Trusted Composition of Internet of Medical Things Over Imperfect Networks

Ehsan Ahmad *, Brian Larson, Abdulbasid S. Banga

Posted Date: 14 May 2024

doi: 10.20944/preprints202405.0908.v1

Keywords: AADL; BLESS; IoT; IoMT; Medical Device Interoperability; Composition of Medical Devices; Healthcare

Article

Trusted Composition of Internet of Medical Things Over Imperfect Networks

Ehsan Ahmad [1,*], Brian Larson [2] and Abdulbasid Banga[1]

[1] College of Computing and Informatics, Saudi Electronic University, Saudi Arabia;
[2] Multitude Corporation, Minnesota, USA;
brl@multitude.net (B.L.); a.banga@seu.edu.sa (A.B.)
* Correspondence: e.ahmad@seu.edu.sa

Abstract: The Internet of Medical Things (IoMT) represents a specialized domain within the Internet of Things, focusing on medical devices that require regulatory approval to ensure patient safety. Trusted composition of IoMT systems aims to ensure high assurance of the entire composed system, despite potential variability in the assurance levels of individual components. Achieving this trustworthiness in IoMT systems, especially when using less-assured, commercial, off-the-shelf networks like Ethernet and WiFi, presents a significant challenge. To address this challenge, this paper advocates a systematic approach that leverages the Architecture Analysis & Design Language (AADL) along with Behavior Language for Embedded Systems with Software (BLESS) specification and behavior. This approach aims to provide high assurance on critical components through formal verification, while using less-assured components in a manner that maintains overall system determinism and reliability. A clinical case study involving an automated opioid infusion monitoring IoMT system is presented to illustrate the application of the proposed approach. Through this case study, the effectiveness of the systemic approach in achieving trusted composition of heterogeneous medical devices over less-assured networks is demonstrated.

Keywords: AADL; BLESS; IoT; IoMT; Medical Device Interoperability; Composition of Medical Devices; Healthcare

1. Introduction

An Internet of Things (IoT) is a collection of inter-connected programmable *things* with the ability of data collection and exchange [1]. Depending on the application domain, the things may include any thing ranging from temperature sensors, cameras, smart phones, home appliances, cars, to medical devices, and so on. IoT systems are used to provide integrated services of monitoring and controlling which are beyond the capabilities of individual devices. Such integration of heterogeneous things enables access from anywhere, anytime, and by anything through networks that span from Body Area Networks (BAN) to networks stretching across the globe.

The term "Internet of Medical Things" (IoMT) refers to a composition of medical devices, applications, and healthcare IT systems that are connected through computer networks. Medical devices equipped with interoperable interfaces along with the communication network form the basis of IoMT. Under the purview of IoMT, medical device composition extends data sharing for electronic health records[2,3], with therapeutic medical devices[4,5]. Therefore, application of IoMT to control patient therapy has regulatory requirements where simply health monitoring does not.

Our reliance on IoMT systems is predicated on both the correct functionality of individual devices and their communication with one another, which depends on the network(s) that connects them. Different wired, and wireless technologies and protocols are used to establish the network that connects the medical devices, depending on the type of working environment, such as an emergency department or telesurgery (a remote surgery performed by a surgeon). Common networking technologies include cellular networks (LTE, 3G, 4G, 5G, etc.), local and personal area networks (WiFi, Bluetooth), low power wide area networks (LoRaWAN), and mesh networks (Zigbee, RFID). As non-regulated, commercial-grade networks are mostly used for availability and cost effectiveness, composition of IoMT must accommodate unreliable communication, yet still be at least safe.

 doi:10.20944/preprints202405.0908.v1

A network used for IoMT may become unreliable as a result of connecting heterogeneous devices from various manufacturers interacting in heterogeneous environments [6]. Vulnerabilities related to security, privacy, and confidentiality present in the individual devices also affect the integrated network. Several studies like [7], [8], and [9] have explored different trust issues in IoMT systems.

Ensuring reliable communication among medical devices is a challenge. Existing literature on composition of medical devices for IoMT system design, mostly focus on data and device interoperability assuming a secure and trusted network. Studies like [2,4,10] typically presume a trustworthy and reliable communication network when considering safe interoperability of medical devices for application platforms and integrated clinical settings.

Every deployed network is imperfect. Messages may be corrupted, delayed, or deleted. Here we are concerned about using formally verified software, together with software with lesser verification, such as commercial-off-the-shelf (COTS) software radio or TCP/IP software stacks. Medical device designers believe the compiled binaries purchased from others are correct, having any (known) security vulnerabilities patched, but we cannot be *sure* even if we compiled the binaries ourselves from sources because they have not been formally verified–merely tested and used.

The trusted composition of IoMT aims to achieve high assurance of the system as a whole even though certain components of the system may have lower assurance. Therefore, if we can't verify that every execution is correct (as we can with rigorous formal verification) we can at least verify that each particular use of COTS software and hardware is correct using loop-back. Every message is looped back to the sender to be compared with the original. If it doesn't match, or doesn't arrive in time, the sender can take other action to assure safety. In this case, an audible alarm to alert a healthcare provider that high assurance of correct interaction of interconnected medical devices has been lost.

1.1. Contributions:

The major contributions of this study are as follows.

- We present a systematic approach for achieving high assurance in IoMT systems by integrating high-assurance software with lower-assurance counterparts, thereby ensuring overall system reliability and safety.
- We demonstrate the application of Architecture Analysis & Design Language (AADL) in facilitating the trusted composition of medical devices within IoMT systems over imperfect networks. Specifically, we address uncertainties related to the timely and reliable exchange of information among devices on a communication-to-communication basis.
- We illustrate the use of the Behavior Language for Systems with Software (BLESS) for behavior specification, implementation, and verification of an IoMT system consisting of pulse oximeter, respiration monitor, patient-controlled analgesic (PCA) pump, and a control application, communicating over a network.

1.2. Outline

The relevant background is provided in the following section. Section 3 presents clinical use case of an IoMT for system of monitoring of the patient-controlled opioid infusion. Section 4 describes the structural modeling of the clinical IoMT while Section 5 details the behavior specification with BLESS properties. Behavior implementation using BLESS annex subclause is presented in Section 6. Section 7 discusses behavior verification conditions and Section 8 covers the behavior correctness proof. Section 9 concludes the study.

2. Background

The Architecture Analysis & Design Language (AADL) is SAE International standard AS5506D, and was created for systems engineering of cyber-physical systems. AADL has well-defined semantics for components and their composition. In particular, AADL component types describe everything visible from outside the component (features, ports, parameters, and accesses), while their component

 doi:10.20944/preprints202405.0908.v1

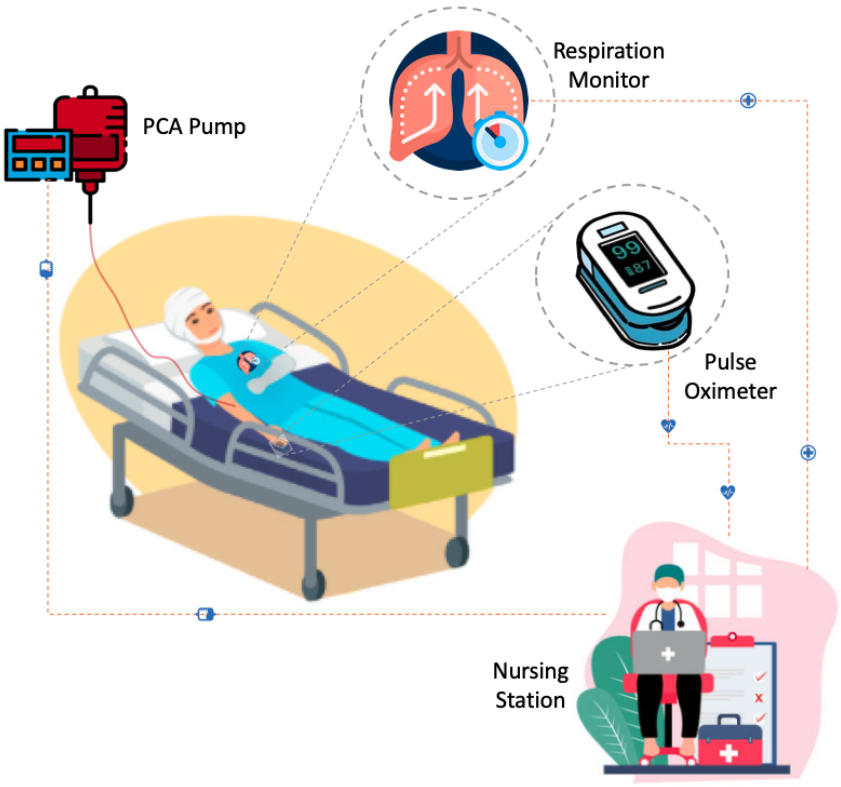

Figure 1. A clinical use case of opioid infusion monitoring

implementations describe their internals. Each component type may have several component implementations. Component implementations may be composite components containing subcomponents and connections between them, or may be hardware and/or software not further partitioned.

The Open Source AADL Tool Environment (OSATE) [11] is an IDE with analysis tools built on Eclipse to implement AADL for the modeling and analysis of real-time embedded systems. The Behavior Language for Embedded Systems with Software (BLESS) is an annex sublanguage of AADL which augments the core architecture language to define specification and behavior implementations [12]. A BLESS plugin to OSATE allows program development and transforms programs with proof outlines into deductive proofs.

AADL facilitates system engineering to minimize problems discovered during systems integration–when they are most costly to remedy. Architecture-Centric Virtual Integration (ACVI) is a design methodology for using AADL to preclude system integration issues[13]. ACVI was adapted for use to compose interoperable medical devices as medical device virtual integration (MDVI) in [5]. With MDVI, systems engineers use AADL's property mechanism to attach assertions (predicates or expressions) to component features expressing what is guaranteed to be true (for outputs) or assumed to be true (for inputs). Connections between features give rise to assume-guarantee contracts which can be formally verified before the components are designed. Additionally, components may have invariants–much like loop invariants. Invariants for composite components are implied by invariants of their subcomponents.

3. Clinical Example System

To demonstrate how the trusted composition of IoMT may be achieved across an imperfect network, a clinical use case is presented in this section and will be used throughout the paper. The use case involves automated monitoring of the patient-controlled opioid infusion after a surgical procedure. As depicted in Figure 1, the patient is connected with a *respiration monitor* to measure breathing rate, and *pulse oximeter* to measure blood oxygen (SpO2) and heart rate. In order to administer opioids for

 doi:10.20944/preprints202405.0908.v1

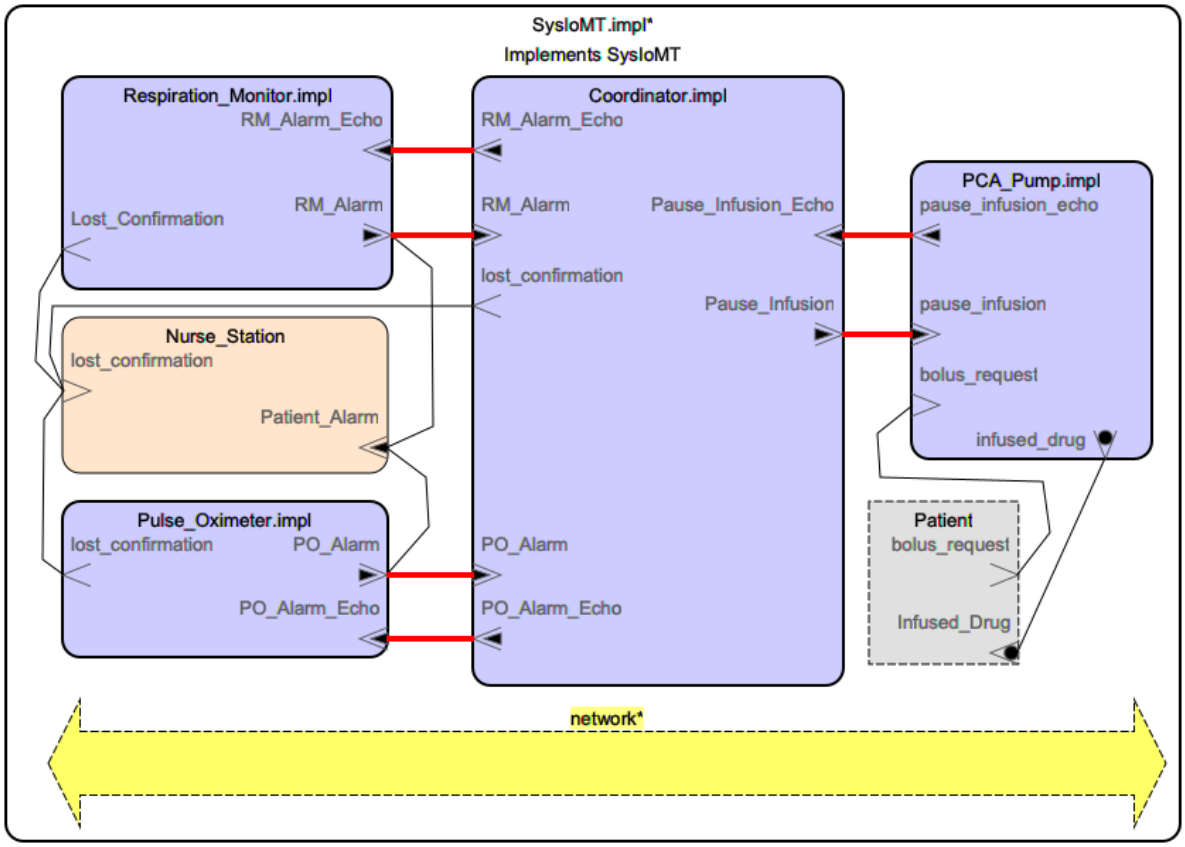

Figure 2. Top-level AADL model

pain management, the patient is attached to a *patient-controlled analgesic* (PCA) infusion pump through an infusion set usually into a vein on the back of a hand. With a PCA pump, a patient receives opioid infusion at a basal rate, but can also request an additional bolus by pressing a button connected to the PCA pump. The typical PCA pump operates as a stand-alone device, similar to the majority of clinical devices.

Overdoses are possible even with the built-in hazard mitigation. One of the main causes of these overdoses is the PCA pump's ignorance of the patient's vital signs, which are tracked by devices like pulse oximeter and respiration monitor. When the monitoring devices do not communicate with the PCA pump, infusion may continue even if the patient's health is entering respiratory depression.

Several studies have investigated integration of medical devices in a closed-loop with a controlling application to coordinate among the monitoring devices and the PCA pump [4,5]. Opioid infusion can be reduced or halted based on vital signs monitored by devices like pulse oximeter or respiration monitor. However, these studies assume devices communicate through a perfect communication network. The IoMT-based automated opioid infusion system we are considering also takes into account the fact that there is a potential of data loss, unauthenticated access, and information delay due to imperfect communication networks.

4. Structural Modeling using AADL

The goal of trusted composition of IoMT is to attain a high level of assurance within the system, acknowledging the potential presence of certain components with lower assurance. This goal is pursued by prioritizing high assurance for safety-critical components of the system and then integrating lower assurance components in a manner that upholds the overall system's safety.

Figure 2 shows the top-level AADL model of the clinical example use case explained in the previous section. This IoMT-based automated monitoring of opioid infusion, modeled as an AADL system `SysIoMT`, is composed of four sub-systems and a communication network. Each system is

 doi:10.20944/preprints202405.0908.v1

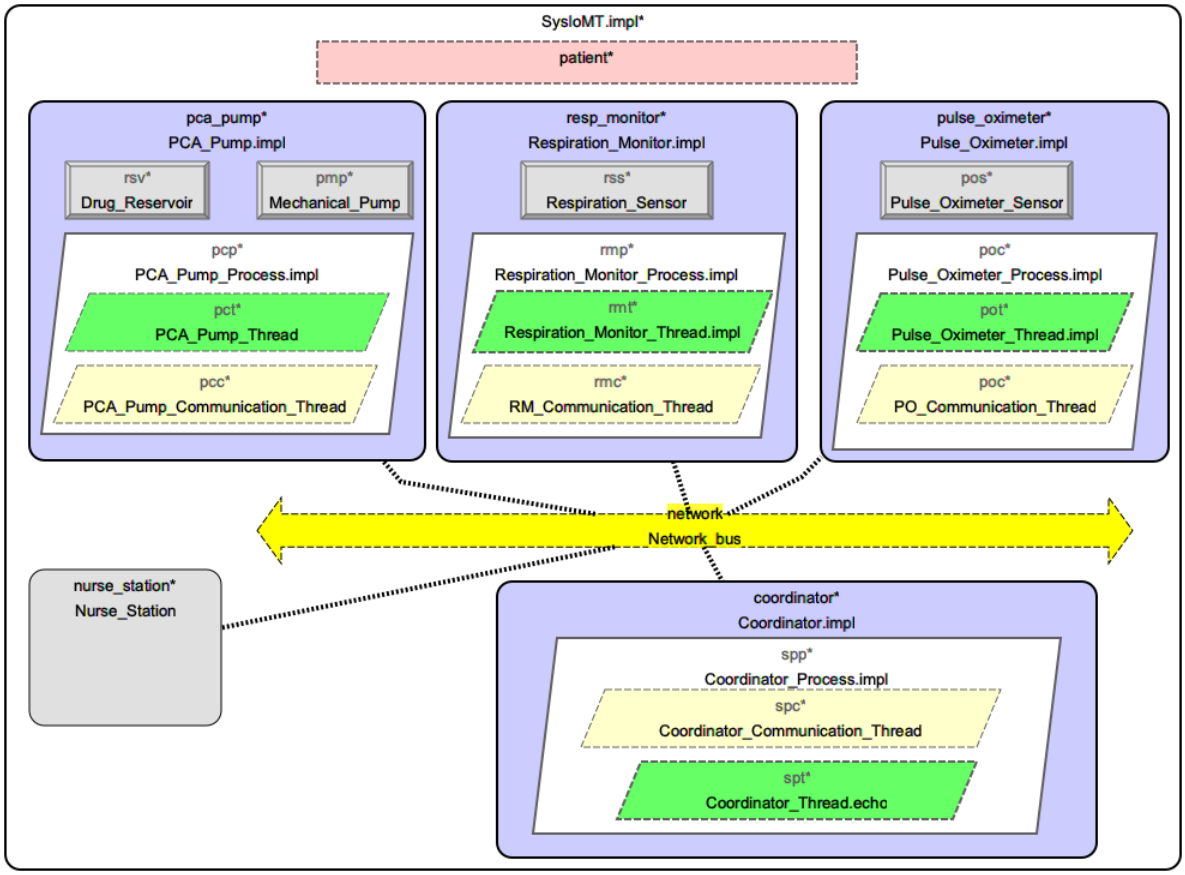

Figure 3. Component containment hierarchy

connected to an imperfect communication network which is modeled with an AADL virtual bus. A `Patient` is modeled as an abstract component. Alarms due to patient vital signs (low heart rate, respiration rate, or blood oxygenation) and warnings that assurance has been lost are sent to a `Nurse_Station` for human intervention. High assurance for the system as a whole is achieved by "loop-back" communication between formally-verified threads. Although any single message will not be assured, when the sender receives a correct copy of its message back, it is assured that these two messages (original and confirmation) have been performed correctly. Pairs of connections using loop-back are shown in red in Figure 2.

Figure 3 shows the containment hierarchy of all components in the model plus bus access connections. Formally-verified components have a light green background color; less-assured components, have a light yellow background. The `Respiration_Monitor` component models the respiration monitoring system attached to the patient for monitoring the respiration rate. It determines respiration rate from a respiration sensor usually strapped around a patient's chest. If the respiration rate is lower than the minimum respiration rate, `Respiration_Monitor` sends `RM_Alarm(true)` to the control application `Coordinator`, otherwise it sends `RM_Alarm(false)`. The `Respiration_Monitor` then waits for message confirmation response, either `RM_Alarm_Echo(true)` or `RM_Alarm_Echo(false)`. If the response is not received within the specified time limit (indicating a delay or data loss) or the received response is different from the sent notification (indicating message corruption), the `Respiration_Monitor` sends a lost confirmation signal through the `Lost_Confirmation` outgoing event port to the `Nurse_Station`. The `Respiration_Monitor` can also make a sound indicating a low respiration rate alarm or lost confirmation.

The patient's heart rate and blood oxygenation (SpO2) are monitored by the pulse oximeter, which is modeled by the `Pulse_Oximeter` system component. The `Pulse_Oximeter` receives values of the SpO2 and heart rate from the pulse oximeter sensor attached to the patient body. Through the

 doi:10.20944/preprints202405.0908.v1

PO_Alarm outgoing event data port, the Nurse_Station is alerted if any of these two parameters falls below minimum limits. Additionally, the Coordinator also receives the alarm. Next, the Pulse_Oximeter waits at PO_Alarm_Echo incoming event data port for the confirmation response from the Coordinator . Nurse_Station is notified through Lost_Confirmation outgoing event data port if the response is not received within the allotted time limit.

The Coordinator system component represents the control application executed by a processor on the network (not shown). The Coordinator notifies the PCA_Pump to reduce infusion to the minimal keep-vein-open (KVO) rate when it receives alarm signals from the Pulse_Oximeter or Respiration_Monitor through PO_Alarm or RM_Alarm , respectively. The acknowledgements are sent to Pulse_Oximeter and Respiration_Monitor through PO_Alarm_Echo and RM_Alarm_Echo ports. Nurse_Station receives notification through Lost_Confirmation outgoing event data port if the response is not received within a specified time frame or does not match the notification.

PCA_Pump models the PCA pump used to infuse drug into the patient. Because patients respond to opioids so differently, a steady basal rate of drug flow can be augmented by a patient-requested bolus by pressing the bolus request button. This puts the patient in-the-loop for their pain management. Too much opioid can cause respiratory depression which may lead to brain damage or death.

Nurse_Station system models the nursing station with human caregivers. It receives alarms and lost confirmation notifications on Patient_Alarm and Patient_Lost_Confirmation ports, respectively. Modeling the detailed functionality of the nursing station is beyond the scope of this study.

When PCA_Pump receives Pause_Infusion(true) from the Coordinator it reduces infusion to a KVO rate until it receives Pause_Infusion(false) to resume basal rate infusion. Every message received on Pause_Infusion is sent back on Pause_Infusion_Echo.

As shown in Listing 1, the BLESS invariant << IS_SAFE() >> in the properties section of the SysIoMT system type specifies the main safety property of the systems to be proved correct. Invariant IS_SAFE() expresses that in the event that the heart rate, SpO2, or respiration rate fall below the minimal threshold, the pump infusion will be adjusted to the KVO rate or will be halted. Section 5 provides further details on the assertions and Section 8 presents the proof details of this invariant.

Listing 1: System type for IoMT system-of-systems

```
system SysIoMT
  properties
    BLESS::Invariant => "<< IS_SAFE() >>";
end SysIoMT;
...
<<IS_SAFE: : ((HEART_RATE < MINIMUM_HEART_RATE
      or SPO2 < PROGRAMMED_SPO2
      or RESPIRATION_RATE < MINIMUM_RESPIRATION_RATE)
      implies PUMP_RATE$() = KVO_RATE_property) or HALT() >>
```

Listing 2 presents partial a specification of IoMT system implementation classifier. The subcomponents section of the SysIoMT.impl (system implementation) defines contained subsystems while the connections section specifies the appropriate connections between them. Connections are functional communication pathways allocated to physical components which actually transmit messages by Actual_Connection_Binding properties.

5. Behavior Specification using BLESS Properties

Behavior of an AADL component can be completely expressed by BLESS::Assertion and BLESS::Value properties on its ports and a BLESS::Invariant property of the component as a whole.

BLESS::Assertion properties are predicates defining what is true about the value sent or received by a port. Similarly, BLESS::Value properties are expressions what is true about the value sent or received by a port. Outgoing ports *guarantee* what is true about values sent; incoming ports *assume*

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0908.v1

Listing 2: A partial specification of the IoMT system implementation

```
system implementation SysIoMT.impl
  subcomponents
    pca_pump: system PCA_Pump.impl;
    pulse_oximeter: system Pulse_Oximeter.impl;
    resp_monitor: system Respiration_Monitor.impl;
    coordinator: system Coordinator.impl;
    network: virtual bus Network_Bus;
    patient: abstract Patient;
    nurse_station: system Nurse_Station;
  connections
   rm_alarm: port resp_monitor.RM_Alarm -> coordinator.RM_Alarm
      {Actual_Connection_Binding => (reference (network));};
   rm_alarm_echo: port coordinator.RM_Alarm_Echo -> resp_monitor.RM_Alarm_Echo
      {Actual_Connection_Binding => (reference (network));};
   --tube and needle between pump and patient
   infusion_set: feature pca_pump.Infused_Drug -> patient.Infused_Drug;
   ...
end SysIoMT.impl;
```

what is true about values received. The BLESS::Invariant property of a composite component must be implied by the conjunction of BLESS::Invariant properties of its subcomponents.

Thread component behavior is implemented by a state-transition machine. That the thread's behavior upholds its specification requires a verification condition for each state and transition. Respiration Monitor is used for illustration of verification conditions for threads.

5.1. Respiration Monitor

As depicted in Figure 4, Respiration_Monitor contains Respiration_Sensor device to measure the respiration rate and Respiration_Monitor_Process process (protected address space) which has two threads, Respiration_Monitor_Thread and RM_Communication_Thread.

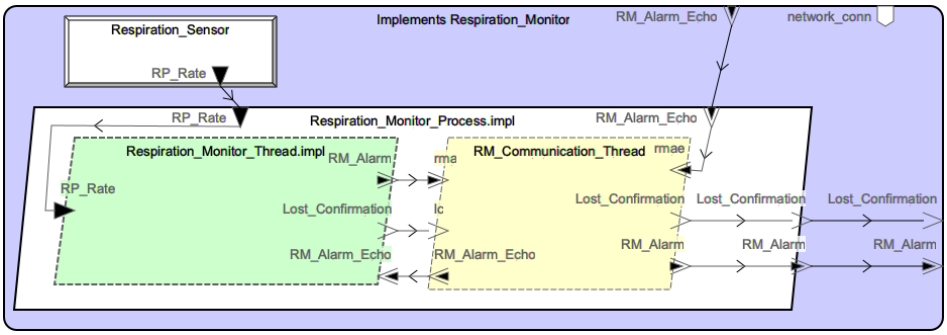

Figure 4. Respiration Monitor system structure

RM_Communication_Thread models COTS software such as a TCP/IP stack and device drivers. As such RM_Communication_Thread is less assured. Respiration_Monitor_Thread performs a safety-critical function so is formally verified with a deductive correctness proof. The type classifier of Listing 3 declares the interfaces, in terms of in and out ports, of the Respiration_Monitor_Thread .

BLESS::Assertion properties define predicates within << >> which must match the values of boolean-typed ports. Therefore port RM_Alarm outputs true when << RM_ALARM() >> holds and false otherwise. Because RM_Alarm is an out port, thread Respiration_Monitor_Thread guarantees its BLESS::Assertion property. Conversely, BLESS::Assertion with in port is assumed. Connections from out ports to in ports have assume-guarantee verification conditions (further explained in Section 7). BLESS::Value properties define expressions which must match the values sent or received on data ports and event data ports. Port RP_Rate expects a value of type brpm with value RESPIRATION_RATE, which is a *ghost* variable. Ghost variables represent the actual values sent or received by ports.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0908.v1

Listing 3: Respiration Monitor thread type

```
thread Respiration_Monitor_Thread
  features
    RP_Rate: in data port IoMT_Types::Respiration_Rate
      {BLESS::Value => "<< returns quantity brpm := RESPIRATION_RATE >>";};
    Lost_Confirmation: out event port;
    RM_Alarm: out event data port BLESS_Types::Boolean
      {BLESS::Assertion => "<< RM_ALARM() >>";};
    RM_Alarm_Echo: in event data port BLESS_Types::Boolean;
  properties
    Dispatch_Protocol => Timed;
    Period => 20 sec;
end Respiration_Monitor_Thread;
```

Listing 4: Repiration Monitor invariant, variables, and states sections

```
thread implementation Respiration_Monitor_Thread.impl
  annex BLESS
  {**
  invariant
     << RA() and MRP_A()>>
  variables
    rpr~quantity brpm; --measured respiration rate
    mrp~quantity brpm := 2.0 brpm  --minimum respiration rate
      << MRP_A: : mrp = MINIMUM_RESPIRATION_RATE >>;
    rma~boolean := false  --respiration monitor alarm
      <<RA: : rma iff (RESPIRATION_RATE < MINIMUM_RESPIRATION_RATE) >>;
  states
    start: initial state << RA() and MRP_A() >>
    run: complete state << RA() and MRP_A() >>
    check_echo:   complete state << RA() and MRP_A() >>
```

6. Behavior Implementation using BLESS State-Transition Machine

In AADL, component implementations are separate from component types. Multiple implementations of a same component type with distinct identifiers are permitted by AADL. The names of implementations are the identifier of the type classifiers followed by a period, then the identifier of the component classifiers. For example, `Respiration_Monitor_Thread.impl` is an implementation of the thread type `Respiration_Monitor_Thread`.

Inside a component implementation, a BLESS state-transition machine is written within a BLESS annex subclause. AADL allows definition of annex subclauses with the reserved word `annex` followed by the identifier of the annex, the body of the annex subclause between `{**` and `**};` . OSATE can be extended with plugins to provide parsers, syntax highlighting, and error markers for an annex sublanguage in the body of an annex subclause. To include a BLESS annex subclause to express thread implementation by a state-transition machine, the annex language name `BLESS` must be the identifier of the annex subclause.

6.1. Invariant, Variables, and States

Listing 4 shows the assertion, invariant, and variable specification of the `Respiration_Monitor_Thread`. An assertion defined in the `invariant` section must be true whenever the thread suspends or dispatches.

The `states` section declares behavior states which may have assertions which are true when the state-transition machine is in that state. There are four kinds of states: `initial` , `final` , `complete` , or execution if not otherwise indicated. Execution starts in the `initial` state and ends in a `final` state (if any). Computation suspends upon entering a `complete` state (execution is complete) until the thread is next dispatched. Execution states are transitory.

 doi:10.20944/preprints202405.0908.v1

Local variables that hold persistent values are declared in the `variables` section. Variable declarations may include an assertion which defines the meaning of the variable.

6.2. Transitions

Transitions have a label, a source state, a transition condition, a destination state, and optionally, an action.Transitions leaving execution states have boolean expression for transition conditions, while transitions leaving `complete` states have dispatch conditions. Listing 5 shows transitions implementing the `Respiration_Monitor_Thread` depicted in Figure 5.

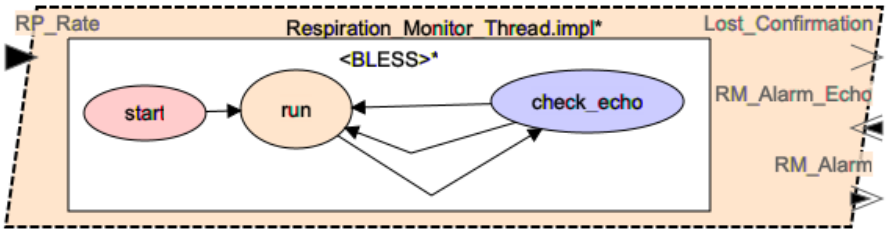

Figure 5. Respiration Monitor state machine

- Transition `go` , with empty execute condition being true, changes state from `start` to `run` .
- Transition `got_rp` , with dispatch condition `on dispatch` , leaves the `run` state periodically (every 20 seconds), performs an action, then changes state to `check_echo` . Its action reads the current respiration rate, sends an alarm (true) if less than the minimum respiration rate, otherwise sends alarm false.
- Transition `got_echo` , with dispatch condition `on dispatch RM_Alarm_Echo` , leaves the `check_echo` state when an event is received on port `RM_Alarm_Echo` , performs an action, then changes state to `run` . Its action compares the boolean value of `RM_Alarm_Echo` with the alarm value sent during the action of transition `got_rp` , and sends an event out port `Lost_Confirmation` if different.
- Transition `late_echo` also leaves `check_echo` state, and changes state to `run` after sending an action out port `Lost_Confirmation` , but only if its timeout dispatch condition is met. The timeout starts when a value is sent from port `RM_Alarm` . If `RM_Alarm_Echo` has not been received within `RM_Echo_Limit` seconds later, the dispatch condition has been met.

Thus `Respiration_Monitor_Thread.impl` employs three different kinds of dispatch conditions.

7. Verification Conditions

Verification conditions are theorems which express what must be true for system behavior to meet its specification. Verification conditions are either for correctness of individual threads, or for compositions of subcomponents. Proving all verification conditions for a system constitutes correctness proof of the system.

Of course verification conditions may be incorrectly generated, or incomplete, which would invalidate a correctness proof. Arguments that the verification conditions are both correct and complete can be found in [14].

Each verification condition (and all other theorems in a correctness proof are Hoare triples in the form of

$$\ll P \gg \ S \ \ll Q \gg$$

where P is a precondition, S is an action and Q is a postcondition. When there is no action, the verification condition reduces to $P \rightarrow Q$.

7.1. Thread Verification Conditions

For threads a verification condition is generated for each state (except final states) and a verification condition is generated for each transition.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0908.v1

Listing 5: Repiration Monitor transitions section

```
  transitions
    go: start -[ ]-> run {}
    got_rp: run -[ on dispatch ]-> check_echo
      {
      RP_Rate?(rpr)
        << rpr = RESPIRATION_RATE and MRP_A() and RA() >>
      ;
      if(rpr >= mrp)~>
        { << rpr = RESPIRATION_RATE and rpr >= mrp and MRP_A() and RA() >>
        rma:=false & RM_Alarm!(false)
        << RA() and MRP_A() >> }
      []
      (rpr < mrp)~>
        { << rpr = RESPIRATION_RATE and rpr < mrp and MRP_A() and RA() >>
        rma:=true & RM_Alarm!(true)
        << RA() and MRP_A() >> }
      fi
      } -- end of got_rp
    got_echo:  check_echo -[ on dispatch  RM_Alarm_Echo]-> run
      { << RA() and MRP_A() and RM_Alarm_Echo@now >>
      declare echo_val~boolean := RM_Alarm_Echo?
        { << RA() and MRP_A() and (echo_val iff RM_Alarm_Echo) >>
        if (echo_val iff rma)~> skip  --echo was same
        []
        (not (echo_val iff rma))~> Lost_Confirmation!  --echo was different
        fi
          << RA() and MRP_A() >>
        }
        << RA() and MRP_A() >>
      } -- end of got_echo
    late_echo: check_echo -[on dispatch timeout (RM_Alarm  or RM_Alarm_Echo)
        IoMT_Properties::RM_Echo_Limit s ]-> run { Lost_Confirmation! }
  **};
end Respiration_Monitor_Thread.impl;
```

For each complete state, its assertion must imply the thread invariant. The invariant for `Respiration_Monitor_Thread.impl` is `<< RA() and MRP_A() >>`. State `run` has the the same assertion. So the verification condition for complete state `run` is a tautology as shown in Listing 6.

Listing 6: Verification condition for run complete state

```
[serial 1012]: IoMTwin::Respiration_Monitor_Thread.impl
P [428] << RA() and MRP_A() >>
S [418]->
Q [418] << RA() and MRP_A() >>
Why created: <<M(check_echo)>> -> <<I>> from invariant I when
complete state check_echo has Assertion <<M(check_echo)>> in its definition.
```

For each execution state, its assertion must imply the disjunction of of the transition conditions of all transitions for which that state is its source.

For each transition, a verification condition is generated as

$$\ll P \wedge b \gg \ S \ \ll Q \gg$$

where P is the assertion of the source state, Q is the assertion of the destination state, b is the transition condition, and S is the action of the transition. The verification condition for transition `go` is generated as shown in Listing 7.

Numbers in the square brackets are the line numbers in the program. Within `IoMTwin.aadl`, which has complete AADL specification, line 426 has definition of the `start` state, and line 427 has definition of the destination state `run`. The transition `go` is defined at line 431 without any behavior action.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0908.v1

Listing 7: Verification condition for transition go

```
[serial 1014]: IoMTwin::Respiration_Monitor_Thread.impl
P [426] << RA()
  and MRP_A() >>
S [431]->
Q [427] << RA()
  and MRP_A() >>
Why created:     <<M(start)>> -> <<M(run)>> for go: start-[ ]->run{};
```

Transitions leaving complete states have dispatch conditions which makes the transition condition b more complex. The dispatch condition holds when dispatched at time `now` and dispatch has not happened since the time-of-previous-dispatch, `tops` . The precondition for the verification condition generated for `got_echo` (shown in Listing 8) includes `RM_Alarm_Echo@now` which means event arrival at the `RM_Alarm_Echo` port in the present instant.

Listing 8: Verification condition for transition got_echo

```
[serial 1016]: IoMTwin::Respiration_Monitor_Thread.impl
P [454] << ( RA()
  and MRP_A() )
  and ( RM_Alarm_Echo@now )
  and not ( exists u ~ time
   in tops ,, now
   that RM_Alarm_Echo@u  )
  and not ( exists u ~ time
   in tops ,, now
   that ( RM_Alarm or RM_Alarm_Echo )@( u - IoMT_Properties::RM_Echo_Limit s )
  and not ( exists t ~ time
   in ( u - IoMT_Properties::RM_Echo_Limit s ) ,, u
   that ( RM_Alarm or RM_Alarm_Echo )@t  )  ) >>
S [456]<< RA() and MRP_A() and RM_Alarm_Echo@now >>
declare
   echo_val ~ boolean := RM_Alarm_Echo?   {
  << RA and MRP_A() and ( echo_val iff RM_Alarm_Echo ) >>
    if
      (echo_val iff rma)~>
        skip
    []
      (not ( echo_val iff rma ))~>
        Lost_Confirmation!
    fi
  << RA() and MRP_A() >>
  }
<< RA() and MRP_A() >>
Q [427] << RA() and MRP_A() >>
Why created: <<M(check_echo) and x>> A <<M(run)>> for
got_echo: check_echo-[x]->run{A};
```

7.2. Composition Verification Conditions

The behavior of a system-of-systems, such as the IoMT in our case study, can be proved correct to meet its specifications based on the proved-correct specifications of its subsystems. As a result verification conditions are generated for each connection implying assume-guarantee contracts among subsystems. Additionally, the conjunction of subcomponents' invariants must imply the invariant of the system-of-systems.

Listing 9 presents a verification condition generated for connection `rm_alarm` between `Respiration_Monitor` and `Coordinator` subsystems (see Listing 2). In all most all the circumstances the source and destination of a connection have same assertions, hence generating a tautology.

 doi:10.20944/preprints202405.0908.v1

Listing 9: Verification condition for connection rm_alarm

```
[serial 1046]: resp_monitor.rmp.rmc.RM_Alarm -> coordinator.spp.spc.rma
P [1] << RM_ALARM() >>
S [2]->
Q [1] << RM_ALARM() >>
Why created:    Composition of Subcomponents via Directional Connection
resp_monitor.rmp.rmc.RM_Alarm -> coordinator.spp.spc.rma: RM_Alarm -> rma
```

Verification condition generated for the IoMT system property `<< IS_SAFE() >>` specified in Listing 1, is shown in Listing 10. It states that the conjunction of invariants of subsystems implies the invariant of system-of-systems (IoMT in our case study).[1]

Listing 10: Verification condition for system property IS_SAFE()

```
[serial 1028]: IoMTwin::SysIoMT.impl
P [1] << ( PAUSE_INFUSION implies PUMP_RATE$() = KVO_RATE_property )
  and ( ( PO_ALARM() or RM_ALARM() ) iff PAUSE_INFUSION ) >>
S ->
Q [2] << IS_SAFE() >>
Why created:    Conjunction of Subcomponents' Invariants imply
SysIoMT.impl's Invariant
```

8. Proof of Verification Conditions

Behavior correctness proof requires to solve all the verification conditions generated based on assertion specifications with ports, states, transitions, and behavior actions. The BLESS proof tool produces the formal proof of the behavior correctness as a list of theorems when all the verification conditions are solved. Each theorem is either a tautology, given, or derived from earlier theorems by applying inference rules. Like other interactive theorem provers, the BLESS proof tool requires human intervention to select proof tactics. Complete proof of the `SysIoMT` requires 332 theorems. Below we present some of the proof theorems used to solve verification conditions generated for the `Respiration_Monitor_Thread` , and for the composition of the system-of-systems. Complete proof of the `SysIoMT` system with all the subsystems, processes, and threads is available as a GitHub repository at [15].

8.1. Thread Verification Condition Proof

Table 1 presents verification conditions along with proof theorems generated for each state and transition for `Respiration_Monitor_Thread` . For example, for `complete` state `run` verification condition `[serial 1012]` (shown in Listing 6) is generated and proof theorem (93) is used to solve it .

Similarly, for transition `go` verification condition `[serial 1014]` is generated and theorem (95) is used to solve it as tautology. For a relatively complex transition `got_echo` , verification condition `[serial 1016]` (shown in Listing 8) is generated which becomes theorem (153), and is proved by theorems (132) to (152) are used to solve it. Listing 11 presents a couple of sample theorems for transition `got_echo` .

[1] For each verification condition and theorem, the `Why created:` part states the reason for generating this verification condition and the `Solved by:` part (only in theorems) states the proof tactics used to solve this verification condition.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 May 2024 doi:10.20944/preprints202405.0908.v1

Table 1. Verification conditions and Theorems Respiration Monitor Thread

	Identifier	Verification Conditions	Proof Theorems
States	start	1013	94
	run	1012	93
	check_echo	1011	92
Transitions	go	1014	95
	got_rp	1015	96—131
	got_echo	1016	132—153
	late_echo	1017	154—159

Listing 11: First and the last proof theorem for transition got_echo

```
Theorem (132)                                  [serial 1510]
P [454] << ( MRP_A() and RA() ) and RM_Alarm_Echo@now and
not ( exists u ~ time in tops ,, now that RM_Alarm_Echo@u  )
  and not ( exists u ~ time in tops ,, now that
  ( RM_Alarm or RM_Alarm_Echo )@( u - IoMT_Properties::RM_Echo_Limit s )
  and not ( exists t ~ time
   in ( u - IoMT_Properties::RM_Echo_Limit s ) ,, u
   that ( RM_Alarm or RM_Alarm_Echo )@t  )  ) >>
S [454] ->
Q [456] << RM_Alarm_Echo@now and MRP_A() and RA() >>
Why created:    normalization of [serial 1497]
  Using: ADD_UNNECESSARY_PARETHESES (bl.ancom)
Solved by: Absorb parentheses, middle term in conjunction (bl.anabpm)
...
Theorem (153)                                  [serial 1016]
P [454] << ( RA() and MRP_A() ) and ( RM_Alarm_Echo@now ) and not
( exists u ~ time in tops ,, now  that RM_Alarm_Echo@u  ) and not
( exists u ~ time in tops ,, now
   that ( RM_Alarm or RM_Alarm_Echo )@( u - IoMT_Properties::RM_Echo_Limit s )
  and not ( exists t ~ time  in ( u - IoMT_Properties::RM_Echo_Limit s ) ,, u
   that ( RM_Alarm or RM_Alarm_Echo )@t  )  ) >>
S [456] << RA() and MRP_A() and RM_Alarm_Echo@now >>
declare
   echo_val ~ boolean := RM_Alarm_Echo?   {
  << RA() and MRP_A() and ( echo_val iff RM_Alarm_Echo ) >>
    if
      (echo_val iff rma)~>
        skip
    []
      (not ( echo_val iff rma ))~>
        Lost_Confirmation!
    fi
  << RA() and MRP_A() >>
  }
<< RA() and MRP_A() >>
[427] << RA() and MRP_A() >>
Why created:   <<M(check_echo) and x>> A <<M(run)>> for
got_echo: check_echo-[x]->run{A};
Solved by: null and theorems 133 135 152:
Theorem (133) [serial 1497] used for:
  as <<P>> -> <<pre>> in <<P>> { <<pre>> S <<post>> } <<Q>> for [serial 1016]
Theorem (135) [serial 1498] used for:
  as <<post>> -> <<Q>> in <<P>> { <<pre>> S <<post>> } <<Q>> [serial 1016]
Theorem (152) [serial 1499] used for:
  as <<pre>> S <<post>> in <<P>> { <<pre>> S <<post>> } <<Q>> [serial 1016]
```

8.2. Composition Verification Condition Proof

For the assume-guarantee relationships on connections, the verification conditions (like the one presented in Listing 9) are easily solved as tautologies. In order to show what is true about the IoMT

 doi:10.20944/preprints202405.0908.v1

system-of-systems is derived from what is true about its subsystems. The compositional verification condition generated for the system property `IS_SAFE()` (presented in Listing 10) is solved using the three theorems (600), (601), and (602), as shown in Listing 12.

Listing 12: Proof theorems for system property IS_SAFE()

```
Theorem (600)                                [serial 1741]
P [1] << ( ( ( HEART_RATE < MINIMUM_HEART_RATE or SPO2 < PROGRAMMED_SPO2 ) or
RESPIRATION_RATE < MINIMUM_RESPIRATION_RATE ) iff PAUSE_INFUSION )
  and ( PAUSE_INFUSION implies KVO_RATE_property = PUMP_RATE$() ) >>
S    ->
Q [2] << ( ( HEART_RATE < MINIMUM_HEART_RATE or RESPIRATION_RATE <
MINIMUM_RESPIRATION_RATE or SPO2 < PROGRAMMED_SPO2 ) implies
KVO_RATE_property = PUMP_RATE$() ) or HALT() >>
Why created:    normalization of [serial 1739]
  Using: (eqcom) ADD_UNNECESSARY_PARETHESES (bl.ancom) (bl.orcom)
Solved by: Absorb parentheses, middle term in disjunction (bl.orabpm)

Theorem (601)                                [serial 1739]
P [1] << ( PAUSE_INFUSION implies PUMP_RATE$() = KVO_RATE_property )
  and ( ( ( HEART_RATE < MINIMUM_HEART_RATE or SPO2 < PROGRAMMED_SPO2 ) or
( RESPIRATION_RATE < MINIMUM_RESPIRATION_RATE ) ) iff PAUSE_INFUSION ) >>
S    ->
Q [2] << ( ( ( HEART_RATE < MINIMUM_HEART_RATE or SPO2 < PROGRAMMED_SPO2 or
RESPIRATION_RATE < MINIMUM_RESPIRATION_RATE ) implies PUMP_RATE$() =
KVO_RATE_property ) or HALT() ) >>
Why created:  Substituted assertions' predicates for labels  [serial 1028]
Solved by: Equality Commutes. (eqcom) [Add unnecessary parentheses]
Conjunction Commutes. (bl.ancom) Disjunction Commutes. (bl.orcom) and
theorem 600:
Theorem (600) [serial 1741] used for: normalization of [serial 1739]


Theorem (602)                                [serial 1028]
P [1] << ( PAUSE_INFUSION implies PUMP_RATE$() = KVO_RATE_property )
  and ( ( PO_ALARM() or RM_ALARM() ) iff PAUSE_INFUSION ) >>
S    ->
Q [2] << IS_SAFE() >>
Why created:  Conjunction of Subcomponents' Invariants imply SysIoMT.impl's
Invariant
Solved by: Predicate Invocation. (PI) and theorem 601:
Theorem (601) [serial 1739] used for:
  Substituted assertions' predicates for labels  [serial 1028]
```

9. Conclusion

The Internet of Medical Things (IoMT) integrates medical devices, applications, and IT systems through communication networks, enabling comprehensive monitoring and control capabilities. However, this convergence poses challenges, notably security vulnerabilities and network disruptions, due to reliance on imperfect communication networks. Crucially, achieving trusted composition in IoMT systems involves prioritizing high assurance in critical functional components while accommodating lower-assured communication infrastructure. Integration of AADL with BLESS behavior specification is investigated to offer a systematic approach to ensure trusted composition of heterogeneous medical devices over such imperfect networks. This approach emphasizes formal verification techniques and a message-confirmation protocol to establish assume-guarantee relationships and enhance safety properties at both individual device and system levels. A case study demonstrates the application of the proposed approach in an IoMT system of a patient-controlled opioid infusion comprising a pulse oximeter, respiration monitor, PCA pump, and coordinator application. Despite the presence of an imperfect communication network, the study illustrates the feasibility of safe and correct information

 doi:10.20944/preprints202405.0908.v1

exchange towards a trusted composition. Enhancing trusted composition techniques can improve IoMT system scalability and dependability ultimately leading to better healthcare delivery.

Acknowledgments: The authors extend their appreciation to the Deanship of Scientific Research at Saudi Electronic University for funding this research (9460).

References

1. Minerva, R.; Biru, A.; Rotondi, D. Towards a definition of the Internet of Things(IoT). Tech. report, IEEE Internet of Things, Telecom Italia S.p.A, 2015.
2. Hao, A.; Wang, L. Medical Device Integration Model Based on the Internet of Things. *Open Biomed Eng J* **2015**, *9*, 256–261.
3. Dinh-Le, C.; Chuang, R.; Chokshi, S.; Mann, D. Wearable Health Technology and Electronic Health Record Integration: Scoping Review and Future Directions. *JMIR Mhealth Uhealth* **2019**, *7*, e12861. doi:10.2196/12861.
4. Arney, D.; Pajic, M.; Goldman, J.M.; Lee, I.; Mangharam, R.; Sokolsky, O. Toward patient safety in closed-loop medical device systems. Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems; Association for Computing Machinery: New York, NY, USA, 2010; ICCPS '10, p. 139–148. doi:10.1145/1795194.1795214.
5. Larson, B.R.; Zhang, Y.; Barrett, S.C.; Hatcliff, J.; Jones, P.L. Enabling Safe Interoperation by Medical Device Virtual Integration. *IEEE Design & Test* **2015**, *32*, 74–88. doi:10.1109/MDAT.2015.2464813.
6. Sicari, S.; Rizzardi, A.; Grieco, L.; Coen-Porisini, A. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks* **2015**, *76*, 146–164. doi:https://doi.org/10.1016/j.comnet.2014.11.008.
7. Ghubaish, A.; Salman, T.; Zolanvari, M.; Unal, D.; Al-Ali, A.; Jain, R. Recent Advances in the Internet-of-Medical-Things (IoMT) Systems Security. *IEEE Internet of Things Journal* **2021**, *8*, 8707–8718. doi:https://doi.org/10.1109/JIOT.2020.3045653.
8. Ahmed, S.F.; Alam, M.S.B.; Afrin, S.; Rafa, S.J.; Rafa, N.; Gandomi, A.H. Insights into Internet of Medical Things (IoMT): Data fusion, security issues and potential solutions. *Information Fusion* **2024**, *102*, 102060. doi:https://doi.org/10.1016/j.inffus.2023.102060.
9. Bharati, S.; Podder, P.; Mondal, M.R.H.; Paul, P.K., Applications and Challenges of Cloud Integrated IoMT. In *Cognitive Internet of Medical Things for Smart Healthcare: Services and Applications*; Hassanien, A.E.; Khamparia, A.; Gupta, D.; Shankar, K.; Slowik, A., Eds.; Springer International Publishing: Cham, 2021; pp. 67–85. doi:10.1007/978-3-030-55833-8_4.
10. Hatcliff, J.; King, A.; Lee, I.; Macdonald, A.; Fernando, A.; Robkin, M.; Vasserman, E.; Weininger, S.; Goldman, J.M. Rationale and Architecture Principles for Medical Application Platforms. 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems, 2012, pp. 3–12. doi:10.1109/ICCPS.2012.9.
11. Software Engineering Institute, Carnegie Mellon University. OSATE2. http://osate.org/. [Accessed: (May 05, 2024)].
12. Larson, B.R.; Chalin, P.; Hatcliff, J. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software. In *NASA Formal Methods*; 2013; Vol. 7871, *LNCS*, pp. 276–290.
13. Boydston, A.; Feiler, P.H. Architecture Centric Virtual Integration Process (ACVIP) : A Key Component of the DoD Digital Engineering Strategy. 2019.
14. Larson, B.R. *Behavior Language for Embedded Systems with Software Language Reference and Proof Assistant Guide*; 2024. https://multitude.net/documentation/[Accessed: (May 05, 2024)].
15. Ahmad, E.; Larson, B. AADL Model, BLESS Behavior Specification, and Verification of IoMT-based Opioid Infusion Monitoring, 2024. https://github.com/brlarson/IoMT [Accessed: (May 05, 2024)].