete realization of the pump, it is reasonable to interpret those as being prefixed with a standard caveat: “Absent exceptional pump conditions that internally trigger stoppage of infusion …". In other words, we approximate each internal system failure as an external command to stop the pump. Figure 8 shows a snapshot of assumptions and guarantees that formally establishes the described abstraction mapping. Of course, these exceptional conditions could vary based on the specific pump under consideration. In the process of mapping the abstract infusion pump model and its properties to the concrete GPCA, absence of these internally triggered exceptional conditions is captured as a system assumption: no_cancel_implies_no_stop_conditions. Such assumptions must then be discharged using properties of the environment in which the GPCA operates. In the present case, one has to manually review these assumptions to ascertain that those are acceptable caveats to the property of interest. This is a desirable outcome, for it clearly calls attention to what those exceptional conditions could be for the specific GPCA device.

Finally, in the concrete model, there are more modes of active drug infusion than the ones specified in the abstract model as shown in Figure 7. Originally the abstract model’s property was, when the system is started and not stopped, the system infuses at BASAL flow rate (when there is no patient bolus request). But the concrete model switches between BASAL and OTHER BOLUS based on the prescription. There were multiple ways to reconcile the differences. A straightforward approach is to assume that there is no prescribed OTHER BOLUS. However, this is not a reasonable assumption for the scenarios under consideration – the essential system properties must hold for any infusion scenario. Therefore, we adopted an alternative route: the abstract property was modified to state that system infuses at no less than the BASAL flow rate". With an assumption that any bolus infusion delivers drug at a rate higher than the BASAL
-- UPPAAL Property : if the pump is not stopped, it delivers medication with at least the basal rate
property infusion_continue =
  true -> (pre(PUMP_DISPLAY_OUT.Current_System_Mode) >= 2 and pre(PUMP_DISPLAY_OUT.Current_System_Mode) <= 4) and not(PUMP_CMDS_IN.Pump_Stop) =>
(PUMP_DISPLAY_OUT.Current_System_Mode >= 2 and PUMP_DISPLAY_OUT.Current_System_Mode <= 4);
guarantee "infusion_continue":infusion_continue;

-- Map modes with flow rates.
guarantee "Mode 1":(PUMP_DISPLAY_OUT.Current_System_Mode = 1) => (DRUG_OUT.Drug_Flow_Rate = 0);
guarantee "Mode 2":(PUMP_DISPLAY_OUT.Current_System_Mode = 2) =>
(DRUG_OUT.Drug_Flow_Rate => CAREGIVER_IN.Normal_Infusion_Rate);
guarantee "Mode 3":(PUMP_DISPLAY_OUT.Current_System_Mode = 3) =>
(DRUG_OUT.Drug_Flow_Rate => CAREGIVER_IN.Normal_Infusion_Rate);
guarantee "Mode 4":(PUMP_DISPLAY_OUT.Current_System_Mode = 4) =>
(DRUG_OUT.Drug_Flow_Rate = CAREGIVER_IN.Bolus_Infusion_Rate);

-- sensor conditions that cause infusion to be stopped.
eq sensor_conditions_that_cause_pause:bool =
  (SENSOR_OUT.Battery_Depleted or SENSOR_OUT_RTC_In_Error or SENSOR_OUT.CPU_In_Error or
  SENSOR_OUT.Memory_Corrupted or SENSOR_OUT.Pump_Too_Hot or SENSOR_OUT.Watchdog_Interrupted or
  SENSOR_OUT.Temp or SENSOR_OUT.Humidity or SENSOR_OUT.Air_Pressure or
  SENSOR_OUT.Air_In_Line or SENSOR_OUT.Occlusion or SENSOR_OUT.Inactive.Open);

-- when there is no infusion cancel, then there is no sensor conditions causing infusion stop occurs.
property no_cancel_implies_no_stop_conditions =
  not(OP_CMD_IN.Infusion_Cancel) => not(sensor_conditions_that_cause_pause);
assume "no_cancel_implies_no_stop_conditions": no_cancel_implies_no_stop_conditions;

-- If there is no infusion cancel then there is no condition stopping infusion.
property no_cancel_implies_no_stop_conditions =
  not(OP_CMD_IN.Infusion_Cancel) => not(infusion_end_conditions or any_alarms or OP_CMD_IN.Infusion_Inhibit);
assume "no_cancel_implies_no_stop_conditions": no_cancel_implies_no_stop_conditions;

-- If the system is in ACTIVE, and if there is no conditions to stop infusion, system will be in ACTIVE.
property in_active =
  true -> (pre(GPCA_SW_OUT.Current_System_Mode) >= 2 and pre(GPCA_SW_OUT.Current_System_Mode) <= 4) and
  not(OP_CMD_IN.Infusion_Cancel) =>
  (GPCA_SW_OUT.Current_System_Mode >= 2 and GPCA_SW_OUT.Current_System_Mode <= 4);
guarantee "in_active": in_active;

Figure 8: Portions of AGREE properties at different layers of system abstraction

flow rate, which is typically true of the problem domain, this change allows the possibility of multiple drug delivery modes with different flow rates even when the abstraction includes only a few specific modes explicitly.

Apart from these changes, some auxiliary properties were required to guarantee that the concrete model’s states and transitions that were not represented by the abstract model are not reachable given the system assumptions and the abstraction mapping. For example, the concrete model has an additional state PAUSED shown in Figure 7. Thus, an auxiliary property to show that PAUSED is never reached under the given environmental assumptions is needed.

In summary, the goal of this process is to justify the following claims:

1. Every abstract state, input and output in the abstract infusion pump model has some matching counterparts in the concrete model.
2. States, inputs and outputs of interest in the concrete GPCA have some matching counterparts in the abstract model.
3. Transitions between such matched states on matched inputs in the concrete GPCA model have a corresponding matching transition in the abstract model and the respective outputs match.
4. Unmatched concrete inputs do not occur given certain environmental assumptions.
5. Unmatched concrete states are not reachable under those environmental assumptions.

Taken together these claims, informally, allow us to see the abstract infusion pump model as a stand-in (simulation) for the concrete GPCA model under the given
environmental assumptions. Thus safety properties that are established for a system model that uses the abstract infusion pump as a component, in a way that satisfies the environmental assumptions made in the matching process, will be upheld when the concrete GPCA infusion pump is substituted in place of the abstract pump.

5.2 Limitations

Our current approach to verification is restricted in several ways. First, AGREE currently only handles synchronous architectural models in which execution proceeds in a deterministic discrete sequence of steps. Second, AGREE can verify only invariants, so liveness properties, cannot be specified in AGREE. In our experience, this is not as severe a limitation as it may seem, since most systems are concerned with bounded liveness in which an action must occur within a time interval that can be written in AGREE.

The current analysis tools use rationals to model the behavior of real numbers; However, most software is implemented using floating point numbers. This can lead to unsoundness in our analysis of software that uses floating point arithmetic. Also, AGREE does not support trigonometric or non-linear functions. These can be approximated in some cases, but many of the interesting numeric properties of systems simply cannot be specified.

6. RELATED WORK

Modeling and analysis of closed-loop medical systems have been primarily studied in the context of diabetes care. Much attention is given to modeling patient physiology and design of algorithms for glucose control; see, for example, \cite{29} \cite{7}. However, we are not aware of any studies, where evaluation of the closed-loop system is tied to the modeling of medical devices that comprise the system. A closed-loop safety interlock for PCA infusion, similar to the one studied in this paper, has been proposed in \cite{9}, however, the authors do not show any analysis results. Similarly, the GPCA pump has been used in a number of case studies that involved a variety of formal methods to model different aspects of the pump behavior. In \cite{16}, code for a simple infusion controller has been generated from UPPAAL-verified code.

In the context of multi-formalism modeling, frameworks have been developed to integrate models specified using different formalisms. The Möbius approach \cite{11} provides a comprehensive infrastructure to support interacting formalisms and solvers. Metropolis \cite{2} provides a meta-model with formal semantics to capture and analyze electronic system designs at multiple levels of abstractions. The Ptolemy project \cite{5} seeks to address the problem of heterogeneous mixtures of models of computation for modeling and analyzing CPS. Synchronous language principles are exploited for hierarchical composition \cite{17}, a strategy also adopted here. Similarly, the OsMoSys approach \cite{28} and AToM³ \cite{10}, require the formalism (graph based) to be defined in terms of an XML based meta-language. The main drawback of these approaches is requiring the formalism to be definable/mappable using common semantics/rules which may restrict the freedom of the designer to models using only those notations the framework. Rajhans and Krogh \cite{25} derive, using behavioral semantics, a set of general conditions for compositionality across heterogeneous abstractions. In contrast, the present work provides ad hoc compositionality for the specific formalisms used in existing models of the GPCA system.

A crucial aspect of relating models in different formalisms is the notion of time. Timed automata used to model the closed-loop system use continuous time, while behaviors of the AADL model rely on discrete time. The usual way of reconciling the discrepancy between the notions of time is through discretization. A number of techniques for discretization have been developed in the literature, such as \cite{13} \cite{14}. Our UPPAAL model satisfies the conditions put forth in \cite{23}, which guarantee that whenever a transition is enabled during an execution, it can occur at an integer-valued time instance. Thus, the model can be faithfully simulated in discrete time.

7. CONCLUSION

When systems are composed from sub-systems, properties of the system must be assessed based on its component properties and the composition. To reason about such systems, no single analysis and modeling method can successfully cope with all aspects of a system or its components. Hence multiple notations and formalisms are used. An approach to logically glue the diverse analysis of the system and its components is required to reason about the system properties.

In this paper, we considered compositional verification of a medical system at multiple levels of abstraction, with different formalisms used at each level. We were able to formally glue distinct verification paradigms by leveraging the system’s hierarchical architectural decomposition. We showed how properties proved for the system components at the lower levels of abstraction can be used to validate the more abstract models, ensuring that properties proved at the higher levels of abstraction remain satisfied.

While techniques used in this paper are specific to the example at hand and to the formalisms used within the case study, we believe that this work can form the basis for a general, scalable and practical approach to layered verification of properties in complex cyber-physical systems. In order to fully realize the promise of this ap-
proach, we are currently working to make the approach more systematic and eliminate the ad hoc aspects we used in this work. Specifically, we are working to extend the AGREE tools to automate the verification of user-defined simulation relation between two AADL components. This would then enable one to even mechanically translate the abstract model from a different formalism into AADL and then define and verify the simulation relation formally using AGREE. Finally, we are also working to make the approach more general, allowing more formalisms to be incorporated.

8. ACKNOWLEDGMENTS
We thank Andrew Gacek, Darren Cofer and John Backes at Rockwell Collins for developing and improving JKind and AGREE. We also thank Miroslav Pajic for helpful discussions on restructuring the model of the closed-loop system.

9. REFERENCES


