# Compositional Verification of a Medical Device System*

Anitha Murugesan
Department of Computer Science and Engineering
University of Minnesota
200 Union Street,
Minneapolis, Minnesota 55455
anitha@cs.umn.edu

Michael W. Whalen
Department of Computer Science and Engineering
University of Minnesota
200 Union Street,
Minneapolis, Minnesota 55455
whalen@cs.umn.edu

Sanjai Rayadurgam
Department of Computer Science and Engineering
University of Minnesota
200 Union Street,
Minneapolis, Minnesota 55455
rsanjai@cs.umn.edu

Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
200 Union Street,
Minneapolis, Minnesota 55455
heimdahl@cs.umn.edu

## ABSTRACT

Complex systems are by necessity hierarchically organized. Decomposition into subsystems allows for intellectual control, as well as enabling different subsystems to be created by distinct teams. This decomposition affects both requirements and architecture. The architecture describes the structure and this affects how requirements "flow down" to each subsystem. Moreover, discoveries in the design process may affect the requirements. Demonstrating that a complex system satisfies its requirements when the subsystems are composed is a challenging problem.

In this paper, we present a medical device case example where we apply an iterative approach to architecture and verification based on software architectural models. We represent the hierarchical composition of the system in the Architecture Analysis & Design Language (AADL), and use an extension to the AADL language to describe the requirements at different levels of abstraction for compositional verification. The component-level behavior for the model is described in Simulink/Stateflow. We assemble proofs of system level properties by using the Simulink Design Verifier to establish component-level properties and an open-source plug-in for the OSATE AADL environment to perform the compositional verification of the architecture. This combination of verification tools allows us to iteratively explore design and verification of detailed behavioral models, and to scale formal analysis to large software systems.

## Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies - Requirements flow down*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods; Model checking*

## Keywords

Compositional Verification; System Decomposition; Cyber Physical Systems

## 1. INTRODUCTION

Software is ubiquitous in safety-critical systems, which have the potential to cause loss of life, injury, or other serious damage to property and environment. The size and complexity of this software continues to grow, making it ever more difficult to capture the correct requirements, design the software correctly, and verify to a high level of confidence that we have the right requirements and that the software indeed satisfies those requirements. To make design and construction possible, such a complex system is typically organized as a composition of subsystems that can themselves be further decomposed if necessary. This hierarchical aspect of design is of crucial importance; it allows the complexity of the entire system to be managed through partitioning and abstraction.

One question related to this decomposition of a system is whether a constraint is considered a requirement or a design decision. The answer depends largely on one's perspective: design decisions at one level of abstraction naturally become requirements on the next lower level of abstraction. This dichotomy illustrates the natural interplay between system/software architecture and requirements refinement. In research and in practice, however, software architecture and software requirements tend to be quite distinct, supported by different research communities, tools and techniques. Based on Nuseibeh's TwinPeaks idea [31], we advocate a close and iterative relationship between software requirements and architecture [36]. Given adequate tools,

---

*This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

*HILT'13,* November 12–14, 2013, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/2527269.2527272 .

this approach allows quick iterations between requirements and design, and supports efficient verification of the *adequacy* of the decomposition, i.e., the requirements allocated to the subsystems and their architectural connections imply the requirements at the next higher level of abstraction.

In this paper, we describe an application of this approach to the control software of a medical device—a Patient Controlled Analgesia Infusion Pump. We model the system architecture in the Architectural Analysis & Design Language (AADL) [33] and the component level behavior in the Simulink and Stateflow languages [24, 25]. For requirements, we start from textual system requirements expressed in natural language (English). We formalize these requirements using an extension of the AADL language that supports specification of formal textual requirements for systems at different levels of the system hierarchy within the AADL model. Currently our extension allows specification of temporal logic invariants using a structuring mechanism similar to the Property Specification Language (PSL) [16]. We then use the AGREE framework [5]—a compositional verification framework developed for AADL verification by Rockwell Collins and University of Minnesota,—to prove that system requirements are established, given the architectural structure of the system and the requirements allocated to sub-systems in the architecture.

The sample system used in this paper is a medical device—a Generic Patient Controlled Analgesia (GPCA) infusion pump system [1]. Infusion pumps are medical devices used to accurately infuse liquids into a patient's bloodstream. Medical devices, such as infusion pumps, are suitable systems to explore since they are generally safety-critical and the maturity level of the V&V process has often been insufficient to ensure the safety and overall quality of the devices. Infusion pumps have been involved in numerous 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 pro-actively increase the safety of these devices by establishing additional regulatory requirements for infusion pump manufacturers. In this context, the research community—in collaboration with the FDA—is exploring various methods to improve the safety of infusion pump systems. Our aim is to contribute to this initiative by building a powerful and scalable proof framework and evaluate its effectiveness by applying it to various medical devices.

We have modeled the GPCA architecture using AADL and the behavior of the architectural components in Simulink and Stateflow. We are in the process of formalizing and proving the GPCA system and software requirements. We have currently formalized and proved a significant fraction (about 30%) of the top-level software requirements using compositional verification. We expect to complete the formalization and verification of the remainder in the near future (we are aware of no technical hurdles; it is simply a matter of time). The component-level proofs and architectural proofs are established efficiently: our component level models require ≈4 minutes to prove and the architectural proofs are in the order of 2 seconds.

In any development effort, we expect that requirements, architectures, and components will co-evolve as the project progresses. Requirements naturally influence the architecture and design, architectural consideration may expose the need for new or modified requirements, and the verification efforts are likely to reveal flaws in the requirements as well as the architecture and design [28]. We also anticipate that verification of different parts may involve formal evidence (for example, proof) as well as empirical and analytic evidence (for example, testing, inspections, etc.). Given appropriate tools, it may be possible to perform top-to-bottom system level formal proofs; however, the approach is designed to support selective proofs for the portions of the system that are most critical, or that can be easily addressed with automated tools. Finally, we have attempted to use established notations for component-level behavior (Simulink and Stateflow) and architectural description (AADL) supported by commercial or open-source tools. Our hope is to demonstrate that this approach is a reasonable and cost-effective *engineering* solution for construction of safety-critical systems; an approach that could be readily adopted into industrial practice.

## 2. TARGET SYSTEM

Infusion pumps are medical cyber physical systems used for controlled delivery of liquid drugs into a patient's body according to a physician's prescription (the set of instructions that governs infusion rates for a medication). These pumps may be classified into various kinds depending on their features, construction, and usage. Patient-Controlled Analgesia (PCA) pumps are generally equipped with a feature that allows patients to self-administer a controlled amount of drug (a patient-bolus), typically a pain medication.

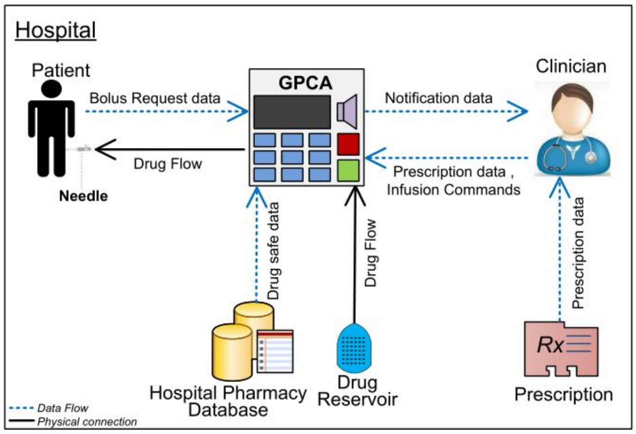


**Figure 1: Environment—GPCA System Overview**

Figure 1 shows an external intravenous Generic Patient Controlled Analgesia (GPCA) device in a typical usage environment, a hospital or a clinic. In an infusion system, the clinician operates the GPCA device, programs the prescription information, loads the drug, connects the device with the patient, and responds to exceptional conditions that occur during the therapy. The patient receives the medication from the device through an intravenous needle. The patient can self-administer prescribed amounts of additional drug by requesting a bolus, a request usually done by pressing a bolus request button accessible at the patient's bed. The hospital pharmacy database is a repository that stores manufacturer provided drug information (for example, upper limits on infusion rates for a specific drug).

In short, the GPCA system has three primary functions (1) deliver the drug based on the prescribed schedule and patient requests, (2) prevent hazards that may arise during its usage, and (3) monitor and notify the clinician of any exceptional conditions encountered. In this paper we will focus our attention on the architecture and behavior of the software portion of the overall GPCA system.

## 3. ARCHITECTURE AND REQUIREMENTS

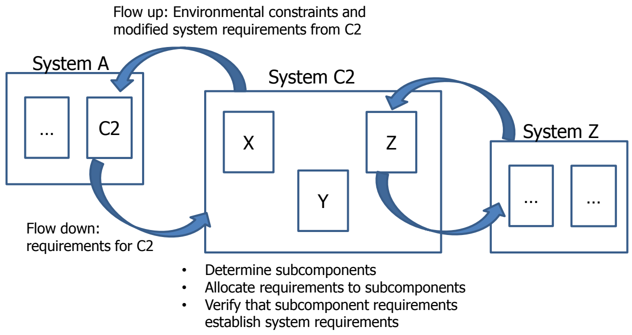

Flow up: Environmental constraints and modified system requirements from C2
System A
...
C2
System C2
X
Z
Y
System Z
...
...
Flow down: requirements for C2
Determine subcomponents
Allocate requirements to subcomponents
Verify that subcomponent requirements establish system requirements

**Figure 2: Requirements and Architectural Decomposition**

Once systems become sufficiently complex, they are decomposed into subsystems that are created by several distinct teams. Thus, the requirements on the system as a whole must be decomposed and allocated to each of the subsystems. This decomposition touches both requirements and architecture, since the architecture describes the structure of the decomposition, and this will affect how requirements "flow down" to each subsystem. We believe that requirements should be organized into hierarchies that follow the architectural decomposition of the system because the act of decomposing a system into components induces a requirements analysis effort in which we need to ascertain whether the requirements allocated to subcomponents in the architecture are sufficient to establish the system-level requirements. Equally importantly, we need to determine whether any assumptions on a component's environment made when allocating requirements to that component can be established. This is shown informally in Figure 2. As we begin to allocate requirements to components, we may find that the architecture we have chosen simply cannot satisfy the system-level requirements. This may cause us to re-architect the system to allow us to meet the system level requirement, levy additional constraints on the external environment, or renegotiate the system-level requirement [36].

The GPCA is a physical device that contains an infusion pump, a user interface containing an input panel as well as audio and visual alarms, a variety of sensors related to the current status of the device, and a microcontroller containing software to control the device. For the software architecture, we have chosen to largely mimic the structure of the physical system. Thus, the major sub-systems include (1) Alarm—responsible for monitoring exceptional conditions and raising alerts to avoid hazards to the patient, (2) Infusion—responsible for determining the current mode of the system and commanding the flow of drug out of the device, (3) Mode—responsible for managing the top-level operating mode of the system, and (4) Logging—responsible for logging the status of the device. These subsystems are shown in Figure 3. As we are currently focusing on the software controller, we are not yet describing the user-interface portion of the GPCA software.

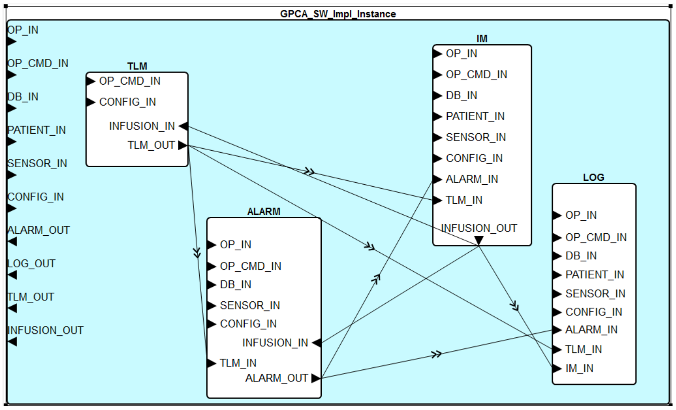


**Figure 3: Software architecture of the GPCA controller**

To illustrate the architectural decomposition with respect to the software requirements for the GPCA pump, consider the following system level requirement:

> *When performing infusion, if the remaining volume of drug in the reservoir drops below the empty-drug-threshold, the GPCA Pump shall raise visual and aural alarms and stop infusion.*

When allocated to software, this requirement addresses the inputs and outputs of the software (as opposed to the physical phenomena addressed in the system requirement):

> *When in the infusing-mode, if the estimated drug remaining in the drug reservoir drops below the empty drug threshold (estimated-drug-remaining $<$ empty-drug-threshold), the GPCA software shall issue the visual-alarm and aural-alarm commands, and stop the infusion.*

Since the particular infusion pump we are modeling does not measure the volume of drug infused, the remaining drug volume is estimated by subtracting the estimated volume of drug infused from the initial volume of the drug contained in the reservoir.

The software requirement is further decomposed and allocated to the *Alarms* (ALARM in Figure 3) and *Infusion Manager* (IM) components. Here, the solution is to require the Alarms component to monitor the estimated remaining drug and—when it drops below the threshold—raise a critical alarm:

> *When in the infusing-mode, if the estimated drug remaining in the drug reservoir is below the empty drug threshold (estimated-drug-remaining $<$ empty-drug-threshold), the Alarms subsystem shall set the highest-level-alarm to 4 (critical alarm) and set the empty-drug-alarm indicator in current-alarm.*

The Alarms subsystem defines four levels of severity from level 1 (informational) to level 4 (critical). These levels are used by the rest of the system to determine correct infusing and logging behavior. The Infusion Manager component is responsible for receiving the notification from the Alarm component and signaling the hardware to stop infusion by commanding the flow rate to zero:

> *The infusion manager shall stop infusion whenever a critical alarm occurs (highest-level-alarm=4)*

In addition, the Infusion Manager component is responsible for estimating the remaining reservoir volume. If the sub-component requirements are adequate, it will be possible to demonstrate that the higher level software requirement is actually met by our design.

In the full system, there are dozens of requirements allocated to software relating to the correct behavior of the system. The requirements involve correct diagnosis of sensor and actuator failures, tolerances for infusion, logging, self test, and many other aspects. An interested reader can examine the full GPCA requirements at the following web site: http://crisys.cs.umn.edu/gpca.shtml.

## 3.1 Architectural Modeling and AADL

In order to document, visualize, and analyze the architecture of the GPCA system, we need to model it. When modeling embedded safety-critical systems such as the GPCA, it is desirable to have an architectural model that supports descriptions of both hardware and software components and their interactions. We need to document component interfaces, interconnections between components, and requirements on components, without describing the implementations of those components. At the leaf level, component implementations are defined separately using model-based development tools or by traditional programming languages, as appropriate.

The Architecture Analysis and Description Language (AADL) is a notation that suits these needs. AADL supports many of the constructs needed to model embedded systems such as processes, threads, devices (sensors and actuators), processors, buses, and memory. Furthermore, it contains an extension mechanism (called an *annex*) that can be used to extend the language to support additional features, such as requirements modeling. AADL, now an SAE standard [33], is a textual language that can be expressed graphically and is accompanied by a UML profile. AADL includes constructs that describe both software and hardware components, as well as mapping software components to physical resources and the devices with which they communicate. It allows for specification of interfaces for flow of control and data. The basic building block of this notation is a component, defined by its category (hardware, software, or composite), type (how the component interacts with the outside world), and its implementation (an instance of the component type). Note that there can be many instances of one component type. For example, highly available systems often have redundant computing resources to support failover; these can be represented as instances of a single component type. Our current GPCA example does not have redundant processing elements, but these may be added in the future.

The graphical representation in AADL of the GPCA software architecture is shown in Figure 3. The GPCA_SW_Impl_Instance describes an instantiation of the GPCA_SW system. In the GPCA model, inputs and outputs are defined along the left side of the figure. We group the inputs from different sources: operator inputs (OP_IN), operator commands (OP_CMD_IN), drug database inputs (DB_IN), patient inputs (PATIENT_IN), sensor inputs (SENSOR_IN), and system configuration inputs (CONFIG_IN), in order to simplify signal routing throughout the model. In the current model, these six input sources contain 76 different scalar signals. Connections between components can be *immediate* (visualized by lines in the diagram containing the >> symbol) or *delayed* (visualized by "plain" lines). The designation determines whether communication between components will happen in the same time frame or delayed by one time frame; immediate connections induce data-dependency constraints on scheduling of components. Given a deterministic single-processor system, these correspond to immediate connections and $1/z$-delayed connections in Simulink/Stateflow. To remove clutter from the figure, we do not show connections from the subsystems to the system boundary; the inputs and outputs of the subsystems are connected to the inports and outports of the system with the same name.

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 reasoning about requirements. The AGREE framework addresses these issues by adding support for requirements capture and formal verification (described in more detail in the next section) to the OSATE AADL tool.

## 3.2 Reasoning about Architectural Models with AGREE

To convincingly argue that a system has the desired effect in its environment (the system satisfies its requirements), Hammond et al. developed the notion of a Satisfaction Argument, based on Jackson and Zave's World and the Machine model [13, 17]. This approach attempts to establish that system requirements hold through an argument involving (i) the specification of the system behavior and (ii) assumptions about the domain of the system.

To formalize satisfaction arguments, assume-guarantee contracts [27] 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. For formally verified components, assumptions are assertions or invariants on component inputs that are used in the proof process. 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 framework, we use the past-time operator subset of *past-time linear temporal logic* (PLTL) [19]. Temporal logics like PLTL include operators for reasoning about the

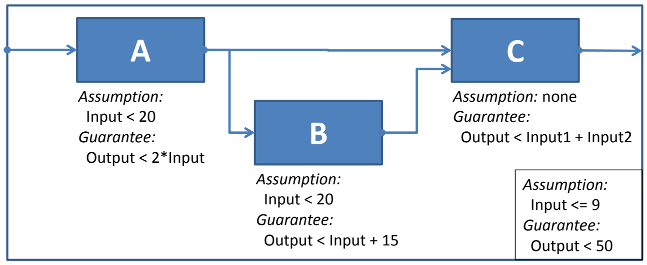


**Figure 4: Toy Architecture with Properties**

behavior of propositions over a sequence of instants in time. For example, to say that property $P$ is always true at every instant in time (i.e., it is "globally" true), one would write $G(P)$, where $G$ stands for "globally".

Figure 4 illustrates the compositional verification conditions for a toy example. In this example, we would like to establish at the system (S) level that the output signal is always less than 50, given that the input signal is less than 10. We can prove this using the assumptions and guarantees provided by the subcomponents A, B, and C that are organized hierarchically. This figure shows one layer of decomposition, but the idea generalizes to arbitrarily many layers. We want to be able to compose proofs starting from the leaf components (those whose implementation is specified outside of the architecture model) recursively through all the layers of the architecture.

The correctness obligations are the form $G(H(A) \Rightarrow P)$, which informally means that it is always the case that if assumption $A$ has been true from the beginning of the execution up until this instant ($A$ is *historically* true), then guarantee $P$ is true. For the obligation in Figure 4, our goal is to prove the formula $G(H(A_S) \Rightarrow P_S)$ given the contracted behavior $G(H(A_c) \Rightarrow P_c)$ for each component $c$ within the system. To prove the obligation, we establish generic *verification conditions* that together are sufficient to establish the goal formula. In the example, this means that for the system $S$ we want to prove that $\mathit{Output} < \mathit{50}$ assuming that $\mathit{Input} < \mathit{10}$ and the contracts for components $A$, $B$, and $C$ are satisfied. For a system with $n$ components there are $n+1$ verification conditions: one for each component and one for the system as a whole. The component verification conditions establish that the assumptions of each component are implied by the system level assumptions and the properties of its sibling components. For this system the verification conditions generated would be:

$G(H(A_S) \Rightarrow A_A)$
$G(H(A_S \wedge P_A) \Rightarrow A_B)$
$G(H(A_S \wedge P_A \wedge P_B) \Rightarrow A_C)$
$G(H(A_S \wedge P_A \wedge P_B \wedge P_C) \Rightarrow P_S)$

In general, these verification conditions may be cyclic, but if there is a delay element in the cycle we can use induction over time as in [27]. The system level verification condition shows that the system guarantees follow from the system assumptions and the properties of each subcomponent. This is essentially an expansion of the original goal, $G(H(A_S) \Rightarrow P_S)$, with the additional information obtained from each component.

### 3.3 GPCA System Architecture and Requirements in AADL/AGREE

AADL distinguishes between a *system*, which describe 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. So, in Figure 3, we are examining the implementation of the *GPCA_SW* system, which is defined in terms of *TLM, ALARM, IM*, and *LOG* systems. The structure of the subsystems is hidden; instead we prove the obligations for GPCA_SW using the contracts defined by the subsystems. 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. Transitively, 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.

An example of the notation for the AGREE framework is shown in Figure 5. The language is based on Property Specification Language (PSL) [16] and defines a Lustre language [11] "flavor" for the PSL Boolean layer expressions and definitions. Lustre is a synchronous dataflow language that describes the behavior of a system through a set of equations, and it can be viewed as a textual analogue to Simulink block diagrams. In this notation, it is possible to define constants, local variables, reusable fragments of temporal logic (called properties), and to make guarantees (called assertions) and assumptions. In specifications, we can reference values of input and output ports;additionally, we can describe stateful relationships between variables using the 'prev' expression, which provides the value of a variable from the previous step of execution of the system (the second argument to this expression provides its value in the initial state). Given this notation, it is possible to encode the requirements that we described informally in the previous section. For example, the empty reservoir requirement from the previous section is formally specified in REQ 59. 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.

## 4. BEHAVIORAL MODELING

In the work described in this paper, the detailed component behavior for the GPCA System has been modeled in Simulink and Stateflow. Simulink and Stateflow are developed by MathWorks [22]. Simulink is a data flow graphical language as well as a tool for modeling and simulating dynamic systems (both the language and the tool are generally referred to as Simulink). Stateflow is a state-based notation similar to David Harel's Statecharts notation [14] (again, Stateflow also refers to the tool). Both Simulink and Stateflow are tightly integrated in the MATLAB environment and can, as mentioned earlier, refer to other languages available in the environment. Although both Stateflow and Simulink, in our opinion, have many problems with their semantics (such as the lack of proper type systems, the event seman-

```
system GPCA_SW
  features
    OP_IN: in data port DATATYPES::Operator_Inputs.Impl;
    OP_CMD_IN: in data port DATATYPES::Operator_Commands.Impl;
    DB_IN: in data port DATATYPES::Drug_Database_Inputs.Impl;
    PATIENT_IN: in data port DATATYPES::Patient_Inputs.Impl;
    SENSOR_IN: in data port DATATYPES::Device_Sensor_Inputs.Impl;
    CONFIG_IN: in data port DATATYPES::Device_Configuration_Inputs.Impl;
    ALARM_OUT: out data port DATATYPES::Alarm_Outputs.Impl;
    LOG_OUT: out data port DATATYPES::Log_Outputs.Impl;
    TLM_OUT : out data port DATATYPES::Top_Level_Mode_Outputs.Impl;
    INFUSION_OUT: out data port DATATYPES::Infusion_Manager_Outputs.Impl;
  properties
    PSL_Properties::Contract => "
    -- Constants
      const IM_MODE_OFF : int = 0;
      const IM_MODE_IDLE : int = 1;
      const IM_MODE_PAUSED : int = 2;
    ...

    -- macros
      property in_off = INFUSION_OUT.Current_System_Mode = IM_MODE_OFF;
      property in_idle = INFUSION_OUT.Current_System_Mode = IM_MODE_IDLE;
      property in_paused = INFUSION_OUT.Current_System_Mode = IM_MODE_PAUSED;
    ...

    -------------------------------------------------------------------
    -- SYSTEM REQUIREMENTS
    -------------------------------------------------------------------
    ...
    -- REQ 4 :
      property RE_system_on_implies_idle =
        ((not prev(TLM_OUT.System_On, true) and (TLM_OUT.System_On)) => in_idle);
      assert RE_system_on_implies_idle;
    ...
    -- REQ 6 :
      property mode_off_implies_infusion_rate_zero =
          in_off => (INFUSION_OUT.Commanded_Flow_Rate = 0);
      assert mode_off_implies_infusion_rate_zero;
    ...
    -- REQ 59:
      property empty_reservoir_implies_no_flow =
      ((((prev(INFUSION_OUT.Reservoir_Volume, 0)) < CONFIG_IN.Empty_Reservoir_Val)
         and prev(INFUSION_OUT.Infusing, false)) =>
         (INFUSION_OUT.Commanded_Flow_Rate = 0));
      assert empty_reservoir_implies_no_flow;
    ...
    -------------------------------------------------------------------------
    -- SYSTEM LEVEL ASSUMPTIONS
    -------------------------------------------------------------------------
      assume DB_IN.flow_rate_kvo > 0 ;
      assume CONFIG_IN.audio_level_val > 0 ;
      assume CONFIG_IN.empty_reservoir_val > 0 ;
  ----------------------------------------------------------------
  -- End of Infusion Manager Contract
  ----------------------------------------------------------------
  ";
end GPCA_SW;
```

**Figure 5: Portions of GPCA AADL/Agree Model**

tics in Stateflow, the distinction between transition actions and condition actions, etc.), they are by far the most widely used notations in industry and suit our modeling needs well.

Furthermore, since our goal is to demonstrate the power of formal reasoning, it is essential to have verification support for the behavioral modeling notation. The MathWorks now supports a plug-in formal verification tool, the Simulink Design Verifier [23], for verification of Simulink and Stateflow behavioral models.

Finally, since one of the goals with the effort described in this paper was to illustrate how to perform architectural modeling, behavioral modeling, and compositional verification in practice, our aim was to work with commercially supported and/or mature open source tools so that adopting our work in industrial settings would have a low threshold. Thus, Simulink, Stateflow, and AGREE were natural choices.

## 4.1 Alarms Component

The Alarms component monitors the system for any exceptional conditions, prioritizes conditions if there are multiple simultaneous exceptional conditions, and raises visual/audio notifications depending on the situation. Figure 6 shows the top level model of the Alarms component. The dashed ”roundangles” indicate state-machines that exist in parallel but that will execute in the sequence indicated by the sequence numbers (in this case, CheckAlarms and then Notification). The CheckAlarms state-machine determines which alarms to raise based on the the current information of the system. Figure 7 shows the behavioral model of the reservoir empty alarms feature, one of the sub-states of CheckAlarms state-machine. The behavior is relatively simple: if infusion is in progress (information gathered from the Infusion Manager component discussed below), and the estimated volume remaining in the reservoir (also from the Infusion Manager) is less than the Empty Reservoir threshold, the reservoir is considered empty and the alarm will be raised. In the Alarms module there are currently 18 different alarms that can be raised. The alarms range from low criticality alarms that amount to an on screen notification to the clinician and no disruption of the therapy, to highly critical alarms (such as Empty Reservoir or Air In Line) that necessitate a visual and aural notification (determined by the Notification state-machine) as well as suspension of therapy.

## 4.2 Infusion Manager Component

The Infusion Manager is responsible for maintaining the state of the infusion of drug into the patient and, based on the therapy selected, determining the appropriate flow-rate for the infusion. In addition, the Infusion Manager is responsible for estimating the total volume infused and the remaining reservoir volume. The top level of the Infusion Manager can be seen in Figure 8.

Initially, the Infusion Manager is idle and no infusion is taking place. After the pump has been configured, that is, the various infusion parameters have been set, the clinician can commence the delivery of an infusion therapy as shown in Figure 8; Figure 9 shows the details of the therapy behavior. When delivering therapy, the pump may be actively delivering drug (ACTIVE) or drug delivery may be suspended (PAUSED). System may be PAUSED for two reasons; (1) a critical alarm may have been raised necessitating the cessation of drug delivery (information from the Alarms subsystem), or (2) the clinician has requested a pause to perform some action.

When the pump is delivering therapy (in the ACTIVE state), this particular version of the GPCA can provide three types of therapy: (1) a basal infusion rate that is the basis for the therapy, (2) a higher patient requested bolus rate that is limited in duration and frequency (to prevent patient self-inflicted overdoses), and (3) a clinician programmed intermittent bolus regime where the patient gets an extra dose of drug at regularly scheduled intervals (see Figure 10). Depending on various conditions, the infusion pump will switch between these delivery therapies. As an example, consider a scenario where the pump is delivering therapy at the basal flow rate. The patient requests a bolus dose and—if all preconditions are met—this therapy is turned on. In this case, the patient bolus takes precedence over all other therapies and the flow rate delivered to the patient is determined by the patient bolus rate. When the patient bolus has been delivered, the previous therapy is resumed. In this model, the prioritization of the various therapies is handled by the ARBITER state machine. The inclusion of an arbiter as well as the modeling of the various therapies as parallel state machines was done for clarity, modularity, and product family reasons; this structure makes it relatively easy to add additional therapies without causing ripple-effects throughout the model. The details and rationale related to the structuring of the behavioral models is outside the scope of this paper and has been discussed elsewhere [30].

## 4.3 Specification of Behavioral Requirements

The required properties of the architectural components is captured in AGREE using the past-time subset of past-time linear temporal logic—PLTL (Section 3.2). Unfortunately, the verification tools available for Simulink and Stateflow do not currently support this formalism for property specification. Naturally, it would be highly desirable if the property specification language was consistent throughout the modeling effort; this is a tool integration and engineering problem we have not yet addressed. Instead, we recapture the required component properties for verification in the Simulink Design Verifier. The Simulink Design Verifier requires all properties to be specified as Boolean expressions in one of the available MATLAB notations. For example, the empty reservoir property for the Alarms component depicted in Figure 11 is captured as a Simulink verification block. The input signals on the left side are the same inputs that are provided to the Alarms component capturing the component behavior (as discussed in the previous section). The gray circle containing a P (for property) indicates that the verification tools will attempt to verify that this Simulink block always generates a signal that is True; if the signal is ever False, the tools have revealed a property violation and will report a counterexample. The logic of the verification condition is in Figure 11 expressed using embedded MATLAB, a subset of the MATLAB computing language that supports efficient code generation for deployment in embedded systems.

Alternatively, the verification conditions could be captured using the Simulink or Stateflow notations. The same condition captured as a Simulink model can be seen in Figure 12.

In our work we have preferred using embedded MATLAB code. Capturing properties in this notation has several advantages. First, the property can be structured to closely resemble the natural language “shall” requirements in the requirement document as well as the PLTL properties used in AGREE. This closeness in structure reduces the opportunities for transcription mistakes and makes the properties easier to inspect. In the future, the translation between the AGREE properties and the Design Verifier properties will be automated. Second, the textual notations makes it easy to comment out properties as the verification process is underway (verifying all properties all the time may be a waste of time when there is one problematic property of interest). Finally, we have developed preference for the textual notation since we find it quicker and easier to create and maintain the required properties textually.

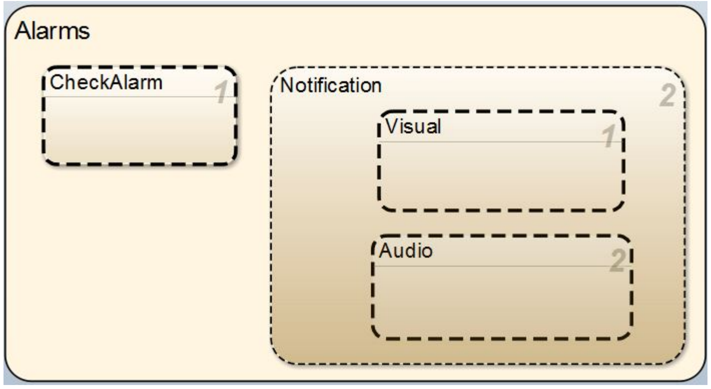


Figure 6: Alarm Behavioral Model Top-Level State Machine

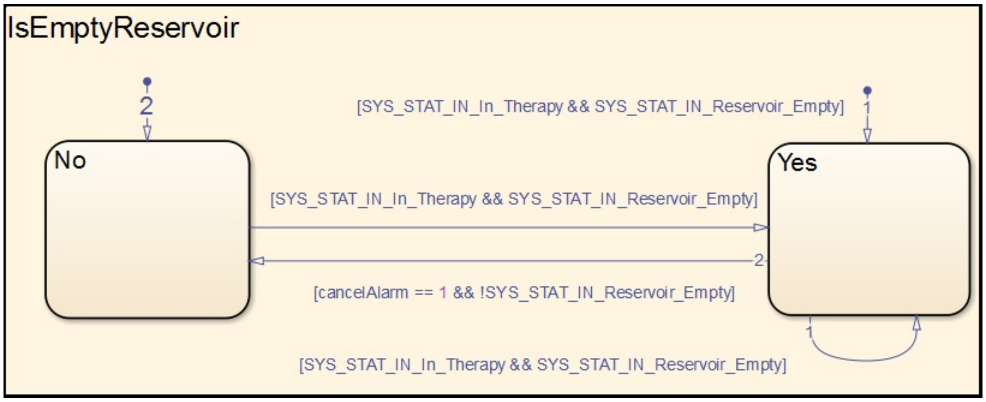

IsEmptyReservoir
2
[SYS_STAT_IN_In_Therapy && SYS_STAT_IN_Reservoir_Empty]
1
No
Yes
[SYS_STAT_IN_In_Therapy && SYS_STAT_IN_Reservoir_Empty]
2
[cancelAlarm == 1 && !SYS_STAT_IN_Reservoir_Empty]
1
[SYS_STAT_IN_In_Therapy && SYS_STAT_IN_Reservoir_Empty]

Figure 7: Behavioral model of the reservoir empty alarms feature.

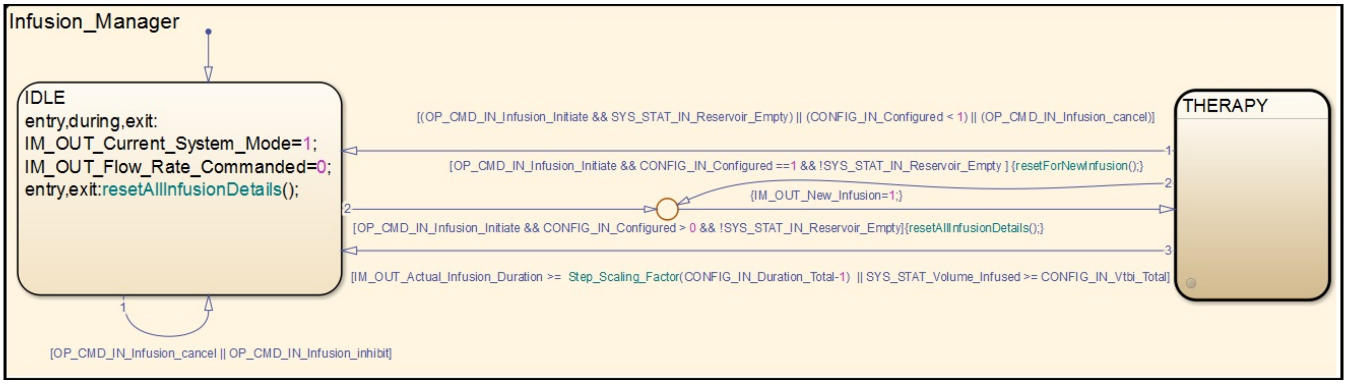

Infusion_Manager
IDLE
entry,during,exit:
IM_OUT_Current_System_Mode=1;
IM_OUT_Flow_Rate_Commanded=0;
entry,exit:resetAllInfusionDetails();
THERAPY
[(OP_CMD_IN_Infusion_Initiate && SYS_STAT_IN_Reservoir_Empty) || (CONFIG_IN_Configured < 1) || (OP_CMD_IN_Infusion_cancel)]
1
[OP_CMD_IN_Infusion_Initiate && CONFIG_IN_Configured ==1 && !SYS_STAT_IN_Reservoir_Empty ] {resetForNewInfusion();}
2
{IM_OUT_New_Infusion=1;}
2
[OP_CMD_IN_Infusion_Initiate && CONFIG_IN_Configured > 0 && !SYS_STAT_IN_Reservoir_Empty]{resetAllInfusionDetails();}
3
[IM_OUT_Actual_Infusion_Duration >= Step_Scaling_Factor(CONFIG_IN_Duration_Total-1) || SYS_STAT_Volume_Infused >= CONFIG_IN_Vtbi_Total]
1
[OP_CMD_IN_Infusion_cancel || OP_CMD_IN_Infusion_inhibit]

Figure 8: Infusion Manager Behavioral Model Top-Level State Machine

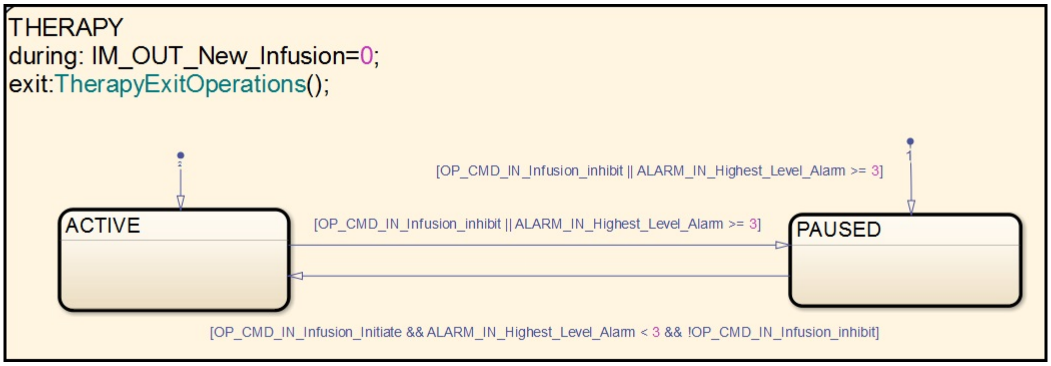

THERAPY
during: IM_OUT_New_Infusion=0;
exit:TherapyExitOperations();
2
1
[OP_CMD_IN_Infusion_inhibit || ALARM_IN_Highest_Level_Alarm >= 3]
ACTIVE
[OP_CMD_IN_Infusion_inhibit || ALARM_IN_Highest_Level_Alarm >= 3]
PAUSED
[OP_CMD_IN_Infusion_Initiate && ALARM_IN_Highest_Level_Alarm < 3 && !OP_CMD_IN_Infusion_inhibit]

Figure 9: Therapy state machine.

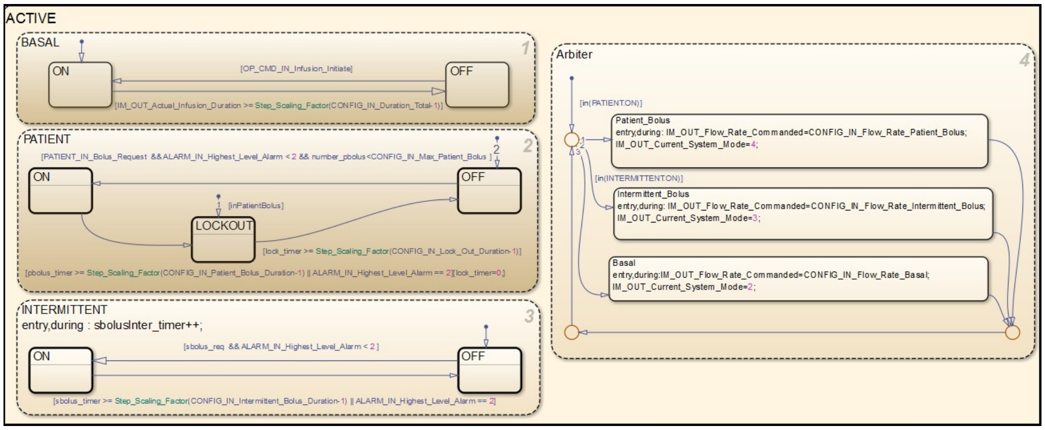
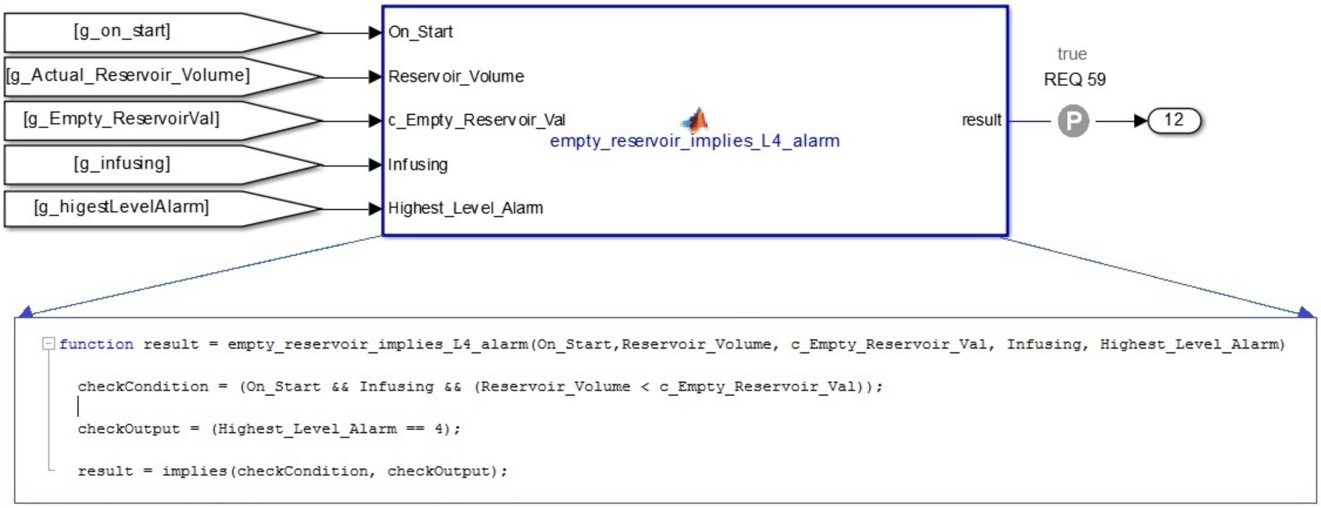

ACTIVE
BASAL
1
ON
[OP_CMD_IN_Infusion_Initiate]
OFF
[IM_OUT_Actual_Infusion_Duration >= Step_Scaling_Factor(CONFIG_IN_Duration_Total-1)]
PATIENT
2
[PATIENT_IN_Bolus_Request && ALARM_IN_Highest_Level_Alarm < 2 && number_pbolus<CONFIG_IN_Max_Patient_Bolus ]
ON
OFF
1
[inPatientBolus]
LOCKOUT
[lock_timer >= Step_Scaling_Factor(CONFIG_IN_Lock_Out_Duration-1)]
[pbolus_timer >= Step_Scaling_Factor(CONFIG_IN_Patient_Bolus_Duration-1) || ALARM_IN_Highest_Level_Alarm == 2]{lock_timer=0;}
INTERMITTENT
entry,during : sbolusInter_timer++;
3
[sbolus_req && ALARM_IN_Highest_Level_Alarm < 2 ]
ON
OFF
[sbolus_timer >= Step_Scaling_Factor(CONFIG_IN_Intermittent_Bolus_Duration-1) || ALARM_IN_Highest_Level_Alarm == 2]
Arbiter
4
[in(PATIENT.ON)]
1
2
3
Patient_Bolus
entry,during: IM_OUT_Flow_Rate_Commanded=CONFIG_IN_Flow_Rate_Patient_Bolus;
IM_OUT_Current_System_Mode=4;
[in(INTERMITTENT.ON)]
Intermittent_Bolus
entry,during: IM_OUT_Flow_Rate_Commanded=CONFIG_IN_Flow_Rate_Intermittent_Bolus;
IM_OUT_Current_System_Mode=3;
Basal
entry,during:IM_OUT_Flow_Rate_Commanded=CONFIG_IN_Flow_Rate_Basal;
IM_OUT_Current_System_Mode=2;

Figure 10: The three main therapies and the arbiter determining which one is in control of the pump.



```
function result = empty_reservoir_implies_L4_alarm(On_Start,Reservoir_Volume, c_Empty_Reservoir_Val, Infusing, Highest_Level_Alarm)

  checkCondition = (On_Start && Infusing && (Reservoir_Volume < c_Empty_Reservoir_Val));

  checkOutput = (Highest_Level_Alarm == 4);

  result = implies(checkCondition, checkOutput);
```

Figure 11: Alarm component property for Empty Reservoir check expressed in embedded MATLAB.

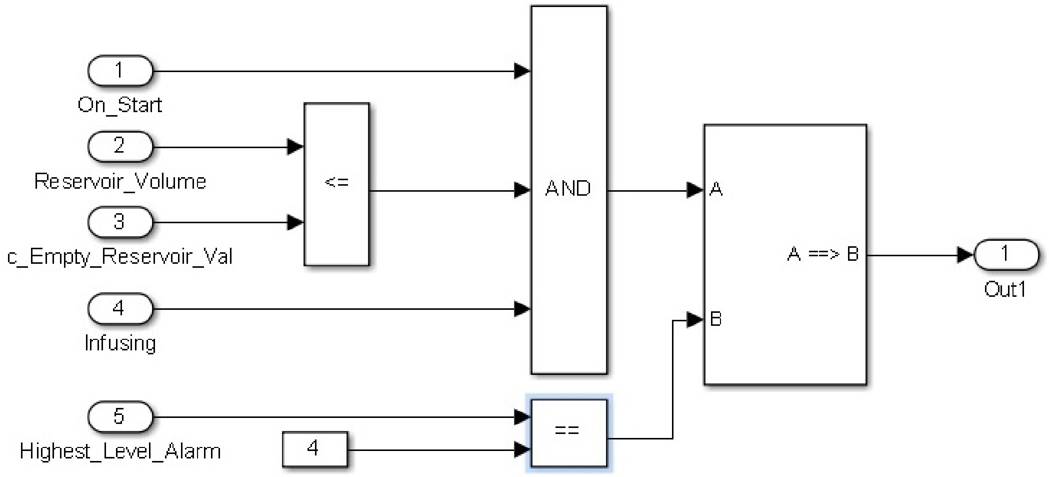


**Figure 12: Alarm component property for Empty Reservoir check expressed in Simulink.**

## 5. VERIFYING THE GPCA SYSTEM

To verify the GPCA system behavior, it is necessary to prove that the components satisfy their component-level requirements and that the component-level requirements are adequate to establish the system-level requirements for each level within the architectural hierarchy. These activities were performed in parallel, with one team member leading the component-level verification effort and another leading the architecture-level verification.

### 5.1 Verification Approach and Tools

For the verification effort of the GPCA model, we used two different model checking tools. For the Simulink/Stateflow verification, we used the Simulink Design Verifier (SLDV) [23], and for the architectural verification, we used the recently developed Rockwell Collins JKind tool[1]. Both of the tools use k-induction [34] algorithms implemented on top of a Satisfiability Modulo Theories (SMT) solver [8] to reason about infinite-state models involving real (rational) numbers and bounded or unbounded integers.

### 5.2 Verification Results

The verification results for our current analysis are shown in Table 1. Currently, we are analyzing component-level properties for the Alarms and Infusion Manager components (we expect to add properties related to logging and the top-level system modes in the near future). For composing analysis results, we are using AADL and the AGREE OSATE plug-in. All data gathering was performed on a Dell Latitude E6430 running Windows 7 64-bit edition with an Intel Core i7-3720QM CPU running at 2.60 GHz and 6 GB of RAM. For each slot in the table, we ran the verification three times and recorded the mean time (though variance was quite low between runs).

To evaluate the scalability of the architectural analysis, we also created a model in Simulink that contains all of the behavioral models in the AADL architectural model; we call the composite Simulink model the *monolithic model*. We can then check the scalability of the compositional vs. monolithic analysis by analyzing the same system twice:

**Compositional Approach:** In this approach, the component-level properties for the Alarms and Infusion Manager subsystems are proven using Simulink Design Verifier; these properties are then used to prove the system property.

**Monolithic Approach:** In this approach, the monolithic model containing the Alarms and Infusion Manager subsystems as well as the 19 system-level properties is analyzed using Simulink Design Verifier.

For the monolithic model, Simulink Design Verifier is unable to prove the current set of system-level properties within 60 minutes. For the compositional approach, the total time required for analysis is slightly under 5 minutes (46.5 + 224.5 + 2 = 273 seconds). Notably, the time required for the compositional portion of the analysis was only 2 seconds. As we add additional subsystems and additional functionality and proof obligations to the component-level models, we expect the difference in analysis time to become more pronounced.

## 6. DISCUSSION

In the process of conducting this case study, we gained some methodological insights towards model structuring and verification.

### 6.1 Model Structure

It is crucial to structure the component models carefully taking into consideration their various uses and evolution. Initially, we placed both the functional model and requirements models (properties) into the same Simulink model. However, we wanted the Simulink models to serve two important purposes—code-generation and verification. Also, we wanted the implementation of the component and the specification of the requirements for that component to be able to evolve in parallel, given that the input/output interface for the component is stable. To support these desires, we create three models for each component. For component `foo`, we create the model `foo_functional.mdl` to capture the detailed component behavior, `foo_properties.mdl`

---

[1] Available at: https://github.com/agacek/jkind

|  | Alarms Sub-System | Infusion Manager Sub-System | Monolithic Model | AADL Model |
|---|---|---|---|---|
| Number of Inputs | 45 | 21 | 55 | 55 |
| Number of Outputs | 10 | 11 | 15 | 15 |
| Number of Properties | 11 | 15 | 19 | 19 |
| Design Verifier Execution Time | 46.5s | 224.5s | >3600s (12/19 proved) | NA |
| AGREE Execution Time | NA | NA | NA | 2s |

**Table 1: Verification Results**

to capture the requirements for the component, and `foo_verification.mdl` that connects the functional and requirements models. Eventually, given an AADL model containing the I/O structure and requirements allocated to the component, we plan to auto-generate the verification models. We intend to use the functional models to generate code using the MATLAB code-generation tools; code that can then be executed and tested on the target platform.

## 6.2 Debugging

The formalization effort found errors in requirements, architectural decomposition, and behavioral models. In terms of requirements, we found implicit assumptions, ambiguous textual requirements, and specifications that were not physically achievable (note that this is not a new insight [12, 28]). For example, we have various requirements related to under- and over-infusion, where at any instant an over- or under-infusion amount greater than a certain threshold vs. the current desired flow rate should trigger a critical alarm and stop infusion:

> *SW3.4.5.21.2 The application shall validate during all infusion modes if the received flow rate is within programmed flow rate with flow rate precision of p % of the programmed flow rate of the respective infusion mode. The received flow rate shall never be more than MAX ml/hr above the prescribed flow rate + precision.*

These requirements as written, however, are not achievable at start of infusion or during mode transitions (such as patient bolus), instants where the threshold changes via a step function and the pump requires a small amount of time to reach the set point. In the process of formalizing the requirement, we identified the problem.

The behavioral models contained the largest share of errors found by analysis, usually involving incorrectly specified boundary conditions. For example, incorrect default transitions (transitions that occur when a state machine is first entered) led to several instances where alarms were not correctly raised at the moment when the fault condition occurred.

In terms of architectural models, a significant error was found related to the structure of the decomposition that we had chosen for the system. A property closely related to the empty reservoir property discussed earlier is:

> *No infusion shall occur while the drug reservoir is below the empty-drug-threshold for the drug in the reservoir.*

This property is violated by the current model. The issue is that while the Infusion Manager will stop infusion if a critical alarm occurs, the Alarms subsystem will raise an alarm for empty reservoir only if the system is infusing. Therefore, the system must start infusion before the alarm will occur and cause infusion to stop. To fix this issue, we will split the Alarms system into a system status and alarms component. The status component will have an “okToInfuse” output to handle this case.

Both SLDV and the AGREE tool-set produce counterexamples when properties are violated. For behavioral models, tracing counterexamples is straightforward; they are test cases that can be run through the simulator for Simulink. On the other hand, if a counterexample occurs in AGREE, it is akin to a proof failure; the counterexample is a trace instance in which the subsystem properties and system assumptions were not strong enough to establish the system level property. In some cases, these counterexamples can be very difficult to debug—they may not correspond to actual executions of the system (because the component level properties are approximations of the actual system behavior) and therefore cannot be simulated. It will be important to provide engineers better tools with which to understand the relevance of different variables towards these counterexamples, to try to assign blame to subcomponents that likely contributed to the proof failure, and to perform some amount of automated strengthening by “opening the architectural boxes” and looking at the behavior of subcomponents. We are just beginning to investigate better tool support and engineering aspects of compositional reasoning as part of a NASA program that started in June 2013.

As mentioned above, the majority of faults have been found in the behavioral models, but the most difficult to fix involved the architectural decomposition. The benefit of having the ability to perform architectural verification is that these problems can be identified early in the verification process rather than in integration and system test phases.

## 6.3 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. This model must be extended in order to deal with distributed computations where components do not share a common clock signal, or where messages between components are not guaranteed to be delivered and/or communication times may vary. Second, AGREE can verify only *invariants*, so liveness properties, such as *the system will eventually deliver a message*, cannot be specified in AGREE. In our experience, this is not as severe a limitation as it may seem. Most systems are concerned with *bounded liveness* in which an action must occur within a time interval; these properties 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. Bit-precise floating point reasoning is an area of active research in the decision procedures community. As floating point decision procedures become integrated into SMT solvers, we will add support for accurate analysis of floating point numbers. 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.

Finally, we are proving properties of *models*. The models that we are analyzing are used for code generation, and so describe a complete implementation of the functional behavior of the GPCA. However, the verification process that we describe will not catch errors in the MATLAB code generator or the C tool chain used to compile the generated code. In addition, external code such as "glue" code for communication with the actual sensors and actuators and (possibly) an RTOS are outside the scope of the model and will not be analyzed.

## 7. RELATED WORK

Compositional verification has attracted significant research attention because of its viability as a scalable technique for reasoning about complex systems. These techniques typically employ some form of an assume-guarantee approach: each component (or module) is shown to individually guarantee some desirable properties under certain assumptions about its environment, which includes other interacting components. These separate arguments are then combined in a logically coherent fashion to construct an argument that the full system satisfies its required properties under the given assumptions about its environment. When a formal technique such as model-checking is used to verify the properties, these "arguments" in essence are proofs of properties for the system models being verified.

Early efforts in such modular reasoning dealt with parallel programs that interact via message passing and shared variables [29, 18, 32] and recently in the context of *behavioral programming* frameworks [15]. In our present work, the focus is on verifying architectural models with behavioral components that interact through data flow and execute in some fixed sequential order at every computation step. The components themselves are modeled as extended finite state machines which are amenable to model-checking techniques. Event-based composition and verification of finite-state transition systems has been explored in the contexts of I/O automata [21], interface automata [6], Computational Tree Logics [3, 9], and in a process algebraic framework [2]. Hagen et al. discuss verification of safety properties of dataflow programs in Lustre using model-checking techniques that use an underlying SMT-solver [10]. In our present work we use the same SMT-based algorithm to verify properties in the AGREE framework.

McMillan describes a compositional approach to hardware verification where the component-level properties are verified using model-checkers, while the higher level properties are reasoned about using human assisted proof-techniques in a semi-automated fashion [26]. In the present work, we use a similar strategy for proving system properties by leveraging the human effort involved in system design for architectural decomposition as well as requirements allocation to sub-components. Such a division of labor between man and machine for full-system verification seems justified given that, in general, finding a good decomposition that is beneficial for verification is hard [4].

Simulink Design Verifier has been used to analyze cyber-physical system designs in the transportation domain [7]. Here, various properties of a train-tracking system are verified using the k-induction and bounded model-checking capabilities of the tools; the state-space explosion problem is addressed by adopting various ad-hoc tactics for model optimization, property decomposition and induction. In our work, we address the state-space explosion problem by using the architectural decomposition to partition the verification task for safety properties between AGREE tool and Simulink Design Verifier.

The SPEEDS approach to embedded system development [35] envisions rapid innovations through a model-based engineering paradigm supported by formal analyses. At its core, it requires that the architectural description of the system is annotated with assume-guarantee style contracts on components that cover both functional and non-functional aspects. Our present work can be seen as a key enabler for realizing the benefits of such an approach in the development of cyber-physical systems.

The BLESS annex and tool [20] supports reasoning over component-level behaviors defined using state machines and a simple imperative language. It supports component invariants and a variety of well-formedness properties for component behavior models. Unlike AGREE, which is fully automated, BLESS uses a user-driven deductive approach to proof. This allows specification of and reasoning about a larger class of properties, such as those involving non-linear arithmetic, than is currently supported with AGREE. On the other hand, BLESS specifications are defined on individual subcomponents and threads; there does not appear to be a notion of n-level composition of proof results across threads and systems.

## 8. CONCLUSION

We have presented a scalable and practical approach to compositionally verify a realistic medical cyber-physical system using commonly used modeling notations and readily available tools. Our assume-guarantee framework allows one to formally establish invariants of the full system by leveraging its hierarchical architectural decomposition for verification. This also allows the effective utilization of formal verification tools at the component-level, which may not be otherwise practical for verifying system-level properties. The partitioning of verification tasks along architectural lines has other benefits for system development: it provides early feedback on key architectural decisions, promotes iterative exploration of the solution space and fosters a synergistic evolution of requirements specification and system design. While there are some limitations in the kinds of models and properties that can be handled given our present choices of specification formalisms and verification techniques, we believe our approach represents a good trade-off point between the possible and the practical.

## 9. ACKNOWLEDGEMENTS

We would like to thank Andrew Gacek and Darren Cofer at Rockwell Collins for their insight and assistance in structuring and debugging the architectural models, and for developing JKind and AGREE.

## 10. REFERENCES

[1] Generic infusion pump project, http://rtg.cis.upenn.edu/gip.php3.

[2] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *Software, IEEE*, 28(3):41–48, 2011.

[3] E. Clarke, D. Long, and K. L. McMillan. Compositional model checking. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 353–362, 1989.

[4] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 97–108, New York, NY, USA, 2006. ACM.

[5] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In A. E. Goodloe and S. Person, editors, *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140, Berlin, Heidelberg, April 2012. Springer-Verlag.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, Sept. 2001.

[7] J.-F. Etienne, S. Fechter, and E. Juppeaux. Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain. In M. Aiguier, F. Bretaudeau, and D. Krob, editors, *Complex Systems Design & Management*, pages 61–72. Springer Berlin Heidelberg, 2010.

[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[9] O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[10] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*, pages 1–9, 2008.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[12] A. Hall. Seven myths of formal methods. *IEEE Software*, September 1990.

[13] J. Hammond, R. Rawlings, and A. Hall. Will it work? [requirements engineering]. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 102 –109, 2001.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[15] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-checking behavioral programs. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 279–288, New York, NY, USA, 2011. ACM.

[16] IEEE. *IEEE Std. 1850-2005. Property Specification Language (PSL)*. IEEE, 2005.

[17] M. Jackson and P. Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE'95)*, pages 15–24, May 1995.

[18] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983.

[19] J. A. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, UCLA, 1968.

[20] B. Larson, P. Chalin, and J. Hatcliff. BLESS: Formal specification and verification of behaviors for embedded systems with software. In *Proceedings of the 5th NASA Formal Methods Symposium.* Springer-Verlag, 2013.

[21] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[22] MathWorks. The MathWorks Inc. corporate web page. Via the world-wide-web: http://www.mathworks.com, 2004.

[23] Mathworks Inc. Simulink Design Verifier product web site. http://www.mathworks.com/products/sldesignverifier/.

[24] Mathworks Inc. Simulink product web site. http://www.mathworks.com/products/simulink.

[25] Mathworks Inc. Stateflow product web site. http://www.mathworks.com.

[26] K. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1Ű3):279 – 309, 2000.

[27] K. L. McMillan. Circular compositional reasoning about liveness. Technical Report 1999-02, Cadence Berkeley Labs, Berkeley, CA 94704, 1999.

[28] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.

[29] J. Misra and K. Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, SE-7(4):417–426, 1981.

[30] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Modes, features, and state-based modeling for clarity and flexibility. In *Fifth International Workshop on Modeling in Software Engineering*, May 2013.

[31] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34:115–117, 2001.

[32] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer Berlin Heidelberg, 1985.

[33] SAE-AS5506. *Architecture Analysis and Design Language*. SAE, Nov 2004.

[34] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.

[35] SPEculative and Exporatory Design in System engineering. http://www.speeds.eu.com/, 2006-2009.

[36] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam. Your what is my how: Iteration and hierarchy in system design. *Software, IEEE*, 30(2):54–60, 2013.