Logic-Based Formalization of System Requirements for Integrated Clinical Environments

Cinzia Bernardeschi, Andrea Domenici, and Paolo Masci

Abstract The concepts of *integrated clinical environments* and *smart intensive care units* refer to complex technological infrastructures where health care relies on inter-operating medical devices monitored and co-ordinated by software applications under human supervision. These complex socio-technical systems have stringent safety requirements that can be met with rigorous and precise development methods. This chapter presents an approach to the formalization of system requirements for integrated clinical environments, using the Prototype Verification System, a theorem-proving environment based on a higher-order logic language. The approach is illustrated by modeling safety-related requirements affecting various aspects of an integrated clinical environments, and in particular the communication network. A simple but realistic wireless communication protocol will be used as an example of computer-assisted verification.

Cinzia Bernardeschi
Department of Information Engineering, University of Pisa,
Largo Lazzarino 1, 56122 Pisa, Italy
Phone +39 2217674
e-mail: `cinzia.bernardeschi@unipi.it`

Andrea Domenici
Department of Information Engineering, University of Pisa,
Largo Lazzarino 1, 56122 Pisa, Italy
Phone +39 2217541
e-mail: `andrea.domenici@unipi.it`

Paolo Masci
HASLab, INESC TEC & Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
e-mail: `paolo.masci@inesctec.pt`

1 Introduction

Clinical care relies on a large number of biomedical instruments, ranging from relatively simple sensors to sophisticated and complex equipment, such as scanners for positron emission tomography or surgical robots. Most of these devices are currently operated under human supervision and as standalone components, i.e., without exchanging data or control signals with other devices, but there is a growing demand to integrate devices into a collaborative environment under computer-assisted supervision. Such integration would afford many benefits, such as enhanced safety and increased automatization of routine procedures.

A network of inter-operating devices can also interact with information systems, possibly cloud-based, to manage data on patients and therapies, gathering information needed both to care for the individual patient and to analyze large-scale statistics.

Concepts such as *Integrated Clinical Environment* (ICE) (F2761-2009, 2009) and *smart ICU* (Intensive Care Unit) (Halpern, 2014; Ahmed and Ali, 2016) have been formulated to characterize these new clinical settings where the interaction of human actors under different roles and complex equipment create a socio-technical system with stringent safety requirements. Figure 1 (adapted from (F2761-2009, 2009)) schematically shows the overall architecture of an ICE: Several devices ineracting with the patient are connected within a network, connected through a network controller to a *supervisor*, a software that executes actions requested by clinicians, reports patient and system status, and performs automatic interventions when certain safety conditions are violated. The system is connected to an external network to access databases, administrative services, or any other kind of data or services that may be needed. This schema is the reference model adopted in the present work.

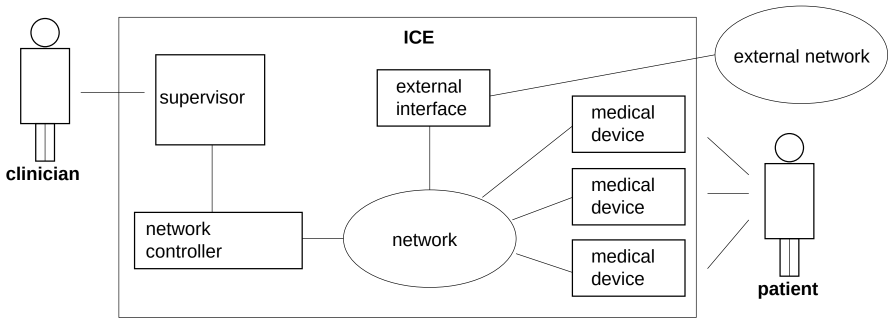

Fig. 1 An Integrated Clinical Environment (adapted from (F2761-2009, 2009)).

An example of device interconnection under supervision of a control application is shown in Fig. 2, where the medical application running on a handheld device enforces a safety interlock on the analgesic infusion pump based on respiratory data provided by the patient monitor.

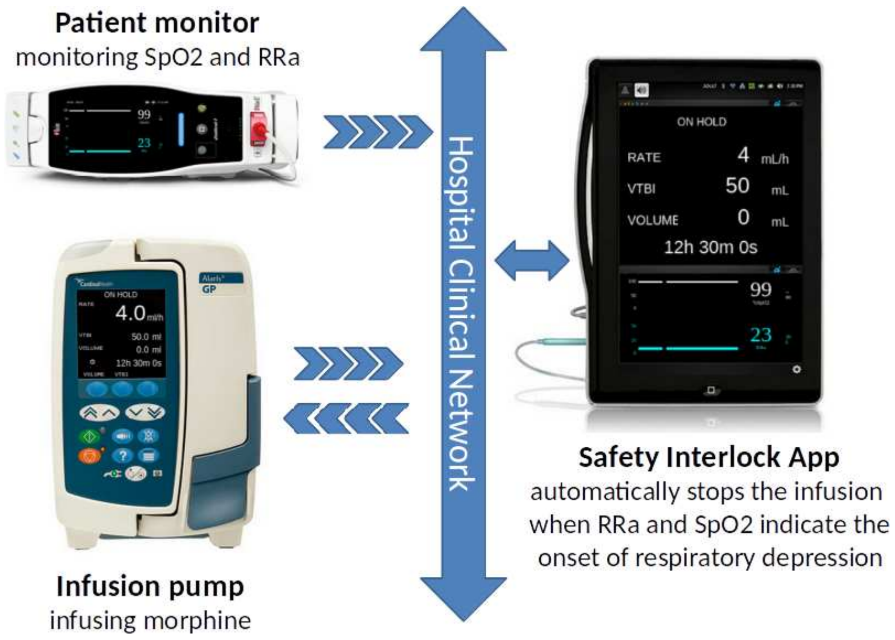

Fig. 2 An example of Integrated Clinical Environment.

Interoperability (AAMI MDI/2012-03-30, 2012; FDA Guidance, 2009) is a fundamental concern for this kind of systems, as it enables devices with different purposes and acquired from different vendors to communicate seamlessly. Clinical-oriented interoperability standards are being developed that formulate interoperability requirements (Kabachinski, 2006; Rhoads et al, 2010) on biomedical devices and clinical information systems. The *communication network* is clearly a critical component from the point of view of interoperability.

The present chapter, extending previous work (Bernardeschi et al, 2015, 2016), discusses the formalization of requirements for an ICE, using a higher-order logic language. It presents an overview of the proposed approach to domain identification and requirements specification (Sec. 4), and then its application to the ICE communication network. One objective of this work is showing how a logic specification language can be used to model a large and complex system in a modular way, where different parts and aspects of the system are specified by separate theories that together produce a safety reference model against which implementations can be checked for correctness. The implementations and the properties they must satisfy are defined in the same language as the reference model, at a different level of abstraction.

This chapter does not present a full specification for an ICE communication network, as it only aims at introducing the main ideas by showing some specification excerpts. It is assumed that the network connects medical devices and computers, using wireless technology. The nodes are assumed mobile and thus liable to sporadic loss of connectivity and other issues. Wired devices are taken to be logically equivalent to wireless ones, since they can generally be unplugged and moved.

The chapter is structured as follows: Section 2 reports related work, Section 3 introduces the specification language of PVS, Section 4 describes the proposed methodology to formalize the ICE requirements, Section 5 presents the PVS theories for the ICE communication network, in Section 6 the use of these theories for implementation and verification purposes is discussed, and Section 7 concludes the chapter.

2 Related work

Among the accepted and proposed standards related to the concept of interoperability of medical systems, we may cite the ASTM F2761 standard (F2761-2009, 2009), defining the high-level architecture of ICEs, and the AAMI White paper MDI/2012-03-30 (AAMI MDI/2012-03-30, 2012). Standard ANSI/HL7 V2.8.2-2015 (ANSI/HL7 V2.8.2-2015, 2015; Kabachinski, 2006) deals with data exchange among healthcare computer applications. Besides formal standards, a great number of papers address general issues in the area of interoperable medical systems. For example, a list of non-functional requirements or high-level guidelines for medical cyber-physical systems middleware has been proposed by Arney et al. (Arney et al, 2014), and Larson et al. (Larson et al, 2012) discuss requirements engineering for *medical application platform* software. Uses of ICE data for health technology management are discussed by Rausch and Judd (Rausch and Judd, 2016). We build on these works, which provide us with a set of relevant safety requirements, and show how a logic-based language can be used to formalize core aspects of the requirements.

Infusion pumps have been extensively used as an example of device deployed in an ICE. For example, a set of requirements for analgesic infusion pumps to be integrated in an interoperable environment has been proposed by Larson et al. (Larson et al, 2013; Larson and Hatcliff, 2014).

Many works address issues related to safe operation of interoperable medical environments. Venkatasubramanian et al. (Venkatasubramanian et al, 2015) discuss hazard analysis for requirements of an *Interoperability Alarm System* meant to signal interoperability failures of medical systems, taking airway laser surgery as a case study. A paper by Leite et al. (Leite et al, 2017) deals with safety assurance for Medical Cyber-Physical Systems of Systems and proposes an extension to Laprie's taxonomy (Avižienis et al, 2004) for dependability.

An example of implementation for the ICE architecture is the *OpenICE* platform, described by Arney et al. (Arney et al, 2017). While the OpenICE platform uses the OMG *Data Distribution Service* (Corsaro and Schmidt, 2012), García-Valls and Touahria (García-Valls and Touahria, 2017) discuss the integration of the iLAND communication middleware (García-Valls et al, 2013) in the ICE framework. Islam et al. (Islam et al, 2015) present a survey of applications and frameworks for health care based on the Internet of Things. A simulation-based study on the performance of medical device networks is presented by Arney et al. (Arney et al, 2012). Differently from our approach, these works focus more on ICE implementation options, rather than formalization of ICE requirements.

The application of verification methods to medical systems is drawing much attention in the research community. A position paper by Kühn and Leucker (Kühn and Leucker, 2014) is focused on the interconnection of devices in the operating room and proposes formal verification approaches, and Ray et al. (Ray et al, 2010) expose a verification strategy called *instrumentation-based verification* for the model-based development of medical cyber-physical systems. The PVS theorem proving environment (Owre et al, 1992) has been used for verification in many application fields, such as autonomous vehicles (Domenici et al, 2017) and nonlinear controls (Bernardeschi and Domenici, 2016). In the field of medical systems, PVS and the PVSio-web prototyping environment (Masci et al, 2015b) have been used to study implantable cardiac pacemakers (Bernardeschi et al, 2017, 2014) and infusion pumps (Mauro et al, 2017). The present chapter complements these works in that we demonstrate how formal methods technologies can be used to formalize network requirements for an ICE system.

3 The PVS specification language

Several types of systems have been formally specified in the language of the *Prototype Verification System* (PVS), including medical devices (Harrison et al, 2014; Masci et al, 2014, 2015b). A PVS specification is a *theory*, defining types, variables, constants, functions, and axioms and theorems. Functions and axioms define the assumed characteristics of the specified system, whereas theorems define properties that must be proved with respect to the theory. Demonstrations are carried out interactively with the PVS theorem prover.

The PVS language is based on higher-order logic, allowing functions to return functions and to be passed as function arguments.

The specification of a complex system is usually composed of a number of PVS theories, each related to a subsystem or to some aspect of the system that is not confined to a single subsystem. Further, the specification can rely on a large number of pre-defined library theories.

The PVS type system makes it possible to define types at any desired level of abstraction, specifying the properties of type members without any assumption on their implementation. It is also possible to specify *subtypes* that inherit the properties of their parent types.

The PVS language is purely declarative, but the PVSio extension (Muñoz, 2003) can compute PVS functions when supplied with fully instantiated arguments. Beside the evaluator (or interpreter) for functions, the PVSio extension includes a library for input, output, and numerical computation. These library fuctions can be used freely in a theory, since they do not affect in any way its semantics. In this way, a PVS theory can be both formally verified and executed, providing a prototyping capability.

Several samples of the PVS language occur in the following. In the samples, a type expression of the form "[$d\_type \to r\_type$]" denotes a function type, where *d_type* and *r_type* are the domain and range type, respectively, which can be any type, including other function types. The usual form for functions with multiple arguments, e.g., $f(x,y)$ and the Curried form, e.g., $f(x)(y)$ are equivalent. The keyword *FROM* denotes subtyping; subtyping can also be expressed by set comprehension, e.g., by an expression of the form $\{n : nat \mid odd(n)\}$, denoting the set of natural numbers n such that n is odd. Other syntactic details will be explained as needed.

The PVS theorem prover is based on the sequent calculus (Girard et al, 1990). A *sequent* has a structure of the form $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$, where the *turnstile* symbol '$\vdash$' separates the *antecedent* formulae on its left from the *consequents* on the right. A sequent is proved if any consequent is true, or any antecedent is false, or any formula occurs both as an antecedent and as a consequent. A formula to be proved (a *goal*) is represented as a sequent without antecedents, and its proof consists in applying various inference rules until one of the three above mentioned final sequent forms is obtained.

4 Overview of ICE Requirements Formalization

The requirements for an ICE are not confined to technical issues concerning medical devices, and not even to clinical issues concerning therapies or patient's conditions, as a large number of apparently unrelated details may affect safety. For example, errors in the managemant of patient identity data may cause a treatment to be delivered to the wrong patient. The management of personal data also entails issues on privacy and security, and so on. The requirements must then deal with disparate concerns, including alarm management, interface to patients, clinicians, and administrative personnel, device interconnection, and software dependability.

Formalizing such a large and heterogeneous set of requirements could easily lead to a cumbersome and unwieldy model that would be scarcely useful as a guide to implementation and a reference for verification. Therefore it is important to rely on specification methods that lead to the construction of well articulated and readable reference model, organized according both to the system's structural decomposition and to various abstraction levels. The rest of this section will introduce the main ideas about these specification methods.

4.1 Domain identification

Domain identification is the activity wherein the fundamental concepts in the application domain are recognized. The ICE application domain comprises several subdomains, each one representable at different levels of abstraction. These subdomains overlap many areas, such as, e.g., device usage and patient-related administrative procedures (e.g., related to patient identification). For example, the domain related to patient data and identification would include information available to clinicians to check that the right treatment is delivered to right patient, including, for example, patient ID, demographical data (name, age, etc.), and location. Such information could be modeled as in the following theory:

```
patient_theory: THEORY
begin
    patient: TYPE+
    patient_ID: TYPE+
    patient_location: TYPE+
    id(p: patient): patient_ID
    location(p: patient): patient_location
    ...
end patient_theory
```

In the above declarations, the *TYPE+* keyword declares *patient*, *patient_ID*, and *patient_location* to be non empty types. The other two declarations introduce *id* and *location* as functions from *patient* to *patient_ID* and from *patient* to *patient_location*, respectively.

Let us consider, as another example, the domain related to medical devices. An object-oriented domain analysis of this subdomain might contain such classes and relationships as shown in the UML diagram of Figure 3.

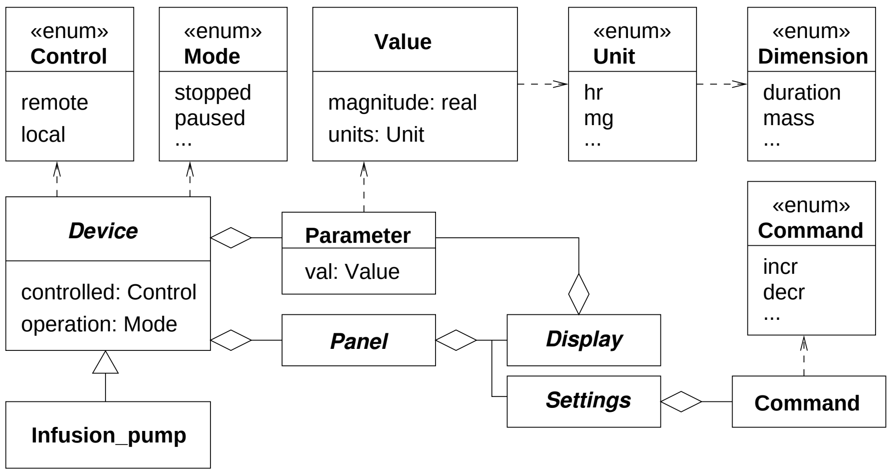

Fig. 3 Part of the domain analysis for the subdomain of medical devices.

From the diagram we gather that a device has two attributes, *controlled* and *operation*. The *controlled* attribute specifies if a device is being controlled remotely or locally, and accordingly can take one of the two values defined for the *Control* enumeration. The *operation* attribute specifies the current mode of operation, such as, e.g., *stopped* or *paused*. For simplicity, all possible modes of operation for all kinds of devices are listed in the *Mode* enumeration, although having different sets of operation modes for different device classes would make the analysis more modular.

A device is characterized by a set of *Parameter* values, such as infusion rate for an infusion pump, or heartbeat rate for a cardiac monitor. A parameter's *Value* has a numerical *magnitude* and a physical *unit*, which in turn is related to a physical *Dimension*. We note that specifying the physical units of a parameter might be overlooked in many analysis models, since the embedded software usually does not keep trace of this information. However, in many applications, and particularly in medical ones, wrong assumptions or confusion about physical units may have fatal consequences. Expressing clearly which units are appropriate for a given value in a given circumstance enables interface developers to formulate verifiable requirements, such as, e.g., "*the concentration must be given in milligrams per liter*".

A device has a *Panel* composed of a set of *Displays* to visualize parameters and of a set of *Settings* to allow users to issue *Commands* to the device. Again, all possible commands for all kinds of devices are grouped in the single *Command* enumeration, complemented by the *changer* predicate specifying if a given command modifies a parameter or mode of operation.

The *Device* class is abstract, as it describes the common characteristics of all devices. An *Infusion_pump* is an example of a concrete class describing one particular device (actually an "infusion pump" is still a very general concept, but for simplicity we'll take it as an example of a concrete class).

The above object-oriented analysis model can be taken as a guide for a logic-based specification. A UML class is the intensional definition of a set of possible instances, i.e., a type. The PVS language has many ways to define a type, including the definition of *uninterpreted* types seen in the previous example. The definition of an uninterpreted type simply declares that a type with a given name exists, with the optional annotation that the type is not empty. Properties of the elements of the type can be expressed with axioms and specific elements can be declared as type members separately from the type definition. The UML *Parameter* class with its associated types can be defined in PVS as a non empty uninterpreted type:

```
parameter_theory: THEORY
     ...
     parameter: TYPE+
     pulse_rate: parameter
     blood_pressure: parameter
     ...
END parameter_theory
```

In the above snippet, the declarations following the *parameter* type introduce *pulse_rate*, *blood_pressure*, and other names, as elements of *parameter*.

A parameter has a value, composed of a numerical magnitude and a physical unit with a corresponding dimension, so the *parameters_th* theory defines the respective types:

```
    dimension: TYPE+
    duration:       dimension
    mass:           dimension
        ...
    unit: TYPE+
    hr:         unit  % hours
    mg:         unit  % milligrams
    ...
    value: TYPE = [# magn: real, units: unit #]
```

Type *value* above is introduced with another form of declaration, the *record* type constructor. In this case, *value* is a record with fields *magn* of type *real* and *units* of type *unit*. Please note that the PVS built-in type *real* represents the mathematical concept of real number, not its approximations used in the programming languages. This means that the PVS environment provides a built-in set of axioms and proved theorems about the real numbers (and of course naturals, integers, rationals ...) that can be referred to in user-defined theories and used to prove theorems.

The *device* type can then be defined as follows:

```
device_theory: THEORY BEGIN
    IMPORTING parameters_th
    command: TYPE+
        incr_cmd:  command
        decr_cmd:  command
        pause_cmd: command
        ...
        changer(c: command): bool
    state:  TYPE = setof[parameter]
    display:  TYPE = setof[parameter]
    commands: TYPE = setof[command]
    panel: TYPE = [# displ: display, cmds: commands #]
    device: TYPE+
    ...
END device_theory
```

The UML *Device* class (Fig. 3) has two attributes, which could be defined as PVS record fields, if *device* were a record type. Another way to map UML attributes to PVS, arguably more flexible, is using functions. The following code shows the signatures of three functions taking an argument of type *device*, which is mapped to an element of type *state*, *mode*, or *panel*, respectively:

```
    state(d: device): state
    operation(d: device): mode
    panel(d: device): panel
```

Similar functions are used in the *parameters_th*, where, in particular, the *readonly* predicate specifies if a parameter cannot be changed by the user:

```
    unit_dimension(u: unit): dimension
    parm_dimension(p: parameter): dimension
    value(p: parameter): value
    readonly(p: parameter):   bool
```

Infusion pumps are a subset of *device* and the main parameters of a pump, settable by the user, are the *volume to be infused* (VTBI) and the infusion *rate*, declared in the *parameters_th* theory. Type *infusion_pump* is declared as a non empty subtype *FROM device*, and its properties are expressed by axioms. For example, there are axioms stating that the commands accepted by the pump change parameter values or the mode of operation:

```
infusion_pumps_theory: THEORY
    IMPORTING device_theory
    infusion_pump: TYPE+ FROM device
    pause_cmd_ax: AXIOM changer(pause_cmd)
    edit_VTBI_ax: AXIOM changer(edit_VTBI)
    edit_rate_ax: AXIOM changer(edit_rate)
    incr_ax:      AXIOM changer(incr)
    ...
end infusion_pumps_theory
```

A specific pump, i.e., an instance if the UML *Infusion_pump* class, can be modeled by a dedicated theory:

```
ACME_pump_theory: THEORY
  IMPORTING infusion_pumps_theory
  ACME_pump: infusion_pump
  ACME_panel: AXIOM
    panel(ACME_pump)`display(VTBI) and
    panel(ACME_pump)`display(rate) and
    panel(ACME_pump)`settings(pause) and
    panel(ACME_pump)`settings(incr) and
  ...
end ACME_pump_theory
```

In the above theory, axiom *ACME_panel* states that parameters *VTBI* and *rate* belong to the *display* part of the pump's panel, and commands *pause*, *incr*, *decr*, *bolus*, and *pwr* belong to the *settings* part. The PVS syntax uses the backtick (‘) notation to access a record field, so that, e.g., *panel(ACME_pump)‘display* is the *display* field of the record associated with *ACME_pump* by the *panel* function. Also, the expression *display(VTBI)* means that *VTBI* belongs to the set *display*: In PVS, the name of a set can be used as a set membership predicate. The declarations in the theory correspond to the displayed data and controls of a real infusion pump shown in Fig. 4. In particular, the settings match buttons in the panel, e.g., *incr* (increment) and *decr* (decrement) match the up and down chevron buttons.

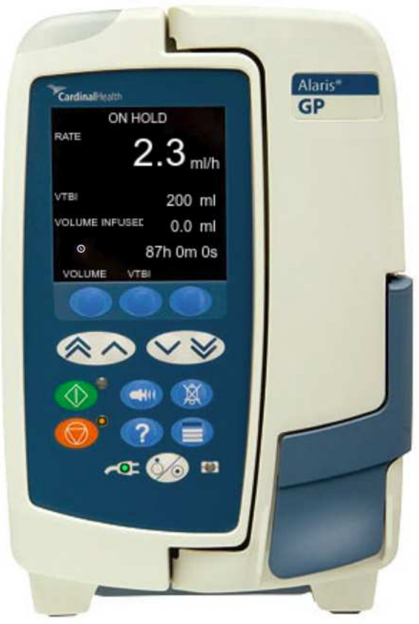

Fig. 4 Front panel of an actual infusion pump.

Another domain concerns the interactions within the ICE, that can be formalized in the following theory, expressing concepts related to commands being issued, confirmed, accepted, enabled, or disabled, concepts related to devices being controlled remotely or locally, and so on:

```
interactions_theory: THEORY BEGIN
    IMPORTING device_theory
    control:  TYPE = {remote, local}
    controlled_under(d: device):     control
    % has cmd_instance i been issued?
    issued(i: cmd_instance):            bool
    % issued locally or remotely?
    issued_under(i: cmd_instance):   control
    enabled(c: command):                bool
    % does c change a parameter or mode?
    changer(c: command):                bool
    confirm_requested(c: command):      bool
    confirmed(c: command):              bool
    accepted(c: command):               bool
    ...
```

```
END interactions_theory
```

Several system-level ICE requirements induce other requirements on the underlying network. Such requirements concern information integrity and availability, and system resilience against malfunctions or improper operation. The basic fact that an ICE is a set of interconnected devices implies that the network must be dependable. Also specific ICE requirements depend on the availability and correctness of the network. For example, data on patient conditions must be available also when the patient is moved to another room. Another important ICE requirement is that the supervisor must be notified of *clinical alarms* related to patient conditions and *technical alarms* related to device or network failures, including device disconnections.

A *communication* theory defines the high-level concepts of communications between devices and supervisor, such as destination device of a command or issue and reception time of a command.

```
communication_theory: THEORY BEGIN
IMPORTING ...
  connected(d: device): bool
  sent_to(i: command, d: device, t: time): bool
  received_by(i: command, d: device, t: time): bool
  ...
END communication_theory
```

4.2 Requirements formalization

In the specification of an ICE, it is necessary to specify requirements on interactions involving devices, people, and supervisor software in various combinations. For example, a simulation showed that an infusion pump could receive a pause command from the ICE while infusion parameters were being manually edited, resulting in over- or underdosing (Masci et al, 2015a); such a situation can be avoided by enforcing a requirement forbidding the intervention of a local operator when a device is being controlled remotely by the ICE supervisor, except for emergency actions. Another requirement, specific to infusion pumps, might state that a confirmation must be requested and granted before a state-changing command issued during an infusion is issued. Other requirements may concern properties of the ICE communication network, such as, e.g., non-duplication of control messages (Bernardeschi et al, 2016). Such requirements refer in particular to the *interactions* theory introduced above.

The requirement that remote control overrides local control, except for the pause command that may be needed in an emergency, can then be expressed as in the following axiom:

```
infusion_pump_reqmts_theory: THEORY BEGIN
  IMPORTING interactions_theory, infusion_pumps_theory
```

```
remote_disables_local: AXIOM
  forall (p: infusion_pump):
    (controlled_under(p) = remote
    => forall (c: command):
        (cmds(pnl(p))(c) and changer(c)
                         and c /= pause_cmd
        => not enabled(c) and enabled(pause_cmd)))
    ...
END infusion_pump_reqmts_theory
```

This axiom means that for any remotely controlled infusion pump *p* all commands in *p*'s settings that change parameter values or operation mode are disabled, except for the *pause* command.

System-level requirements on communication can then be expressed in the following theory:

```
communication_reqmts_theory: THEORY BEGIN
IMPORTING communication_theory
  ...
  cmd_delivery: AXIOM
    forall (i: command, d: device, t: time):
         connected(d) and sent_to(i, d, t)
      => exists (tr: time):
          received_by(i, d, tr) and t < tr
  once: AXIOM
    forall (i: command, d: device, t, t1: time):
         received_by(i, d, t) and received_by(i, d, t1)
      => t1 = t
  disconnect_notification: AXIOM
    forall (d: device):
        not connected(d)
      => disconnect_alarm(d)
END communication_reqmts_theory
```

The first two axioms above concern guarantee of delivery and integrity of communication: *cmd_delivery* states that every command *i* sent to a connected device *d* at time *t* will be received by *d* at a later time t_1, while *once* states that any command *i* received by a device *d* is received only once. Suppose, for example, that the ICE supervisor resets a life-supporting device so that it can be reprogrammed and then restarted. If the data packet carrying the reset command is duplicated and resent by a node, the spurious copy could reach the device after restart and reset it, blocking its life-supporting operation. The *once* axiom forbids this kind of hazard.

The third axiom requires that a disconnection notification to the ICE supervisor related to device *d* be produced when the network controller detects that *d* is disconnected.

5 Formalization of the ICE Communication Network

As briefly anticipated in the Introduction, an ICE communication network carries data and control signals between devices (including operator interfaces) and the ICE supervisor. The network may also be interfaced to external networks. For example, the supervisor might fetch a patient's electronic medical record from a database, update it, and return it to the database.

The network may be structured on a number of subnetworks, each of which could rely on multiple physical infrastructures, possibly shared among subnetworks. For example, the communication network could use small wired networks for the operating rooms and a larger, both wired and wireless network for the ward. The operating room subnetworks would support medical apps with special safety and dependability requirements.

Such a complex system must be modeled at different levels of abstraction, down to the level of network topology and communication protocols. The rest of this section sketches a network specification that is general enough to allow for many different choices of hardware and communication protocols.

5.1 Network structure

The supervisor and the devices can be modeled abstractly as network *nodes*. The networking infrastructure may have routing elements, also modeled as nodes. Each node is identified by a natural number (of type *node_id*) smaller than the number *network_size* of nodes. The mapping from devices to their network identifiers is expressed by a function *dev2node_f*. Nodes with special roles, such as the network controller (associated with the supervisor), are identified by constants of type *node_id*. Further, two subsets of *node_id* (*router_id* and *device_id*) can be used to identify router and device nodes, as shown in the *nodes* theory.

```
nodes_theory: THEORY BEGIN
  IMPORTING devices_theory
  network_size: posnat
  node_id: TYPE = below(network_size)
  router_id: TYPE FROM node_id
  device_id: TYPE FROM node_id
  network_controller: node_id
  dev2node_f: TYPE = [device -> node_id]
  ...
END nodes_theory
```

Graphs are an obvious tool to represent the communication network. In PVS, we can use the *digraphs* theory provided by the NASA PVS libraries (Butler and Sjogren, 1998), which is parametric with respect to the type of graph nodes. In the *network_graph* theory below, type *network_graph* is the set of directed graphs *g* such that each node *n* is in the set *vert(g)* of vertices and each pair (m,n) of nodes is in the set *edges(g)* of edges only if the two nodes are distinct, i.e., the graph has no self-edge. The theory also defines the type *topology* of functions from node identifiers to finite sets of node identifiers, meant to represent the set of immediate neighbors of each node.

```
network_graph_theory: THEORY BEGIN
  IMPORTING nodes_theory, digraphs[node_ids]
  network_graph: TYPE =
    {g: digraph[node_ids] |
      (FORALL (n: node_ids):
        vert(g)(n))
        and (forall (m, n: node_ids):
          edges(g)((m, n)) => (m /= n))}
  topology: TYPE = [node_ids -> finite_set[node_ids]]
  ...
 END network_graph_theory
```

5.2 Network dynamics

The kinds of information flowing through the network range from very simple messages, such as "*start infusion*" to highly structured data, such as DICOM images. The definition of such application-specific data must be given in separate theories, while the general model of the communication network provides a low-level, application-agnostic specification of packet communication. The *packet* theory defines packets as records with fields for timestamp, originating (*source*) node, destination nodes, and payload. The theory also defines the type of packet *trains* (*pktrain* in the PVS code), i.e., groups of packets sent in a single burst from a *sender* node to a common set of *receiver* nodes.

```
packet_theory: THEORY BEGIN
IMPORTING nodes_theory, time_theory
    address: TYPE = finite_set[node_id]
    ...
    packet: TYPE = [#
        timestamp: time,
        source_addr: node_id,
        destination_addr: finite_set[node_id],
        payload: finite_sequence[int] #]

    pktrain: TYPE = [#
        pks:
          {pks: finite_set[packet] | NOT empty?(pks)},
        time_tx: time,
        sender_addr: node_id,
        receivers_addr: {rcv: finite_set[node_id] | NOT
            member(sender_addr, rcv)} #];
END packet_theory
```

Each node stores incoming packet trains in a *receive buffer*. At any time, the state of the network is defined by the contents of each node's buffer and, in a scenario of mobile nodes, by the current topology and the current physical locations of the nodes. The *network* theory defines the network state as a record with fields for a global clock, functions mapping each node to its receive buffer and location, and a log recording the sequence of packets processed by each node (the definitions of buffers and locations are omitted). Communication primitives, such as *forward*, handle packets and update the network state accordingly.

```
network_theory: THEORY BEGIN
  IMPORTING time_theory, receive_buffer_theory,
    location_theory
  network_state: TYPE = [#
    global_clock: time,
    net_rcv_buf: [node_id -> rcv_buf],
    net_location: [node_id -> location]
    log: [node_id -> finite_sequence[packet]] #]

  forward(p: packet)
    (forwarder: node_id)
    (net: network_state, g: network_graph):
        network_state = ...
  ...
END network_theory
```

Using the above theories, different messaging protocols can be modeled, according to the needs of different applications. A network protocol is an algorithm executed by each node in order to perform a network service, primarily to propagate application-specific information. The most general description of this concept is *an algorithm that updates the state of a network*, i.e., a function from network states to network states. Since such a function in general depends on the network structure, the *protocol* type can be specified as in the following theory:

```
protocol_theory: THEORY BEGIN
  IMPORTING network_graph_theory, network_theory
  protocol: TYPE =
    [network_graph, node_id
        -> [network_state -> network_state]]
END protocol_th
```

Most protocols need a routing table to store, for each node, paths towards other nodes. In the *routing_table* theory, the type of routing tables is defined by the set of functions mapping each source node *i* to a vector of paths leading to the other nodes. The *digraphs* theory defines a *path* as a nonempty finite sequence of nodes connected by edges, and a vector of paths originating from node *i* is in turn a function mapping each correspondent node *j* to the path from the source node. Predicate *routing_tbl?* means that a function *rt* of type *routing_table* is a routing table of a network graph *g* if *rt* maps any ordered pair *i*, *j* of nodes to a path from *i* to *j*. Predicate *valid_routing_tbl?* imposes the further condition that the route does not contain any self-edge.

```
routing_table_theory: THEORY
BEGIN
IMPORTING network_graph_theory, digraphs[node_id]
  routing_tbl: TYPE =
    [i: node_id -> [j: node_id -> path[node_id]]]

  routing_tbl?(rt: routing_tbl, g: network_graph):
        bool =
    FORALL (i, j: node_id):
      path_from?(g, rt(i)(j), i, j)

  valid_route?(g: network_graph,
               p: path[node_id],
               i, j: node_id): bool =
    ((i /= j) AND (l(p) > 1)
      AND path_from?(g, p, i, j))

  valid_routing_tbl?(rt: routing_tbl,
                     g: network_graph): bool =
    routing_tbl?(rt,g)
    AND FORALL (i, j: node_id):
      valid_route?(g, rt(i)(j), i, j)
   ...
 END routing_table_theory
```

5.2.1 Mobility

In order to express node mobility, the network model can be extended with a theory where mobility is expressed with functions that change network connectivity in three steps: i) select a target direction among those allowed by the topology, ii) determine the new set of neighbors of the mobile node, iii) modify and return the new topology. The *node_mobility* theory defines such functions; three auxiliary functions are used to implement the corresponding steps.

```
node_mobility_theory: THEORY
BEGIN
```

```
IMPORTING network_graph_theory
  %-- select a target direction
  select_target(s: finite_set[node_id]): node_id

  %-- set of neighbours for the mobile node
  new_neighbours(tp: topology,
                 mobile_node, target_node: node_id):
        finite_set[node_id] =
    {n: node_id | n /= mobile_node
              AND (tp(target_node)(n) OR n = target_node)}

  %-- change topology tp according to
  %-- the new neighbourhood
  change_topology(tp: topology)
                 (mobile_node:
                  node_id, nbs: finite_set[node_id]):
          topology =
    LET tp = remove_node(mobile_node, tp)
    IN add_node(mobile_node, nbs, tp)

  %-- node mobility function
  node_mobility(m: node_id, tp: topology): topology =
    LET target = select_target(tp(m)),
        new_nbs = new_neighbours(tp, m, target)
    IN change_topology(tp)(m, new_nbs)
  ...
END node_mobility_theory
```

5.3 Requirements

The requirements of the communication network derive from the higher-level ICE requirements, i.e., they express the properties that any network implementation must exhibit in order to be used in an ICE. Consider, for example, the *once* axiom in Theory *communication_reqmts*. In terms of network-specific concepts, the absence of packet duplication can be expressed as "*in any network state, for all packets p and node n, the set of packet trains containing p transmitted by n is either empty or a singleton*", as shown in the following excerpt, where *transm_tpkts* is the set of packet trains containing *p* and sent by a node *n* (definition not shown):

```
comm_netwk_reqmts_theory: THEORY BEGIN
  IMPORTING ...
  no_duplication: AXIOM
    FORALL (net: network_state, p: packet):
      FORALL (n: node_id):
        empty?(transm_tpkts(p, log(net), n)) OR
        singleton?(transm_tpkts(p, log(net), n))
  ...
END comm_netwk_reqmts_theory
```

Other types of requirements concern the interaction between devices and supervisor at a higher abstraction level. For example, in order to express the *disconnect_notification* system requirement (Sec. 4) as a network requirement, the following declarations are included in the *network* theory:

```
  alarm_cause: TYPE = {disconnection, ...}
  severity_t: TYPE = {low, medium, high}
  disconnected(d: node_id): bool
  alarm(d: node_id, c: alarm_cause, s:severity): bool
  severity(d: node_id, c: alarm_cause): severity_t
```

The *alarm* function is true if device d is in the condition described by c, with severity level s. The latter is obtained by function *severity*, whose value depends both on the affected device and on the cause of the alarm.

The following function, from the *network* theory, checks if node n is disconnected, by analyzing the network graph in the current state. The actual definition of the function will be specified by axioms.

```
  node_disconn(s: network_state,
               g: network_graph, n: node_id): bool
```

The following axioms from the *comm_netwk_reqmts* theory specify the above stated requirement:

```
  dev_disconn: AXIOM
    FORALL (d: device):
      FORALL (s: network_state):
        FORALL (g: network_graph):
         node_disconn(s, g, dev2node(d))
         => disconnected(d)

  disconn_alarm: AXIOM
    FORALL (d: device):
    LET n = dev2node(d)
    IN disconnected(d)
       => alarm(n, disconnection,
                severity(n, disconnection))
```

The above discussion shows that PVS language is well suited to specifying such a complex system as an ICE. The modular composability of PVS theories and the flexibility of the type system make it possible to structure the overall specification in a set of interrelated theories, each devoted to a specific (sub)domain or level of abstraction. Such a specification can be analyzed for consistency using verification tools whenever an update is necessary, e.g., in case of changes of regulations or introduction of new equipment or therapies.

6 Verification

This chapter is focused on the formal specification of requirements for integrated clinical environments. To integrate this discussion, this section hints at the application of formal specification as a basis for verification.

An advantage of the approach used in this work is its ability to describe a system at different levels of abstraction. A number of different versions of the theories can be developed for each component, each one at a different level of detail. The most abstract theories provide the declarations of the basic set of interface functions (i.e., functions meant to be used in other theories) and types. More detailed theories can be derived from the abstract definitions by specifying the definition of the functions and by extending types. If different versions of a theory provide the same declarations for interface functions and types, they are interchangeable, hence, when building the model, the minimal set of details needed for analysis can be used, by importing the appropriate version of the theory.

For example, consider the high-level definition of the *protocol* type in Section 5.3 above. An instance of that type is a function defining the sequence of actions performed by a generic node. Actions may depend on the content of received packets (e.g., the sender address of a received packet) and on the state of the node (e.g., the value of patient physiological data).

Verification is accomplished by producing a formal model of the implementation to be verified. Then a verification theory can be built, where the *axioms* from the requirements theories are expressed as *theorems* on the implementation model, as shown in the following schema, where the *no_loop* requirement is taken as an example:

```
implementation_theory: THEORY BEGIN
IMPORTING protocol_theory, ...
  init_state: network_state = (# ... #)
  transm_pkts(p: packet,
      l: [node_id -> finite_sequence[packet]],
      n: node_id): bool =
    % a predicate depending on
    % the protocol
   ...
END implementation_theory

verification_theory: THEORY BEGIN
IMPORTING implementation_theory ...
  ...
    no_loop_property(ns, pk_star): bool =
      FORALL (i: sensor_id):
        LET tp = transm_pkts(pk_star)(net_log(ns), i)
        IN empty?(tp) OR singleton?(tp)
  ...
  no_loop_thm: THEOREM
      FORALL(n: nat):
        no_loop_property(t(n)`state, pk_star)
  ...
END verification_theory
```

The implementation theory contains assumptions on the implemented network, including structural and behavioral properties, including the definition of, or assumption on, the initial state, and how a packet is transmitted through a given protocol. In the verification theory, it is then possible to prove that the chosen protocol satisfies the above requirement.

Figure 5 shows a simplified view of the dependencies among theories related to an ICE communication network.

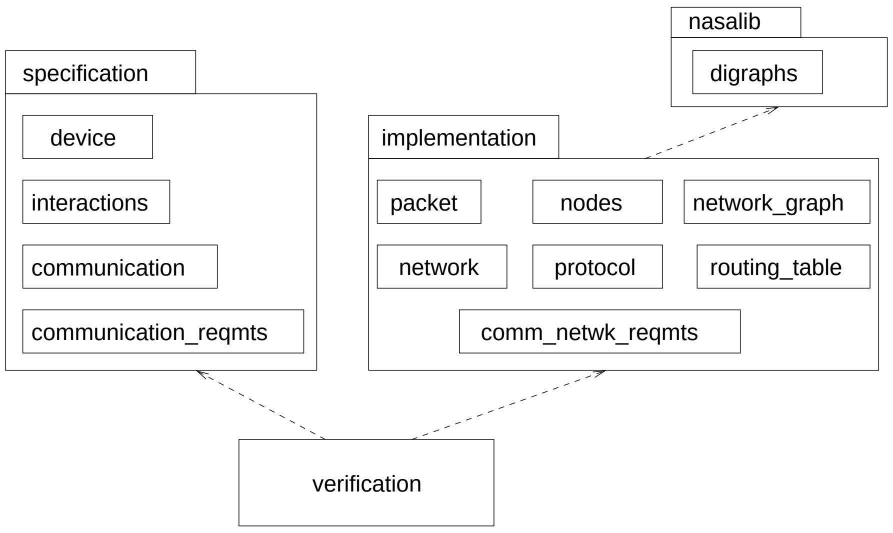

Fig. 5 Dependencies among theories.

6.1 Verification of the Surge protocol

As an example of a concrete protocol verification, let us consider the *Surge* protocol (Levis et al, 2003a) often used in mobile wireless networks. This protocol routes packets along the branches of a spanning tree embedded in the network topology and rooted in a base station. In the Surge protocol, the spanning tree can change dynamically to accommodate for changes of topology caused by node mobility. Dynamic routing could create loops in the path of some packets. Such loops could pose safety threats in an ICE where the network must deliver alarms or control commands. *Surge* identifies routing loops by inspecting the source address of packets, and suppresses routing loops by dropping packets that revisit their origin (Levis et al, 2003b). This design decision is simplistic and is due to the fact that WSNs packets cannot contain a field that reports a list of all traversed nodes (control information included in the packets must be as limited as possible to keep the packet size small in order to save energy). Each node should avoid to forward packets already transmitted, including those originated at different nodes. In this verification example, a loop-free version (*SurgeNL*) of the original protocol is confirmed to be actually loop-free. The reader should be aware that this protocol, however, does not guarantee packet delivery and has been chosen only as a conveniently simple proof of concept for protocol verification.

In the *SurgeNL* protocol, each node alternates between reception and transmission phases whose durations are left unspecified in the most general network model (they would be specified by a scheduler theory for each given application). In the transmission phase, a node *x* executes a protocol step: (i) if the node's receive buffer is empty, the node idles, i.e., returns to the reception phase; otherwise, (ii) it examines the packets in the receive buffer; (iii) if any of them originates from *x*, all received packets are dropped; otherwise, (iv) the received packets are forwarded along with a new packet to be used in further steps for loop detection.

The protocol step is defined by the following function:

```
surgeNL(x: sensor_id)
   (net: network_state, g: network_graph)
   (rt:
     {rt: routing_table | valid_routing_table?(rt, g)}):
                                     network_state =
  LET received_pks = net_receive_buffer(net)(x),
      next_hop = next_hop(x, network_controller)(g, rt)
  IN
    IF empty?(received_pks)
    THEN idle(x)(net, g, rt)
    ELSE
      IF EXISTS (pk: {pk: packet | received_pks(pk)}):
             source_addr(pk) = x
      THEN
        drop(received_pks, next_hop) (x)(net, g, rt)
      ELSE
        LET t = net_time(net)(x),
            originated_pk = new_packet(t, x)
        IN inject_and_forward(originated_pk,
                              received_pks, next_hop)
                             (x)(net, g, rt)
      ENDIF
    ENDIF
```

Functions *idle*, *drop*, and *inject_and_forward* are low-level single-hop communication primitives from the *network* theory. Function *next_hop* is declared in a *routing_table* theory (not shown).

The theorem can be proved with the strategy of *configuration diagrams* (Rushby, 2000), based on the construction of intermediate lemmas of the form *configuration* ⇒ *invariant*, where *configuration* is a predicate defining a particular state of a system and *invariant* is a property to be proved. In our case, the invariant is *no_loop_property* and the configurations are *base*, *injected*, and *preLoop*, representing the situations where, respectively, a packet *pk** has not yet been injected in the network by a scheduled node (*base*), the packet has been injected and it is not in the receive buffer of nodes already visited by the packet (*injected*), and the injected packet is in the receive buffer of a node that had already transmitted the packet (*preLoop*). The configuration diagram is shown in Figure 6.

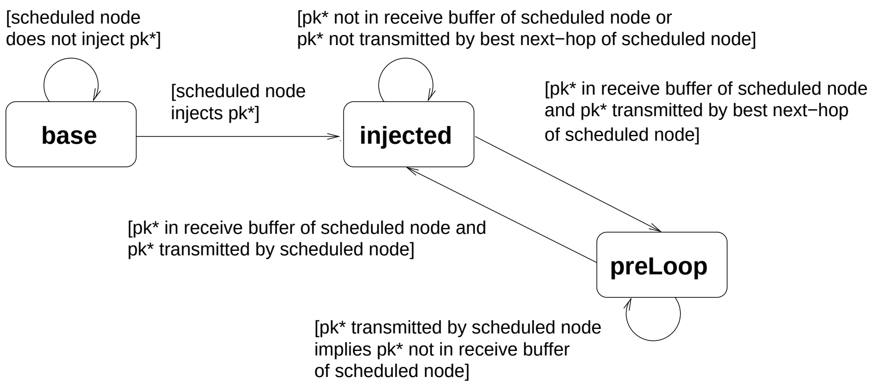
[scheduled node
does not inject pk*]
[pk* not in receive buffer of scheduled node or
pk* not transmitted by best next−hop of scheduled node]
[scheduled node
injects pk*]
base
injected
[pk* in receive buffer of scheduled node
and pk* transmitted by best next−hop
of scheduled node]
[pk* in receive buffer of scheduled node and
pk* transmitted by scheduled node]
preLoop
[pk* transmitted by scheduled node
implies pk* not in receive buffer
of scheduled node]

Fig. 6 Configuration diagram used to prove *noLoop_property*.

Lemma *base_implies_noLoop* and invariant *noLoop_property* below are shown as an example of the intermediate lemmas.

```
  base_implies_noLoop : LEMMA
    base(ns, pk) => noLoop_property(ns, pk)
```

where

```
noLoop_property(ns, pk): bool =
 FORALL (i: sensor_id):
  LET tp = transmitted(pk)(net_log(ns), i)
   IN empty?(tp)        % packet not yet transmitted
      OR singleton?(tp) % or a single copy exists
                        % in the network
```

After proving the above lemmas and invariants, the proof of the main theorem reduces to performing simple sequent transformations, as shown in the following subsection.

6.2 Interactive proof

The proof is carried out under the hypothesis of reliable single-hop communication, i.e., if a node *i* transmits a packet *pk* to a node *j* in communication range, then *j* will eventually receive packet *pk*.

The PVS interface presents the theorem to be proved as a sequent without antecedents, that the user simplifies by instantiating the universal quantifier with arbitrary constants, using the *skosimp** (Skolemize and simplify) rule:

```
noLoop_property :

  |-------
{1}   FORALL (pk_star: packet,
          t: Trace[State, init_states, surge_transition],
          n: nat):
        noLoop_property(t(n)`state, pk_star)

Rule?: (skosimp*)
noLoop_property :

  |-------
{1}   noLoop_property(t!1(n!1)`state, pk_star!1)
```

Then, rule *use* introduces lemma *config_diagram_closed* as an antecedent, finds appropriate instantiations for the lemma and produces the new sequent. The previously proved lemma *config_diagram_closed* ensures that for each configuration the disjunction of the transition conditions covers all possible cases.

```
Rule?: (use "config_diagram_closed")
noLoop_property :

{-1}  LET ns = t!1(n!1)`state IN
        base(ns, pk_star!1) OR
         injected(ns, pk_star!1) OR
          preLoop(ns, pk_star!1)
  |-------
[1]   noLoop_property(t!1(n!1)`state, pk_star!1)
```

Rule *beta* replaces *ns* with its definition provided by the *LET* clause:

```
Rule?: (beta *)
```

```
noLoop_property :

{-1}  base(t!1(n!1)`state, pk_star!1) OR
       injected(t!1(n!1)`state, pk_star!1) OR
        preLoop(t!1(n!1)`state, pk_star!1)
  |-------
[1]   noLoop_property(t!1(n!1)`state, pk_star!1)
```

Rule *prop* splits the disjunction and produces three subgoals, where the first one is *always_noLoop_property.1*:

```
Rule?: (prop)
noLoop_property.1 :

{-1}  base(t!1(n!1)`state, pk_star!1)
  |-------
[1]   noLoop_property(t!1(n!1)`state, pk_star!1)
```

This subgoal and the other two are solved by invoking a lemma and simplifying with the *ground* rule, which applies automatically a series of simplifications:

```
Rule?: (use "base_implies_noLoop")
noLoop_property.1 :

{-1}  base(t!1(n!1)`state, pk_star!1) =>
       noLoop_property(t!1(n!1)`state, pk_star!1)
[-2]  base(t!1(n!1)`state, pk_star!1)
  |-------
[1]   noLoop_property(t!1(n!1)`state, pk_star!1)

Rule?: (ground)

This completes the proof of noLoop_property.1.

...

This completes the proof of noLoop_property.3.

Q.E.D.
```

Figure 7 shows the corresponding proof tree.

7 Conclusions

In a typical clinical setting, several disparate devices are used both for monitoring patient conditions and for administering therapy. Each device normally operates

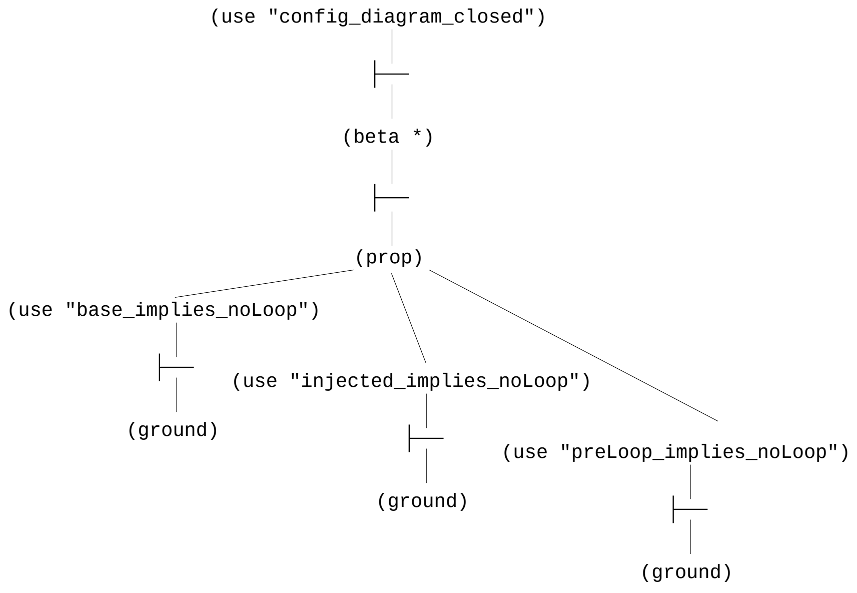
(use "config_diagram_closed")
⊢
(beta *)
⊢
(prop)
(use "base_implies_noLoop")
⊢
(ground)
(use "injected_implies_noLoop")
⊢
(ground)
(use "preLoop_implies_noLoop")
⊢
(ground)

Fig. 7 Proof tree for *noLoop_property*.

independently of other devices and offers its own interface to patients and clinicians. Smart integrated clinical environments aim at improving patient care by coordinating and supervising medical devices and providing support to clinicians. The development of these environments poses many challenges, as they must face the complexity of socio-technical systems and satisfy strict safety requirements. In particular, rigorous requirements specification is an essential basis for development.

In this chapter, an approach to the formalization of system requirements for integrated clinical environments is proposed. The fundamental feature of this approach is the use of a higher-order logic language, provided by the PVS theorem-proving environment.

The approach has been illustrated by providing and discussing short excerpts of logical theories describing concepts of, and requirements on, different aspects of communication networks for clinical environments, at different abstraction levels. The examples are meant to support the thesis that logic-based formal specification is a useful tool in the development of complex, safety-critical systems, including integrated clinical environments, as it enables developers to produce modular, detailed, and flexible specifications, which can then be used for verification and validation activities necessary to gain confidence that a medical system complies with safety requirements.

References

AAMI MDI/2012-03-30 (2012) Medical device interoperability. Association for the Advancement of Medical Instrumentation

Ahmed HS, Ali AA (2016) Smart intensive care unit design based on wireless sensor network and internet of things. In: 2016 Al-Sadeq International Conference on Multidisciplinary in IT and Communication Science and Applications (AIC-MITCSA), pp 1–6, DOI 10.1109/AIC-MITCSA.2016.7759905

ANSI/HL7 V2.8.2-2015 (2015) Health level seven standard version 2.8.2 — an application protocol for electronic data exchange in healthcare environments. Health Level Seven International

Arney D, Goldman JM, Bhargav-Spantzel A, Basu A, Taborn M, Pappas G, Robkin M (2012) Simulation of medical device network performance and requirements for an integrated clinical environment. Biomedical Instrumentation & Technology 46(4):308–315, DOI 10.2345/0899-8205-46.4.308

Arney D, Plourde J, Schrenker R, Mattegunta P, Whitehead SF, Goldman JM (2014) Design Pillars for Medical Cyber-Physical System Middleware. In: Turau V, Kwiatkowska M, Mangharam R, Weyer C (eds) 5th Workshop on Medical Cyber-Physical Systems, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, OpenAccess Series in Informatics (OASIcs), vol 36, pp 124–132, DOI 10.4230/OASIcs.MCPS.2014.124, URL `http://drops.dagstuhl.de/opus/volltexte/2014/4529`

Arney D, Plourde J, Goldman JM (2017) OpenICE medical device interoperability platform overview and requirement analysis. Biomedizinische Technik Biomedical Engineering DOI 10.1515/bmt-2017-0040

Avižienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1:11–33, DOI doi.ieeecomputersociety.org/10.1109/TDSC.2004.2

Bernardeschi C, Domenici A (2016) Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System. Information Processing Letters 116(6):409–415, DOI 10.1016/j.ipl.2016.02.001

Bernardeschi C, Domenici A, Masci P (2014) Integrated simulation of implantable cardiac pacemaker software and heart models. In: CARDIOTECHNIX 2014, 2d International Congress on Cardiovascular Technology, SCITEPRESS, pp 55–59

Bernardeschi C, Domenici A, Masci P (2015) Towards a formalization of system requirements for an integrated clinical environment. In: 5th EAI International Conference on Wireless Mobile Communication and Healthcare – "Transforming healthcare through innovations in mobile and wireless technologies", ACM, DOI 10.4108/eai.14-10-2015.2261701

Bernardeschi C, Domenici A, Masci P (2016) Modeling communication network requirements for an integrated clinical environment in the Prototype Verification System. In: 2016 IEEE Symposium on Computers and Communication (ISCC), IEEE, pp 135–140, DOI 10.1109/ISCC.2016.7543728

Bernardeschi C, Domenici A, Masci P (2017) A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. IEEE Transactions on Software Engineering PP(99):1–1, DOI 10.1109/TSE.2017.2694423

Butler R, Sjogren J (1998) A PVS Graph Theory Library. Nasa technical memorandum 1998-206923, NASA Langley Research Center, Hampton, Virginia

Corsaro A, Schmidt DC (2012) The data distribution service – the communication middleware fabric for scalable and extensible systems-of-systems, INTECH. DOI 10.5772/30322

Domenici A, Fagiolini A, Palmieri M (2017) Integrated simulation and formal verification of a simple autonomous vehicle. In: 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems, Springer, in press

F2761-2009 (2009) Medical Devices and Medical Systems — Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) — Part 1: General requirements and conceptual model. ASTM International

FDA Guidance (2009) Design Considerations and Pre-market Submission Representations for Interoperable Medical Devices — Guidance for Industry and Food and Drug Administration Staff. US Food and Drug Administration

García-Valls M, Touahria IE (2017) On line service composition in the integrated clinical environment for ehealth and medical systems. Sensors 17(6), DOI 10.3390/s17061333

García-Valls M, Lopez IR, Villar LF (2013) iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. IEEE Transactions on Industrial Informatics 9(1):228–236, DOI 10.1109/TII.2012.2198662

Girard JY, Lafont Y, Taylor P (1990) Proofs and Types, Cambridge Tracts in Theoretical Computer Science, vol 7. Cambridge University Press, URL `http://www.paultaylor.eu/stable/Proofs+Types.html`

Halpern NA (2014) Advanced informatics in the intensive care unit: Possibilities and challenges. URL `http://healthcare.nist.gov/medicaldevices/publications.html`

Harrison MD, Masci P, Campos JC, Curzon P (2014) Demonstrating that medical devices satisfy user related safety requirements. In: 4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014)

Islam SMR, Kwak D, Kabir MH, Hossain M, Kwak KS (2015) The internet of things for health care: A comprehensive survey. IEEE Access 3:678–708, DOI 10.1109/ACCESS.2015.2437951

Kabachinski J (2006) What is Health Level 7? Biomedical Instrumentation & Technology 40(5):375–379, DOI 10.2345/i0899-8205-40-5-375.1, URL `http://dx.doi.org/10.2345/i0899-8205-40-5-375.1`, `http://dx.doi.org/10.2345/i0899-8205-40-5-375.1`

Kühn F, Leucker M (2014) OR.NET: Safe Interconnection of Medical Devices, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 188–198. DOI 10.1007/978-3-642-53956-5_13

Larson B, Hatcliff J, Procter S, Chalin P (2012) Requirements specification for apps in medical application platforms. In: Proceedings of the 4th International Workshop on Software Engineering in Health Care, IEEE Press, Piscataway, NJ, USA, SEHC '12, pp 26–32

Larson BR, Hatcliff J (2014) Open Patient-Controlled Analgesia Infusion Pump System Requirements — DRAFT 0.11. Tech. Rep. SAnToS TR 2014-6-1, Kansas State University

Larson BR, Hatcliff J, Chalin P (2013) Open source patient-controlled analgesic pump requirements documentation. In: Proceedings of the 5th International Workshop on Software Engineering in Health Care, IEEE Press, Piscataway, NJ, USA, SEHC '13, pp 28–34

Leite FL, Adler R, Feth P (2017) Safety Assurance for Autonomous and Collaborative Medical Cyber-Physical Systems, Springer International Publishing, Cham, pp 237–248. DOI 10.1007/978-3-319-66284-8_20

Levis P, Lee N, Welsh M, Culler D (2003a) TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, ACM, New York, NY, USA, SenSys '03, pp 126–137, DOI 10.1145/958491.958506

Levis P, Lee N, Welsh M, Culler D (2003b) TOSSim: accurate and scalable simulation of entire TinyOS applications. In: Proc. Intl. Conf. on Embedded Networked Sensor Systems, ACM Press, pp 126–137, DOI http://doi.acm.org/10.1145/958491.958506

Masci P, Zhang Y, Jones P, Curzon P, Thimbleby H (2014) Formal verification of medical device user interfaces using PVS. In: Gnesi S, Rensink A (eds) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 8411, Springer Berlin Heidelberg, pp 200–214, DOI 10.1007/978-3-642-54804-8_14

Masci P, Mallozzi P, De Angelis FL, Di Marzo Serugendo G, Curzon P (2015a) Using PVSio-web and SAPERE for rapid prototyping of user interfaces in Integrated Clinical Environments. In: Verisure2015, Workshop on Verification and Assurance, co-located with CAV2015

Masci P, Oladimeji P, Mallozzi P, Curzon P, Thimbleby H (2015b) PVSio-web: Mathematically Based Tool Support for the Design of Interactive and Interoperable Medical Systems. In: Proceedings of the 5th EAI International Conference on Wireless Mobile Communication and Healthcare, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, MOBIHEALTH'15, pp 42–45, DOI 10.4108/eai.14-10-2015.2261720

Mauro G, Thimbleby H, Domenici A, Bernardeschi C (2017) Extending a user interface prototyping tool with automatic MISRA C code generation. In: Dubois C, Masci P, Méry D (eds) Proceedings of the Third Workshop on Formal Integrated Development Environment, Limassol, Cyprus, November 8, 2016, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol 240, pp 53–66, DOI 10.4204/EPTCS.240.4

Muñoz C (2003) Rapid prototyping in PVS. Tech. Rep. NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA

Owre S, Rushby J, Shankar N (1992) PVS: A prototype verification system. In: Kapur D (ed) Automated Deduction — CADE-11, Lecture Notes in Computer Science, vol 607, Springer Berlin Heidelberg, pp 748–752, DOI 10.1007/3-540-55602-8_217

Rausch TL, Judd TM (2016) Using integrated clinical environment data for health technology management. In: 2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI), pp 607–609, DOI 10.1109/BHI.2016.7455971

Ray A, Jetley R, Jones PL, Zhang Y (2010) Model-based engineering for medical-device software. Biomedical Instrumentation & Technology 44(6):507–518, DOI 10.2345/0899-8205-44.6.507

Rhoads JG, Cooper T, Fuchs K, Schluter P, Zambuto RP (2010) Medical Device Interoperability and the Integrating the Healthcare Enterprise (IHE) Initiative. Biomedical instrumentation & technology suppl.:21–27

Rushby J (2000) Verification diagrams revisited: Disjunctive invariants for easy verification. In: Emerson EA, Sistla AP (eds) Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 508–520, DOI 10.1007/10722167_38

Venkatasubramanian KK, Vasserman EY, Sfyrla V, Sokolsky O, Lee I (2015) Requirement Engineering for Functional Alarm System for Interoperable Medical Devices, Springer International Publishing, Cham, pp 252–266. DOI 10.1007/978-3-319-24255-2_19