1-1-2016

Foundations for Safety-Critical on-Demand Medical Systems

Andrew Lewis King
University of Pennsylvania, kingand@cis.upenn.edu

Foundations for Safety-Critical on-Demand Medical Systems

Abstract
In current medical practice, therapy is delivered in critical care environments (e.g., the ICU) by clinicians who manually coordinate sets of medical devices: The clinicians will monitor patient vital signs and then reconfigure devices (e.g., infusion pumps) as is needed. Unfortunately, the current state of practice is both burdensome on clinicians and error prone.

Recently, clinicians have been speculating whether medical devices supporting ``plug & play interoperability'' would make it easier to automate current medical workflows and thereby reduce medical errors, reduce costs, and reduce the burden on overworked clinicians. This type of plug & play interoperability would allow clinicians to attach devices to a local network and then run software applications to create a new medical system ``on-demand'' which automates clinical workflows by automatically coordinating those devices via the network.

Plug & play devices would let the clinicians build new medical systems compositionally. Unfortunately, safety is not considered a compositional property in general. For example, two independently ``safe'' devices may interact in unsafe ways. Indeed, even the definition of ``safe'' may differ between two device types.

In this dissertation we propose a framework and define some conditions that permit reasoning about the safety of plug & play medical systems. The framework includes a logical formalism that permits formal reasoning about the safety of many device combinations at once, as well as a platform that actively prevents unintended timing interactions between devices or applications via a shared resource such as a network or CPU. We describe the various pieces of the framework, report some experimental results, and show how the pieces work together to enable the safety assessment of plug & play medical systems via a two case-studies.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Insup Lee

Keywords
Medical Systems, On-Demand, Plug and Play, Real-Time Systems, Safety Critical Systems

Subject Categories
Computer Sciences

FOUNDATIONS FOR SAFETY-CRITICAL ON-DEMAND MEDICAL SYSTEMS

Andrew L. King

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2016

Supervisor of Dissertation

_____

Insup Lee

Cecilia Fitler Moore Professor, Computer and Information Science

Graduate Group Chairperson

_____

Lyle Ungar, Professor, Computer and Information Science

Dissertation Committee:

Oleg Sokolsky, Research Prof., Computer and Information Science, University of Pennsylvania

Rajeev Alur, Prof., Computer and Information Science, University of Pennsylvania

Rahul Mangharam, Prof., Computer and Information Science, University of Pennsylvania

John Hatcliff, Prof., Computing and Information Science, Kansas State University

FOUNDATIONS FOR SAFETY-CRITICAL ON-DEMAND MEDICAL SYSTEMS

ABSTRACT

FOUNDATIONS FOR SAFETY-CRITICAL ON-DEMAND MEDICAL SYSTEMS

Andrew L. King

Insup Lee

In current medical practice, therapy is delivered in critical care environments (*e.g.,* the ICU) by clinicians who manually coordinate sets of medical devices: The clinicians will monitor patient vital signs and then reconfigure devices (*e.g.,* infusion pumps) as is needed. Unfortunately, the current state of practice is both burdensome on clinicians and error prone.

Recently, clinicians have been speculating whether medical devices supporting "plug & play interoperability" would make it easier to automate current medical workflows and thereby reduce medical errors, reduce costs, and reduce the burden on overworked clinicians. This type of plug & play interoperability would allow clinicians to attach devices to a local network and then run software applications to create a new medical system "on-demand" which automates clinical workflows by automatically coordinating those devices via the network.

Plug & play devices would let the clinicians build new medical systems compositionally. Unfortunately, safety is not considered a compositional property in general. For example, two independently "safe" devices may interact in unsafe ways. Indeed, even the definition of "safe" may differ between two device types.

In this dissertation we propose a framework and define some conditions that permit reasoning about the safety of plug & play medical systems. The framework includes a logical formalism that permits formal reasoning about the safety of many device combinations at once, as well as a platform that actively prevents un-

intended timing interactions between devices or applications via a shared resource such as a network or CPU. We describe the various pieces of the framework, report some experimental results, and show how the pieces work together to enable the safety assessment of plug & play medical systems via a two case-studies.

Contents

List of Tables

List of Illustrations

Chapter 1

Introduction

1.1 Motivation

Traditionally, safety-critical systems have been designed and integrated as monolithic units before they are delivered to the customer. Typically, a prime contractor manages development of the system from design through final systems integration. Because the prime contractor manages the entire development process, they are in a unique position to assess the completed product for safety: They know what components comprise the system, how those components interact (*e.g.,* as verified via integration testing), the intended use of the system and the system-level safety requirements. Very often in regulated domains, such as aviation and medical systems, the prime contractor must also construct a regulatory submission that contains an argument (and supporting evidence) that their system behaves safely and effectively. The system in question cannot be marketed or otherwise deployed until the regulator accepts the argument and approves and/or certifies the system.

In medicine, clinicians currently deliver therapy by manually coordinating collections of independently developed devices. Now that many devices marketed today already include some form of network connectivity (serial ports, Ethernet, 802.11 or Bluetooth wireless) clinicians are recognizing the potential to automate device coordination via external control applications [70]. Ideally, future medical devices would support plug & play protocols which would allow clinicians to construct networks of medical devices *on-demand* that automatically interoperate to automate life-critical clinical workflows [69]. The integration model for on-demand systems would differ from traditional systems because they would not be supplied or integrated by a single vendor. Instead, a Health Delivery Organization (HDO) would purchase interoperable devices, infrastructure (*i.e.,* computational platforms) and software applications implementing clinical algorithms (*i.e.,* "apps") from a variety of different vendors. Specific medical systems would then be assembled from devices on-hand to address a particular clinical need.

While practical use of such systems is still in the future, the ability for the operator to construct new systems on-demand from off the shelf components has the potential to yield several important benefits:

- **Reduce Errors & Clinical Workloads** - The ability to automate workflows can reduce clinical errors, improving patient outcomes, while at the same time reducing clinician workload [70, 134].

- **Avoid Vendor Lockin** - Vendor neutral interoperability protocols means that HDOs would not be locked into a single vendor for all their medical systems. They would be able to create "best-of-breed" systems of systems by purchasing the best devices produced by different manufacturers [69].

Furthermore, it allows the HDO more flexible device procurement strategies in general [184].

- **Enable Innovation** - Interoperability could enable new startups to contribute new capability to the interoperable "ecosystem" via software applications. Currently, it is difficult for a company to create new medical systems capability unless that company is an incumbent because it is not practically feasible to add new capability independent of a physical medical device.

In anticipation of interoperable plug & play medical systems and the benefits they will bring there are a number of on-going efforts to create interoperability standards [156, 88, 23, 139, 71], build prototype implementations that aim to support this vision [108, 155], and develop new regulatory frameworks [105, 172].

1.2 Challenges

Plug & play medical systems will be assembled by their (non-technical) users which means that there will not be a single entity with technical competency (*e.g.,* prime contractor) positioned to assess the safety of a specific combination of devices. The lack of a traditional prime contractor poses a challenge to ensuring the safety of these systems for two reasons: First, safety is a property of systems that arises from the interactions between system components [126, 128]. Second, what constitutes safe inter-device interactions will vary considerably between different clinical scenarios. Often, we say that these two facets of safety mean that safety is an *emergent* property [126, 38, 127]. It is critical to ensure that interactions

between devices are predictable *and* that those interactions satisfy the safety requirements of the given clinical scenario. Since it will not be known specifically which devices will be assembled into the composite system *a priori*, we are presented with a number of challenges to achieving safe interoperability. Here we identify three significant challenges we try to address in this dissertation that must be overcome if safe interoperable plug & play systems are to be achieved.

Challenge 1: Current Regulatory Frameworks Won't Scale

Current regulatory frameworks (*e.g.,* the US FDA device approval process) are not designed with large scale, vendor neutral interoperability in mind. While the FDA approval process does not explicitly disallow interoperable or plug & play capabilities, it does make the approval of truly interoperable devices prohibitively costly.

For example, as a device manufacturer, it is possible to develop a device designed to work with other devices via a network interface. The FDA regulations would then treat the other devices as "accessories" [62, 142] and subject *each possible combination* of devices to regulatory approval. From the perspective of a safety assesment, subjecting each combination isn't a bad idea: It gives the manufacturer (and regulator) the opportunity to detect any unsafe interactions between specific devices.

However, If we want a large ecosystem of interoperable devices, regulating each combination is hugely impractical. Consider an ecosystem with n devices. If its possible for every *pair* of devices to interoperate this would require n^2 regulatory submissions (and approvals). Additionally, many medical systems would likely involve more than just pairs of devices. If we want to combine m deivces

together, then we will need n^m regulatory submissions and approvals (and if we want to combine *up to* m its $n^{m!}$!). Since the regulatory process is expensive for both manufacturers and regulators, the current approach is not practical for anything other than small values of n and m.

Challenge 2: Standardization of Behavior is not Practical

Historically, standards bodies have chosen to achieve interoperability by developing large, complex, and detailed standards that attempt to cover many (if not all) aspects of the interoperable domain. For example, consider the IEEE-11073 interoperability standard for medical devices. IEEE-11073 standardizes a nomenclature (a mapping of numerical codes to device types), a comprehensive domain information model (a definition of physiological signals and what device types can provide those signals), a communications model (how information is exchanged) and a data encoding specification [2]. The development of large complex standards is time-consuming and difficult: Work on IEEE-11073, started in 1986 [73] (as IEEE-1073) but didn't officially stabilize until 2005 [2]. Furthermore, the complexity and long development cycles of large complex standards can hinder their adoption, as has apparently been the case with IEEE-11073 [88].

Most standards like IEEE-11073 only cover *data interoperability*, *i.e.,* how to exchange data and how to interpret the data that is exchanged. However, reasoning about safety fundamentally comes down to reasoning about behavior. One approach could be to follow the footsteps of what the IEEE-11073PHD [52, 47] standard does for data but with behavior. IEEE-11073PHD defines the *types* of data each different type of device (*e.g.,* glucometer, scale, or pulse-oximeter) must be able to report. We could imagine doing the same, but for behavior (*e.g.,* define

precisely how an PCA pump *ought* to behave and bake it into the standard).

However, we contend that standardized behavior is not practically achievable. IEEE-11073 took almost two decades, was only about data, and hasn't found significant adoption. Device behavior adds another significant layer of complexity. Furthermore, among devices of the same type, their behavior can vary significantly between manufacturer and even model. Getting manufacturers to implement devices with precisely the same behavior is unlikely. There needs to be some other mechanism for devices to communicate their beahvior in a standardized way that also lets manufacturers and regulators assess safety.

Challenge 3: Timing Predictability in Open & Distributed Systems

Typical safety-critical systems are *closed*, meaning the components that comprise the system do not change while the system is active and deployed. The behavior of closed systems are easy, relative to open systems, to predict because the systems developers can constrain the possible component interactions at design time.

For example, consider a distributed, time-sensitive, closed system like the avionics on a modern commercial or military aircraft. Flight control signals generated by the flight control computer must reach the target flight-control surface at the right time or the aircraft will not fly as intended. In less extreme examples control authority would be diminished (loss of expected performance). In more extreme examples the aircraft could depart stable flight and possibly crash (loss of safety). Aircraft designers can ensure the distributed timing behavior of avionics systems because they know precisely what nodes are on the network, how much data each node will send, when each node will send data, and the timing requirements for each data flow. They can then use real-time scheduling theory [164] to

devise a transmission schedule that satisfies the timing requirements and specialized networking hardware (*e.g.,* Avionics Full-Duplex Switched Ethernet [150]) to enforce that schedule.

Unfortunately, plug & play medical systems are inherently *open*, meaning its users can add or remove components and otherwise change the configuration of the system while it is running. The designers of plug & play medical systems won't have the luxury of knowing specifically what components comprise the system, or what the timing requirements of the complete system are *a priori*. Traditional techniques and technologies designed to robustly ensure the timing behavior of the system can not be relied upon.

It will be important to find some way of guaranteeing the timing behavior of plug & play medical systems. Many of the clinical scenarios [139] that can benefit from plug & play medical systems outlined by the MD Plug & Play program involve device coordination that is fundamentally time sensitive. Furthermore, most medical closed-loop control systems require some bounds on timing to remain safe. While there are many medical system designs where a failure to achieve desired timing behavior (*e.g.,* a missed message delivery deadline) won't strictly cause an immediate safety violation (See, for example, the systems designs in [20, 104] or [21]) a divergence from the expected timing behavior usually causes the system to compensate with a fail-safe state (*e.g.,* deactivate an infusion pump) that will diminish the clinical utility of the system and should be avoided.

Of course, Challenges 1-3 are not the only challenges confronting safe plug & play medical systems, they are only the challenges we are seeking to address at some level in this work. Indeed, plug & play interoperability for medical devices

has been described as a *wicked* problem (*i.e.,* "*resistant to solutions; incomplete, contradictory, and changing requirements; and complex dependencies?*") [72]. For example, one broad, challenging, and important area this dissertation does not directly address is the *security* of plug & play medical systems (See [173, 174] or [22] for some of the security challenges).

1.3 Contributions

We propose a number of ways to partly address Challenges 1-3 and then evaluate our proposals. Our contributions are:

A Regulatory Framework & Assurance Argument Pattern

We propose and describe a "platform-oriented" regulatory framework that will allow for a type *"compositional certification"* and avoid the problem of a combinatorial number of regulatory submissions associated with current regulatory regimes (Challenge 1). The framework regulates three types of system components: interoperable medical devices, medical applications (apps), and medical application platforms (MAPs). The medical devices are certified to comply with an interface, applications are certified assuming they are used with devices that satisfy some stated interfaces, and the platforms run the applications and ensure applications are only coupled with compatible devices. The premise of our work is that *application vendors* will be able to to produce system-level safety arguments by leveraging assurances provided by the regulatory framework and ecosystem. We give an *assurance argument pattern* that captures essential reasoning needed to bring assurance associated with certified components into a system-level saftey

claim.

Time Parametric Modal Specifications

We describe Time Parametric Modal Specifications (TPMS). TPMS is a new *interface theory* with several features that make it useful for describing the device interfaces needed by our proposed regulatory framework. While we designed TPMS to work under our proposed regulatory framework, it is novel as an interface theory on its own. To our knowledge, TPMS is the first interface theory to allow the modeling of *timing* variability separately from *functional* variability and also allow for the parallel composition of interfaces. Both of these features can be useful to model families of real-time systems in general. In the context of our proposed regulatory framework, it will let us model device behavior (Challenge 2) and help formally reason about system behavior in a top-down fashion (Challenge 1).

The On-Demand Systems Description Language

We propose and describe the On-Demand Systems Description Language (ODSDL). The ODSDL lets application designers implement the algorithm of their application and specify both the logical architecture of their application and what devices their application requires. The ODSDL uses TPMS to let app developers and device manufacturers directly specify the behavior needed from the devices, avoiding the need to standardize the behavior of different device types (Challenge 2). The ODSDL also has Logical Execution Time (LET) semantics for both application tasks and distributed interactions (*e.g.,* signaling over a network). The LET semantics precisely define how the application behave with respect to

time, making it easier for the application developer to predict the behavior of their application when it is deployed (Challenge 3).

A Prototype Medical Application Platform

We describe the Medical Device Coordination Framework / MIDdleware Assurance Substrate (MDCF/MIDAS). MDCF/MIDAS is a protoype medical application platform where we implement many of the capabilities required of MAPs by our proposed framework. We focus primarily on how the MDCF/MIDAS is able to achieve the ODSDL's LET semantics in an open and distributed environment (Challenge 3) and perform an experimental evaluation of our implementation.

1.4 Organization

The rest of this dissertation is organized as follows:

In Chapter 2 we first provide several detailed motivating clinical examples. These examples will serve as running examples throughout the rest of the dissertation. Next, we describe our proposed regulatory framework and the associated assurance argument pattern.

In Chapter 3 we describe Time Parametric Modal Specifications (TPMS). We define TPMS, define and prove important properties (*e.g.,* refinement, property preservation, compositional reasoning) of TPMS, and then give some algorithms to decide important decision problems (*e.g.,* refinement & reachability checking) of TPMS.

In Chapter 4 we propose the On-Demand Systems Description Language (ODSDL). We give a sketch of the language and provide some simple examples

of its use to program a plug & play medical application. We also define its (LET) semantics and show how its semantics result in *input determinism* (*i.e.,* system behavior is completely determined by its inputs).

Chapter 5 we describe the MDCF/MIDAS. We enumerate its features, show its software architecture, and discuss the design features and scheduling techniques it uses to achieve the LET semantics required by ODSDL applications. We evaluate our implementation by running randomly generated ODSDL applications and showing that the implementation in fact achieves LET semantics.

Chapter 6 reports on two case studies. We use these case studies to illustrate how the different aspects of our contributions all work together to enable a system safety. For our case studies we picked two of the running examples originally described in Chapter 2. For each running example we describe some key assumptions on the regulatory framework and interoperability ecosystem, design an application using ODSDL, verify the safety of the designs, and produce a safety argument using the assurance case pattern originally introduced in Chapter 2. We also use the ODSDL artifacts produced for the case-studies to evaluate one of the important TPMS decision procedures using realistic examples.

In Chapter 7 we conclude by summarizing the work and performing a *gap analysis*. The gap analysis identifies the technical advances *and* social developments needed to make the framework described in this dissertation viable.

Chapter 2

Motivating Examples & a Regulatory Framework

2.1 Introduction

In this Chapter we first describe a number of real clinical scenarios that could benefit from plug & play medical systems. These scenarios will be used as running examples throughout this dissertation. Each of these scenarios represents a situation where automation could prevent clinical errors and subsequent injury to the patient. Automation of the type needed is generally not available due to the lack of interoperability between different devices and device types produced by different manufacturers. Under current regulatory frameworks, the creation of medical systems supporting the type of automation required would need device manufacturers to work together and prepare joint regulatory submissions; something that current manufacturers aren't likely to justify given the costs and perceived business benefits [125].

After we introduce the motivating examples we propose a regulatory framework based on a "platform-oriented" ecosystem of interoperable medical components. We give an overview of the stakeholders, their safety/assurances responsibilities, and the regulatory processes the different types of interoperable components will be subject to. Our goal is to allow application vendors to produce system-level safety arguments by leveraging assurances provided by the regulatory framework and ecosystem. We show how ecosystem level assurance can be combined into a system-level safety argument using an "assurance case pattern". Assurance cases are informal (but structured) documents that give an argument why we should believe that a system satisfies some property. While an assurance case is not a proof, we give an informal discussion of the underlying logic of our argument.

2.2 Motivating Examples

2.2.1 Xray/Ventilator Coordination

A simple example of automating clinician workflows via cooperating devices addresses problems in acquiring accurate chest x-ray images for patients on ventilators during surgery [118]. To keep the lungs movements from blurring the image, doctors must manually turn off the ventilator for a few seconds while they acquire the x-ray image, but there are risks in inadvertently leaving the ventilator off for too long. For example, Lofsky [131] documents a case where a patient death resulted when an anestheseologist forgot to turn the ventilator back on due to a distraction in the operating room associated with dropped x-ray film and a jammed operating table:

A 32-year-old woman had a laparoscopic cholecystectomy performed under general anesthesia. At the surgeons request, a plane film x-ray was shot during a cholangiogram. The anesthesiologist stopped the ventilator for the film. The x-ray technician was unable to remove the film because of its position beneath the table. The anesthesiologist attempted to help her, but found it difficult because the gears on the table had jammed. Finally, the x-ray was removed, and the surgical procedure recommenced. At some point, the anesthesiologist glanced at the EKG and noticed severe bradycardia. He realized he had never restarted the ventilator. This patient ultimately expired.

These risks can be minimized by automatically coordinating the actions of the x-ray imaging device and the ventilator. Specifically, a centralized automated coordinator running a pre-programmed coordination script can use device data from the ventilator over the period of a few respiratory cycles to identify a target image acquisition point where the lungs will be at full inhalation or exhalation (and thus experiencing minimal motion). At the image acquistion point, the controller can pause the ventilator, activate the x-ray machine to acquire the image, and then signal the ventilator to unpause and continue the respiration [74]. Note that this case above involves a very simple form of coordination logic that can significantly improve the safety or the effectiveness of treatment for the patient. In our experience, once the concept of device coordination is explained to a surgical clinician, they can almost always come up with an scenario that they have encountered where device coordination would be beneficial. For example, consider the example from modern ear-nose-throat surgery in the next sub-section.

2.2.2 Laser-Ventilator Interlock

Many modern Ear-Nose-Throat (ENT) surgeries are now performed with a laser-scalpel. While laser-scalpels can have different designs and utilize different physical processes to create the laser (*e.g.,* [141] or [8]), the fundamental concept of operations is the same: Laser-scalpels create a high energy laser beam that surgeons can use to cut flesh [106]. When the surgeon wants to begin cutting s/he will aim the scalpel where s/he wants to cut and then toggles the beam on. When the cut has been completed s/he toggles the beam off.

In many cases, laser scalpels provide a number of benefits and better patient outcomes versus normal blades [16, 178, 153]: First, laser scalpels can result in less bleeding because the heat generated by the laser beam automatically cauterizes the wound. Second, swelling can be reduced because the cauterizing effect also seals nerve endings and small lymph vessels. Finally, since there is no contact between the surgical instrument and the patient, there is a reduced risk of infection.

Laser-Ventilator Hazards

While there are several accepted benefits of laser-scalpels for ENT surgeries, the introduction of a laser to ENT surgeries increases the risk of surgical fire [166, 167]. The increased fire risk arises from the fact that ENT surgeries are performed under general anesthesia: The patient will be unconcious and supported by mechanical ventilation where they will be intubated with a breathing tube carrying a gas mixture of concentrated O_2. The close proximity of an oxidizer (O_2) with a heat source (laser scalpel) is dangerous and can lead to a fire *inside the patient* if the surgeon accidentally impinges the breathing tube [147].

In current practice this hazard is mitigated manually by the surgical team: When the surgeon wants to cut, s/he will signal to the anethesiologist, who will reduce the O_2 flow to the patient (if it is safe to do so). As the surgeon cuts, the anethesiologist will monitor the SpO_2 levels of the patient. When the SpO_2 approaches some defined lower threshold the anesthesiologist will ask the surgeon to stop cutting and they will resume the flow of O_2.

Despite these hazard mitigation efforts, approximately 650 surgical fires are reported in the United States each year and it is estimated that nearly four times that number go unreported [117, 61]. Clearly, some more effective measures are needed. One possible solution is to build a system of medical devices that automatically enforces mutual exclusion between laser-scalpel and ventilator activity.

2.2.3 Closed-Loop Management of Patient Controlled Analgesia

The use of Patient Controlled Analgesia (PCA) infusion pumps has emerged as the premier process for meeting the goals of pain management. The computerized pump is loaded with an analgesic drug such as morphine, fentanyl, or hydromorphone and can be programmed with a background, or basal, infusion rate as well as a bolus dose. The basal infusion rate is delivered constantly and is selected to be sufficient to control the patients normal pain level. The bolus dose is an additional quantity of drug that is delivered only when the patient requests it by pressing a button. The pumps are also programmed with dose limits that are set for the specific patient, *e.g.,* only allowing one dose to be delivered within a certain time frame. In addition to the drug delivery mechanism itself, components of the PCA process include appropriate patient selection, proper patient education,

frequent patient assessment, and collaboration among the prescriber, pharmacist and nursing staff.

Patient controlled analgesia provides consistent control of pain by allowing patients to self-administer doses of a drug. Evidence from systematic reviews of randomized controlled clinical trials indicate that the use of IV PCA leads to better pain relief, improved patient outcomes (*e.g.,* reduction in pulmonary complications) and increased patient satisfaction compared with conventional nurseadministered parenteral opioids [91]. The ability of patients to maintain some control over their care appears to be a strong contributor to PCA associated improvements in patient satisfaction. One of the major opioid side effects is respiratory depression. Opioids have a direct effect on the respiratory center in the medulla [44]. Respiratory depression increases progressively with dose. The use of background infusions in some patients may provide increased pain relief however this increases the risk of respiratory depression and has led to a general recommendation of eliminating background infusions. Symptoms of respiratory depression include increasing sedation, decreased respiratory rate, decreased oxygen saturation and increased end tidal carbon dioxide [115]. Breathing may become irregular and periodic.

To address these issues, current nursing standards of care for monitoring patients during PCA administration include assessment of pain and sedation, along with heart rate, blood pressure and respiration rate every four hours. Pulse oximetry (SpO_2) is used to monitor falling arterial oxygen saturation.

PCA Hazards

Despite these positive outcomes, PCA pumps are also associated with a large number of adverse events [82, 100]. The most common type of adverse event is oversedation [133]. An excessive dose of the analgesic can cause neurologic depression which may lead to respiratory depression and eventually respiratory distress. In extreme cases the patient may not be able to breathe adequately, leading to death. Overdoses may have many causes including programming errors [75], the use of the wrong concentration of drug, drug interactions, and PCA-by-proxy.

Programming errors may be caused by confusing drug names, *e.g.,* hydromorphone and morphine or morphine and meperidine [82], by making a mistake in dose or drug concentration calculations [179, 82] or entering the wrong values for bolus dose size, infusion rate, or lockout interval. A common source of error is entering a value that is off by a power of 10 or using the wrong units. For example, entering 5 mL / minute instead of 5 mG / minute or programming a pump with a drug concentration of 1 mG/mL when it is actually 10 mG/mL [82]. [179] discusses a number of cases where patients were fatally overdosed because of an improperly programmed drug concentration.

When someone other than the patient presses the button to request a bolus dose, it is called PCA-by-proxy. Normally if the patient is oversedated they are unable to press the button to get another bolus dose. If someone else presses the button, this safeguard is bypassed and an overdose may occur. In 2004 the Joint Commission made PCA-by-proxy their 33rd sentinel-event. Sentinel events are occurrences that must be reported and investigated to their root cause or the facility risks losing their accreditation [99]. Healthcare facilities that have completed staff education programs and incorporated a warning about PCA-by-proxy into

their patient education have seen lower overall rates of oversedation [99].

An analysis of reports to the MAUDE database maintained by the Food and Drug Administration (FDA)s Center for Devices and Radiological Health (CDRH) from 1984 to 1989 found that 67% of problems associated with PCA pumps were caused by operator error [46]. This early study took place before the 1990 change in Federal Reporting Guidelines that requires reporting of incidents involving device malfunctions and serious injuries or deaths to FDA. A later study [76] found that nearly 80% of the 2009 reported incidents in 2002 and 2003 were blamed on device malfunctions and that nearly 65% of these suspected device malfunctions were confirmed by the device manufacturers. The human factors of pump interface design are an important means of reducing use errors [19, 20]. Respiratory depression associated with PCA varies between 0.3% and 6% depending on the patient population and how respiratory depression is defined [152]. Most cases of respiratory depression do not lead to permanent harm to the patient, but these still represent serious incidents with the potential to harm or kill patients.

The Institute for Safe Medicine maintains a voluntary database of medication errors. This MedMarx database contains 9500 PCA related errors in the span 2000 - 2004 [82]. These account for only 1% of the medication errors submitted to the database, but this 1% accounts for 6.5% of harmful outcomes. This almost certainly under-reports the actual number of occurrences, since the voluntary database can only track the rate of reporting, not the rates of errors or adverse events [123]. Adequate pain control provides benefits including improved patient satisfaction, lower rates of complication, reduced length of hospital stays, and lower rates of litigation [82]. Some biomedical engineers take the attitude that the only safe medical device is one thats never taken out of the box, but discontinuing

use of PCA pumps is simply not an option. While providing inadequate levels of medication would indeed reduce the chance of overdose, pain management is an essential part of the care of these patients.

As noted earlier, patients receiving PCA therapy are usually also connected to a patient monitor that records their vital signs. These monitors typically measure at least heart rate, blood pressure, respiratory rate, and oxygen saturation (SpO_2). The monitor has simple alarms which sound when the vital signs go outside of some preset limits. If the patient receives an overdose, their vital signs will eventually go outside of the limits and the alarms will sound, summoning a caregiver to the bedside.

However, by the time their vital signs drop far enough to cause the alarm to sound, damage may have already been done. Caregivers are desensitized by frequent false positive alarms, and they may not respond as quickly as would be optimal. Furthermore, the infusion pump continues running until it is manually stopped by a caregiver, which may not happen immediately on their arrival at the bedside.

If there was an ecosystem of interoperable or plug and play medical devices with the appropriate capabilities it would be possible to write an application that automatically takes data from a monitoring device like a pulse-oximeter and then deactivates the pump if there is the potential for overdose.

2.3 A Platform-Oriented Ecosystem

We believe that a platform-oriented ecosystem of medical components, if appropriately designed, could be used to ensure the safety of plug & play medical

systems. In this paper, we define an *ecosystem* as as set of devices, software applications and computational platforms intended to interact with one another using standardized plug & play interoperability protocols; the stakeholders that organize, manufacture, and use these products; as well as the explicitly defined processes that are followed to develop, certify, and use these products.

Our proposed ecosystem contains three categories of interoperable system components. The component categories are *device*, *application*, and *platform*. Devices expose a logical interface that acts like an API which applications can use to control or receive data from the device. Applications implement the clinical algorithms used to address a specific clinical scenario. Applications are not just executable code; they have a requirements specification which declares what interfaces compatible devices must implement and a QoS specification that declares timing requirements (*e.g.,* periods and deadlines on program execution). Each platform consists of a network, computational resources (CPU, RAM, *etc.*), real-time operating system and *platform services*. The platform's job is to act as a trusted base to enforce the correct assembly of on-demand systems: When the Health Delivery Organization (HDO) plugs a device into the platform the device will upload its interface specification. Then, when the HDO staff tries to launch an application with a set of selected devices the platform will (1) check if those devices' interfaces are compatible with the application requirements and (2) verify that the application's requested QoS can be guaranteed. If either 1 or 2 is false the platform will prevent application launch.

There are a number of *actors* that participate in our vision of the ecosystem. Each actor has different responsibilities and assurance obligations:

- *The Ecosystem Standards Consortium* consists of representives of the other

actors and follows a consensus process to define ecosystem standards: The connectivity protocols used by each component to exchange data, the logical interfaces devices can implement, what it means for a device to be compatible with an application, and the compliance requirements that each type of component (applications, devices, and platforms) must satisfy before that component can be certified as a member of the ecosystem. We emphasize that the consortium does not explicitly define specific systems - rather it establishes constraints on the architecture and interfaces of such systems and their sub-components.

- *The Device Vendor* designs, manufactures, and markets their devices. Before their device can be admitted to the ecosystem they must provide assurance (*e.g.,* via an assurance case) that their device satisfies the ecosystem compliance requirements for all interfaces the device claims it implements.

- *The Application Vendor* is responsible for providing assurance that their application is safe when instantiated with compatible devices. Application vendors play a role analogous to"system integrators" in conventional systems: They define the overall system function, and reason about overall system safety. However, the distinction is that they define the system using a software application and requirements/assumptions on the devices and platforms. They do not specify a single system but a family of possible system instances that satisfy the functional and safety goals of the clinical scenario. Thus, the integration is "virtual": they do not integrate specific physical devices and platforms but specifications of devices and platforms where each such specification represents a set of compliant components.

- *The Platform Vendor* must provide assurance that their platform will correctly perform its responsibilities. Because the platform is the trusted base for each system these responsibilities include correctly executing application code, correctly implementing the ecosystem device-application compatibility check and providing adequate system security.

- *The Certification Authority* polices component membership in the ecosystem: The certification authority only grants certification to components that satisfy the ecosystem compliance requirements. When a component becomes certified the authority will sign the component with a digital certificate. If postmarket surveillance reveals that a component has a previously undetected problem resulting in non-compliance, the certification authority can revoke the certificate associated with that component's make and model. The digital certificate enables the platforms to use cryptographic methods to verify whether or not applications or devices have been certified. [77] contains an overview of how cryptographic methods can be used to establish trust and how the platform acts as a trusted base for this process.

- *The HDO* does not have assurance responsibilities *per se* (*i.e.,* they are not required to provide assurance to any other ecosystem actor) however, they must still use the application as intended. If the HDO uses an application in an unintended way (*i.e.,* off-label use [30, 168, 163]) then the (safety) assurances provided by the Application Vendor for that application are not guaranteed to apply.

The ecosystem assurance and compliance obligations (combined with the runtime checks performed by the platform) create a series of "gating functions" that

prevent the HDO from assembling potentially unsafe combinations of devices and applications. Figure 2.1 illustrates the relationship between the ecosystem actors, ecosystem components, the gating functions and the final physical instantiation of a system. The unringed circles indicate steps in development or assembly of the physical system. Lines indicate interactions between the actors. The ringed circles indicate completion of one of the primary assurance steps and represent the gates. First the Ecosystem Standards Consortium must establish the ecosystem standards and component compliance requirements. Once the standards have been defined the component manufacturers can design their respective components. The certification authority enforces the first set of gates: components are only allowed into the ecosystem if they satisfy their respective compliance requirements. The final set of gates are enforced by the platform: The platform will only let the application run if it is being paired with compatible and compliant (*i.e.,* certified) devices and if the platform can guarantee that the application's QoS requirements will be met.

Interfaces, Compatibility, and Device Compliance:

For the purposes of this chapter[1] we imagine that device and application interfaces are analogous to software interfaces from programming languages like Java: When an application specifies that it requires a device interface it is much the same as declaring a field variable in a Java class to have an interface-type: Any object that implements that interface can be substituted for that variable. Compatibility checking between devices and applications thus amounts to checking if the device implements the required interface(s). A device is compliant with an interface if it

[1]Later chapters will introduce a more sophisticated interface theory: TPMS.

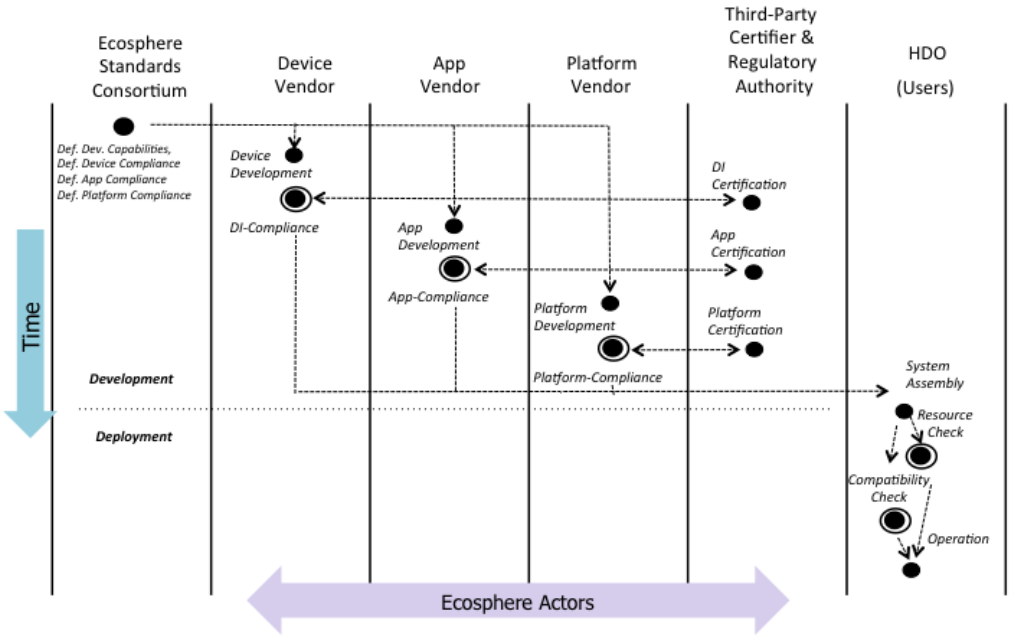


Figure 2.1: Ecosystem actors, their interactions and certification activities.

satisfies the Consortium defined compliance requirements for that interface type. Consider the PCA pump from the motivating example. The Ecosystem Consortium could define a standardized interface for PCA pumps called "void $\mathsf{InfusionTimedTicket}(x)$" which applications could use to send a timed ticket to the pump. The Consortium would then define the behavior a PCA pump must have in order to comply with that interface. In this case, the pump should correctly implement the ticket timer and cease infusion after some mandated amount of time.

Platform Assurance & Compliance:

A compliant platform must correctly implement compatibility checking and resource management. Ideally, applications would be portable across platforms in the ecosystem. This means that the Consortium would also standardize an execution model for the applications (*i.e.,* application byte code format, semantics, and available APIs). A compliant platform must then also correctly implement the standard model of execution. Different applications will have different levels of criticality: Applications with low criticality do not pose serious consequences in the event of failure while failure of a high-criticality application may result in catastrophic consequences. Because the application is totally dependent on the platform to function correctly, the assurance requirements for each platform should be at least as stringent as the assurance requirements for the most critical application that will be admitted to the ecosystem. While the specifics of these requirements are well beyond the scope of this paper, we can imagine that the Consortium could mandate that all Platform Vendors follow guidance that would result in levels of assurance similar to that of DO-178C Level A. [83].

2.4 The Platform Argument Pattern

Each on-demand application defines a *set* of possible systems: One for each allowed combination of devices and platform with the application. The multitude of potential systems implied by a single application presents a challenge for both the application developer and Certification Authority. The application vendor will need to devise an assurance argument that explains why all these possible combinations are safe. Practically, it won't be possible for the vendor to analyze or

test each combination individually because the number of possible combinations would prohibitively large. Additionally, new components (*i.e.,* platforms & devices) may be admitted to the ecosystem *after* the application is certified. Because the application vendor will not be able to directly analyze all possible device combinations they will have to use some form of model-based reasoning: They would analyze their application for safety using models as proxies for the concrete devices. In theory, as long as the models capture the range of behavior allowed by the different ecosystem gating functions (*i.e.,* the compliance/certification checks and the platform compatibily checks) then safety conclusions derived from the model-based reasoning should hold for any allowed instantiation of the application.

Of course, in practice, it is generally impossible to capture *all* the allowed behavior of a physical system in a model. If an application vendor is using model-based reasoning to support safety-claims they should justify *why* the models they used are *adequate*. In our context, adequacy depends on the intended use of the application (*i.e.,* the meaning of adequate will vary from application to application) *as well as* the assurances on each component provided by the ecosystem itself. To this end we propose an assurance argument pattern that requires application vendors to make model adequacy arguments explicit. Our hope is that it can help both application vendors and the Certification Authority to quickly identify assurance deficits or other fallacious reasoning in application assurance arguments, especially those related to model-based reasoning. The remainder of this section is organized as follows: First we define the terms the pattern uses. Then we introduce and explain the platform argument pattern. We will show how this pattern can be instatiated with our case studies later in Chapter 6.

2.4.1 Pattern Terms

Figure 2.2 maps out the terms used in the pattern. Our terms make an explicit distinction between models[2] and physical embodiments. We ultimately care about the physical embodiments but we are left with the models to analyze. The rows correspond to the different types of ecosystem components (with the addition of a row for the environment and instantiated system). The columns separate out different abstractions for each of the component categories: The specifications refer to the actual specification artifacts created by either the application developer or device manufacturer, the models are semantic (*i.e.,* analyzable) objects created by the application developer based on the specifications. The last column (physical embodiments) represent the physical object that correspond to the models and specifications.

Each entity in Figure 2.2 is defined as follows: The l devices admitted to the ecosystem are $D_1, \ldots, D_l$. Each D_i is compliant with its interface DI_i, Each application consists of an A and set of AI_j. The A is the algorithm of the application and represents executable code. Since these applications are typically real-time we assume any QoS specifications in the application are contained within A. The AI_js represent the application's required device interfaces (If the application uses n devices then $1 \leq j \leq n$). The physical emobodiment of each AI_j is the set of devices that implement the interface AI_j (we use $\simeq$ to represent the compat-

[2]Through out this section we adopt a formal notation that might lead some readers to believe that when we use the term "model" we are explicitly refering to formal models (*i.e.,* ones that could be analyzed by a model-checker). This is not the case. We are using "model" in a very general sense and a model could range from an informal "mental model" to an executable model that could be simulated to a fully formal model that could be analyzed by a model-checker.

		Model	Specification	Physical Embodiment
Devices		-	$DI_1, \ldots, DI_l$	$D_1, \ldots, D_l$
App	Algorithm	A^m	A	$P(A)$
	Interface	AI_j^m	AI_j	$\mathbb{D}_j = \{D_i \mid DI_i \simeq AI_j\}$
Platform		-	-	P
Environment		E^m	-	E
System		$A^m \mid\mid_{j=1}^n AI_j^m \mid\mid E^m$	$A \mid\mid_{j=1}^n AI_j$	$\{P(A) \mid\mid_{j=1}^n D_j \mid\mid E \mid D_j \in \mathbb{D}_j\}$

Figure 2.2: Pattern Terms: The relationship between models, specifications, and physical embodiments.

ibility relation). The AI_j^m are models created by the application developer and are intended to capture all the behaviors of the devices that implement the AI_js. Since A is a program, it has no physical embodiment until it is executed on a platform, therefore $P(A)$ represents platform P executing A. The device interfaces of an application are syntactic objects. They don't have explicit semantics but they do imply a set of behaviors (*i.e.,* the union of the behaviors of all the compliant devices that are compatible with that interface)[3]. Each platform is represented by a P. E represents the environment where the application will be deployed and E^m is the model of that environment. The last row are the system entities. $A^m \mid\mid_{j=1}^n AI_j^m \mid\mid E^m$ is the model of the system. It is the composition of the application model, the device models, and the environment model (We borrow the parallel composition operator, $\mid\mid$, from process algebras to denote the combination of two or more components running together). $A \mid\mid_{j=1}^n AI_j$ (*i.e.,* the application) represents a specification of the system. $\{P(A) \mid\mid_{j=1}^n D_j \mid\mid E \mid D_j \in \mathbb{D}_j\}$ is the

[3]This is true for the purposes of this section. Later, we will introduce a specification theory called TPMS for device interfaces that does have semantics.

set of possible physical systems specified by the application (one system for each compatible combination of application and device(s)).

2.4.2 The Pattern

Figure 2.3 is a specification of the argument pattern using Goal Structured Notation (GSN) [102]. There are five components of a GSN-formatted diagram. First, *goals*, which are represented by squares, establish some claim to be argued for. Second, *solutions*, which are represented by circles, explain how those goals will be met (and are usually used to refer to evidence). Third, *strategies*, which are represented by parallelograms, establish an argumentation strategy (the soundness of which is typically either intuitive or established externally). Fourth, *contexts*, which are represented by stadiums, provide a specification of "the context within which safety is to be argued" [102]. Contexts apply to all children of the node the context is attached to. Finally, diamonds represent undeveloped goals (also often used as placeholders for goals that are elided due to space). Because Figure 2.3 is a pattern specification the subject(s) in each node are variables surrounded by curly braces; this indicates that the variables are placeholders that will be filled in with concrete subjects when the pattern is instantiated. If a connector contains a dot next to a variable (*e.g.,* n), that indicates a multiplicity of sub-goals that must be unrolled when the pattern is instantiated.

The top level goal (**G:AllSat**) states that all instantiantions of the application must satisfy some property ϕ in a specified environment. Assurance for this claim is argued via the platform argument strategy (**S:PlatArg**). The strategy must always be applied in at least two contexts: One referencing the models used in the model-based reasoning and the other referencing the ecosystem as-

surance and compliance requirements. **S:PlatArg** requires adequate assurance for three sub goals. The first goal (**G:ModelSat**) is the model-based reasoning step. The argument application vendor must argue that the chosen models satisfy ϕ. The remaining two goals explicitly relate the models used in **G:ModelSat** to the possible physical systems via the ecosystem assurance and compliance requirements. **G:ModelsAdequate** asks the developer to argue why the models chosen in **G:ModelSat** capture all the possible (relevant) behaviors allowed by the application's specification. Typically, the arguments for the adequacy of the environment, application, and devices models will all take on a different character so the pattern separates the arguments for each as a different sub-goal (Note the multiplicity on the **G: DevModel**$\{$**N**$\}$**Adq.** that forces a sub-goal for each device model). **G:PlatformAssurance** asks the developer to argue why the minimum level of assurance provided by any ecosystem compliant platform is sufficient to support the application: The application developer relies on the platform to correctly execute their application and ensure that the application is only instantiated with compatible devices. If a platform fails to do either of these correctly, then ϕ could be violated even if sound models were used in **G:ModelSat**.

2.5 Related Work

The original impetus for the research that resulted in this dissertation came from the Medical Device Plug & Play (MD PnP) program at CIMIT [139]. The ultimate goal of the MD PnP program is a medical device ecosystem where medical devices can seamlessly integrate in a plug & play manner. The MD PnP program has spawned a great deal of research seeking to understand the safety aspects of inter-

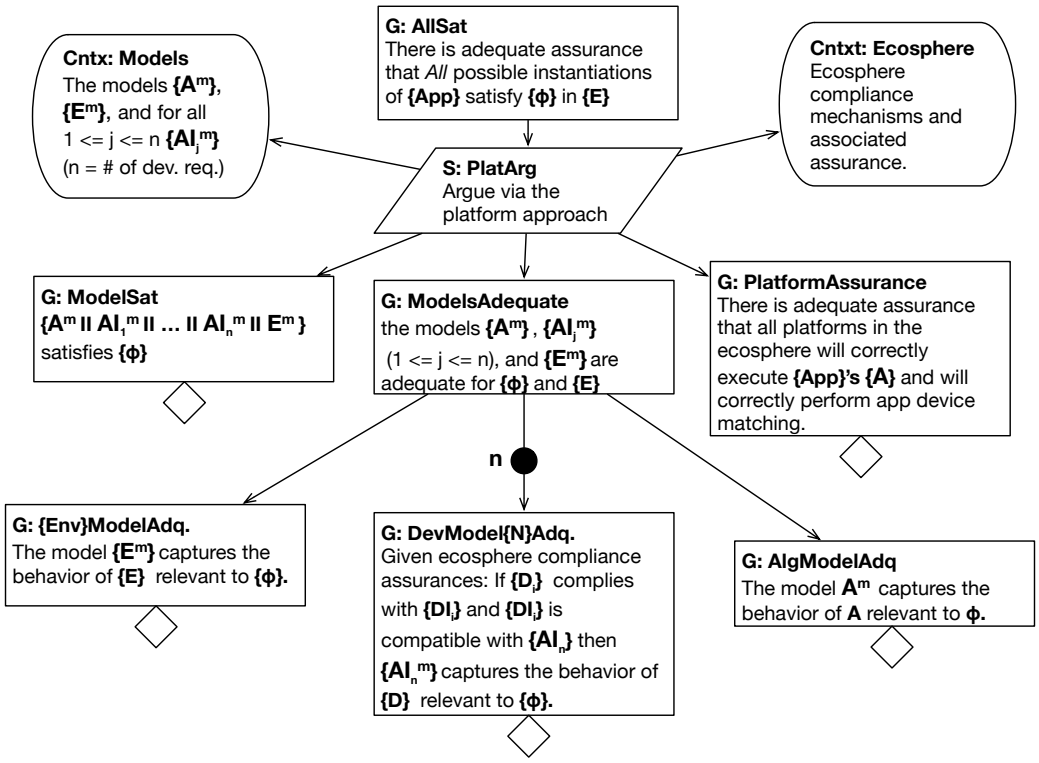


Figure 2.3: The argument pattern for application assurance.

operable medical systems and much of that research either inspired or influenced the contents of this dissertation.

One result of the MD PnP program is the Integrated Clinical Environment (ICE) standard [23]. ICE defines a functional architecture for plug & play medical systems (see Figure 2.4). The architecture consists of a supervisor computer to run supervisory applications, a network controller to manage communications between devices and applications, and the interoperable devices themselves. In terms of the vocabulary of this dissertation we can think of the supervisor and networkcontroller as the platform. While ICE defines a platform-based architecture at a high-level it does not specifically proscribe the technologies required to make

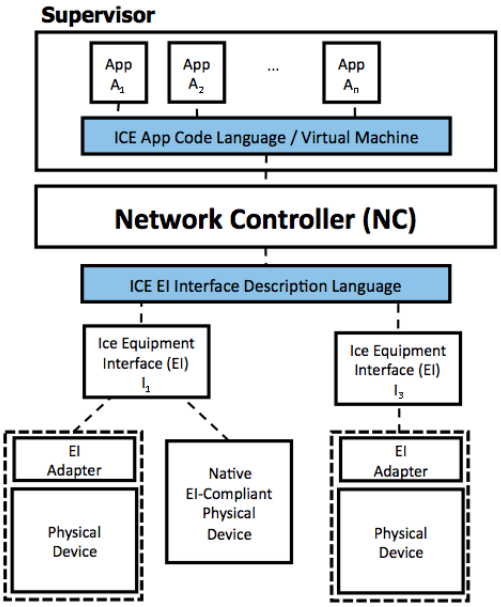


Figure 2.4: The Integrated Clinical Environment functional architecture.

the system work, nor does it describe how to formulate a regulatory framework or make assurance arguments.

There have been a number of research efforts to understand how to engineer medical applications *assuming* an interoperable ecosystem exists. [20] describe how to design an formally verify an application that synchronizes the operation of an x-ray and ventilator. [104] developes a device coordination protocol to ensure that a laser scalpel and ventilator can safely synchronize even in the presence of network faults. [149] developed and verified a closed-loop algorithm that manages the delivery of opiods administered by a PCA pump and maintains patient safety even in the presence of network outages.

One problem faced by the platform approach is *trust*, *i.e.,* how can a platform trust that a device actually has the capabilities it claims it has and is not counter-

feit? One solution to this problem was described in [77]. Their solution allows the platform to determine the authenticity of a device by examinging a device's cryptographic certificate: Whenever a device is certified the certifier will sign the devices cryptographic certificate. Then the platform determines the authenticity of the device much in much the same way as a web browser determines the authenticity of a remote server via SSL.

While, as far as we know there has not been any work on assurance arguments for plug & play or on-demand systems, there has been some work on assurance arguments for model-based development [28]. The authors of [28] describe an assurance argument pattern for systems developed using a model-based development process. Like our proposed pattern, their pattern requires that the argument preparer to first prove a propery using a model, and then justify the use of that model. Their pattern does not address the peculiarities of model based reasoning for on-demand systems. There has been some interesting work on modular certification [161] and compositional safety arguments [103]. These works are primarily concerned with argument reuse but introduce some concepts that may be applicable to providing assurance for on-demand systems.

Chapter 3

Time Parametric Modal Specifications

Our goal is to make a formal specification language to model device behavior suitable for use in a regulatory framework as described in Chapter 2. Application developers would use the formalism to express requirements on device behavior and would embed the requirements specification into the application. Device manufacturers would use the formalism to express the behavior of their device and would embed the behavior specification into the device. When the user tries to start an application with a set of devices, the platform will check to see if the devices behavior specification satisfies (*i.e.,* is compatible with) the applications requirements. Ideally, the formalism would allow for explicit descriptions of reactive component behavior, allow for variability (*i.e.,* let application developers indicate acceptable variability), enable top-down model-based reasoning, support automated compatibility checking between device behaviors and application requirements, and allow us to model variablilty in allowed behavior (both functional

and timing).

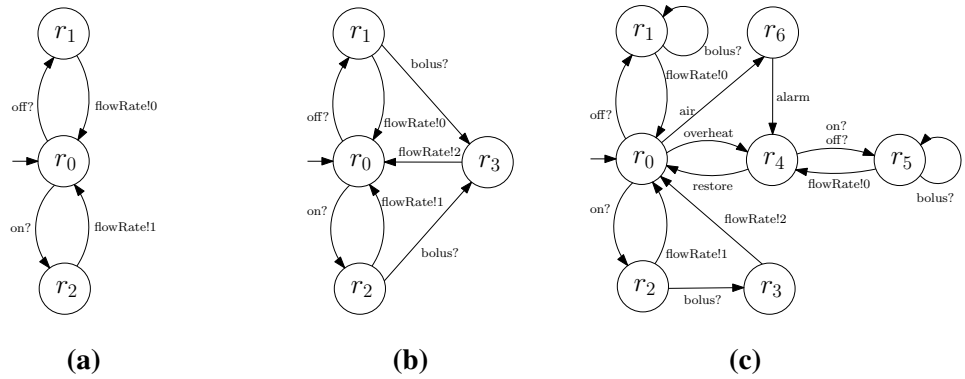


Figure 3.1: Labeled Transition Systems for three different types of PCA pumps. devices.

Modal specifications [15] are category of formalisms with attributes that make them attractive as a formalism for device behavior modeling. First, they allow for the explicit modeling of component behavior via labeled transition systems (LTS). For example, Figure 3.1 shows the (unitmed) behavior of three different types of infusion pumps modeled as LTSs. The labels on the edges indicate interactions with the environment. The pump in Figure 3.1a is very simple, it outputs flow as long as it receives an on signal, and it outputs no flow while it receives an off signal. Figure 3.1b models a simple PCA pump. It behaves like the pump of Figure 3.1a except that it outputs an increased flow when it gets a bolus signal. Figure 3.1c models a "smart" PCA pump. The pump will automatically shut itself off if it detects some dangerous situations (*e.g.,* motor overheating or bubbles in the infusion line).

Second, modal specifications are designed to allow for functional variability. Modal specifications get their name because they extend classical labeled transi-

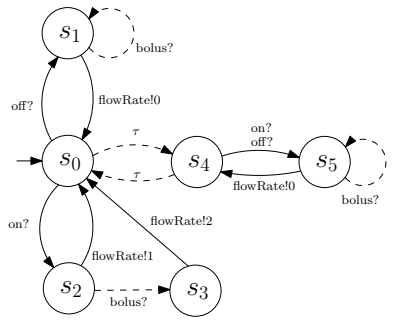


Figure 3.2: Example MTS specification of a PCA pump.

tion systems with a *modality* on transitions. *Must*-transitions specify behaviors all satisfying implementations are required to contain. Figure 3.2 shows an example modal input/output automata (MIOA) [121] specifying a family of infusion pump behaviors. The solid lines indicate must-transitions and the dotted lines indicate may-transitions. Implementations (*i.e.,* LTSs with no may-transitions) are related to specifications via a *refinement relation*. We say that an implementation *refines* (*i.e.,* satisfies) a specification if it has all the required behavior of the specification and all of its behavior is allowed by the specification. The pumps modeled in Figure 3.1 are all refinements of the MIOA specification in Figure 3.2. In this way, modal specifications would let application designers succinctly specify the set of device behaviors compatible with the safety and effectiveness goals of their application.

May/must modality is useful for an on-demand ecosystem because it allows application developers to be explicit about which behaviors are necessary and which behaviors are irrelevant. For example, lets say an application developer knows that their application needs an infusion pump but the safety of their application is not affected by the pump shutting itself off for some unspecified reason.

They could specify the device requirements of their application using the MIOA in Figure 3.2. This specification would let users use any of the pumps from Figure 3.1 with the application (as well as any others that refines the specification), increasing the flexibility of the application.

Third, many modal specification theories inherently allow for top-down reasoning. For most modal specification formalisms it is possible to show they preserve many useful properties (*e.g.,* safety or liveness) under composition and refinement. Imagine if our on-demand ecosystem used modal specifications to express device behaivor and uses modal refinement as a means to check device application compatibility. If so, it would be possible to verify system safety properties by verifying a system model created by composing a model of the application with its device requirements specification. Then, because the platforms would use refinement to check compatibility between applications and devices we would be ensured that any allowed combination will satisfy the safety properties.

Lastly, it's important for a behavior specification formalism to have the ability to model behavior w.r.t. wall clock time: Most medical systems are inherently real-time systems, *i.e.,* if they do something at the wrong time their safety or effectiveness could be compromised. While there are a collection of modal specification formalisms with support for modeling time (*e.g.,* [68, 182, 35, 56, 34]), they either dont treat timing variability separately from functional variability (*e.g.,* to model differences in the precision of actuators or processing capabilities of different devices), or they dont support the composition of specifications (*e.g.,* [32]). The inability to model timing variability separately from functional variability can be awkward in practice: Effectively, it means all implementations of a timed modal specification must inherit *all* of its timing behavior, including any non-

determinism or imprecision. We argue that it is important for designers to specify whether implementations can refine the timing behavior of specifications, *e.g.,* to allow for implementations that are faster or more precise.

In this Chapter we introduce Time Parametric Modal Specifications (TPMS). TPMS allow for the specification of functional variability separately from timing variability, support parallel composition, and preserve important (safety and liveness) properties under refinement and composition. To the best of our knowledge, TPMS is the first modal specification theory that explicitly supports parallel composition of specifications and the modeling of timing variability.

This Chapter is organized as follows: First we give a *notation glossary* in Section 3.1. Then, we formally define TPMS, their semantics, notion of refinement, parallel composition and property preservation in Section 3.2. Section 3.3 gives a number of procedures for deciding important decision problems (*i.e.,* reachability checking and refinement checking). Section 3.3 gives a survey of related work and explains why other modal specifications don't separate functional from timing variability.

3.1 Notation Glossary

$\mathcal{I}, \mathcal{M}, \mathcal{P}, \mathcal{Q}, \mathcal{S}, \mathcal{X}, \mathcal{Y}$ ..... A specification (*i.e.,* a IOTA, MIOTA, or TPMS).

I, M, P, Q, S, X, Y ..... Transition transition system (TTS or MTTS).

α, β, γ ..... Parameters.

a, b ..... Actions.

c, x, y ..... Clock variables.

v, p, f ..... Valuations.

r ..... Clock reset set.

s ..... State.

S ..... States.

Θ ..... The set of natural-numbered parameters for a TPMS.

$\sim$ ..... Used where the comparison operators $<, \leq, >$, or $\geq$ could be used.

$\models_{\diamond}$ ..... May satisify.

$\models_{\square}$ ..... Must satisify.

$\hookrightarrow_{\square}$ ..... Syntactic must transition (sometimes set of).

$\hookrightarrow_{\diamond}$ ..... Syntactic may transition (sometimes set of).

$\rightarrow_{\square}$ ..... Semantic must transition (sometimes set of).

$\rightarrow_{\diamond}$ ..... Semantic may transition (sometimes set of).

$\rightarrow_{\mathsf{d}}$ ..... Semantic delay transition (sometimes set of).

$\rightsquigarrow_{\square}$ ..... Zone-graph must transition (sometimes set of).

$\rightsquigarrow_{\diamond}$ ..... Zone-graph may transition (sometimes set of).

$\rightsquigarrow_{\mathsf{d}}$ ..... Zone-graph delay transition (sometimes set of).

$\sqsubseteq$ ..... Refinement between syntactic specifications (*e.g.,* TPMS or MIOTA).

$\underset{\approx}{\sqsubseteq}$ .. Weak refinement between syntactic specifications (*e.g.,* TPMS or MIOTA).

$\preceq$ ..... Refinement between transition systems (*e.g.,* TTS or MTTS).

$\precsim$ ..... Weak refinement between transition systems (*e.g.,* TTS or MTTS).

$[\![\mathcal{M}]\!]$. Semantics of $\mathcal{M}$. If $\mathcal{M}$ is an MIOTA $[\![\mathcal{M}]\!]$ is an MTTS. If $\mathcal{M}$ is a TPMS then $[\![\mathcal{M}]\!]$ is a *set* of MTTS.

ϕ, ψ ..... State property (linear formula over clocks and locations).

3.2 Modal Specifications for Timing Variability

In this section, we propose a new notion of *time-parametric modal specifications* which can be used to describe the variability of timing and functional behavior in corresponding implementations. We first define *parametric clock constraints* in Section 3.2.1, and present the *syntax* and *operational semantics* of time-parametric modal specifications in Section 3.2.2. We develop the specification theory on *modal refinement*, *property preservation* and *compositional reasoning* in Sections 3.2.3, 3.2.4 and 3.2.5, respectively.

3.2.1 Clocks

As in the classical theory of timed automata [5], we use a finite set of real-valued clock variables, called *clocks* for short, to describe the progress of time. Given a finite set of clocks Clk, we refer to a function $v : Clk \to \mathbb{R}_{\geq 0}$ as a *clock valuation*. Given $d \in \mathbb{R}_{\geq 0}$, let $v + d$ denote the clock valuation that assigns all clocks $x \in Clk$ to $v(x) + d$. The set of all clock valuations is denoted by $\mathbb{R}_{\geq 0}^{Clk}$. We use $\mathbf{0}$ to denote the clock valuation that assigns 0 to all clocks in Clk. Clocks can be reset to zero: for $r \subseteq Clk$, let $v[r \mapsto 0]$ denotes the clock valuation that resets all clocks in

r to 0 and keeps the value $v(x)$ for clocks $x \in \mathit{Clk} \setminus r$. A *clock constraint* is a conjunctive formula of atomic predicates $x \sim c$, where $x \in \mathit{Clk}$ is a clock, $\sim \in \{\leq, <, =, >, \geq\}$ is an equality/inequality relation operator and $c \in \mathbb{N}$ is a constant. A clock valuation v satisfies a clock constraint g, denoted by $v \models g$, iff the proposition formula of g resolves to true when substituting all occurrence of clock x with value $v(x)$.

To specify timing variability, we extend the notion of clock constraints to *parametric clock constraints*, which are conjunctive formulae of predicates $x \sim (c+\alpha)$ where α is a non-negative integer parameter bounded by a set of linear constraints C. Let Θ be a set of parameters, we call a function $f : \Theta \to \mathbb{N}$ a *parameter assignment*. A parameter value $f(\alpha)$ is *valid* if it satisfies the linear constraints C and $c \pm f(\alpha) \in \mathbb{N}$. Let g be a parametric clock constraint; by assigning a set of valid values f to its parameters Θ, we obtain an *instance* of g as a clock constraint, denoted by $g[f(\Theta)]$. For example, a parametric clock constraint $x \leq 1+\alpha$ bounded by $1 \leq \alpha \leq 3, \alpha \in \mathbb{N}$ has three instances: $x \leq 2$, $x \leq 3$ and $x \leq 4$. A clock valuation v satisfies a parametric clock constraint g, denoted by $v \models g$, iff v satisfies *all* instances of g; and we say that v *partially* satisfies g, denoted by $v \models^{\mathsf{p}} g$, iff v satisfies *some* instances of g. Let g_i for $i = 1, 2$ be two parametric clock constraints, each of which is bounded by a set of linear constraints C_i over parameters Θ_i; their conjunction $g = g_1 \wedge g_2$ is then bounded by the linear constraints $C = C_1 \wedge C_2$ over parameters $\Theta = \Theta_1 \cup \Theta_2$.

For the rest of the paper, we use $\mathcal{B}(\mathit{Clk})$ (*resp.* $\mathcal{P}(\mathit{Clk})$) to denote the set of clock constraints over clocks in Clk (*resp.* parametric clock constraints over clocks Clk and parameters in Θ). We have $\mathcal{B}(\mathit{Clk}) \subseteq \mathcal{P}(\mathit{Clk})$, since a clock constraint can be considered as a special case of parametric clock constraints whose

instance is itself.

3.2.2 Syntax and Semantics

Now we propose the notion of *time-parametric modal specifications*, sometimes called *specifications* for short in this paper, to capture the timing and functional variability of different *implementations* modeled as *Input/Output Timed Automata* (IOTA) (timed automaton [33] that distinguishes *input*, *output* and *internal* actions. Our IOTA is comparable to the restricted class of Lynch's Timed I/O Automata (TIOA) whose trajectories can be modeled with what Lynch calls Alur-Dill automata [101]

Formally a IOTA is defined as follows:

Definition 3.2.1 (Input/Output Timed Automata)**.** An *Input/Output Timed Automata* (IOTA) $\mathcal{A}$ is a tuple $(Loc, \bar{l}, Clk, Act, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond})$ where

- Loc is a finite set of *locations*, and $\bar{l} \in Loc$ is an *initial location*
- Clk is a finite set of *clocks*
- $Act = Act^I \uplus Act^O \uplus \tau$ is a finite set of *actions* partitioned into *input* Act^I, *output* Act^O and *internal* Act^τ actions
- $Inv : Loc \rightarrow \mathcal{B}(Clk)$ assigns *invariants* to locations
- $\hookrightarrow_{\Box} \subseteq Loc \times \mathcal{B}(Clk) \times Act \times 2^{Clk} \times Loc$ is the *transition relation* describing the action transitions.

The semantics of a IOTA specification is an infinite transition system (Definition 3.2.2) that models the time-delay and action transitions allowed by the clocks constraints:

Definition 3.2.2 (Timed Transition System)**.** A *timed transition system* (TTS) M is a tuple $(S, \overline{s}, Act, \rightarrow_{\square}, \rightarrow_{\diamond}, \rightarrow_{\mathsf{d}})$ where

- S is an (infinite) set of states, $\overline{s} \in S$ is an initial state, and Act is a set of actions
- $\rightarrow_{\square} \subseteq S \times Act \times S$ is the *action* transition relation, and $\rightarrow_{\mathsf{d}} \subseteq S \times \mathbb{R}_{\geq 0} \times S$ is the *delay* transition relation

Definition 3.2.3 (Semantics of IOTA)**.** The semantics of an I/O timed automaton $\mathcal{A} = (Loc, \bar{l}, Clk, Act, Inv, \hookrightarrow_{\square}, \hookrightarrow_{\diamond})$, written $[\![\mathcal{A}]\!]$, is a *timed transition system (TTS)* represented as a tuple $(S, \overline{s}, \Sigma, \rightarrow)$ where

- $S = \{\langle l, v\rangle \in Loc \times \mathbb{R}_{\geq 0}^{Clk} \mid v \models Inv(l)\}$ is an infinite set of states
- $\overline{s} = \langle \bar{l}, \mathbf{0}\rangle$ is an initial state
- $\Sigma = Act \cup \mathbb{R}_{\geq 0}$ is the alphabet
- $\langle l, v\rangle \xrightarrow{a}_{\square} \langle l', v'\rangle$ if there is an action transition $l \overset{g,a,r}{\hookrightarrow}_{\square} l'$ in $\mathcal{A}$ such that $v \models g$, $v' = v[r \mapsto 0]$ (each clock in r is reset to 0) and $v' \models Inv(l')$
- $\langle l, v\rangle \xrightarrow{d}_{\mathsf{d}} \langle l, v + d\rangle$ for $d \in \mathbb{R}_{\geq 0}$ if $v + d \models Inv(l)$

A transition $l \overset{g,a,r}{\hookrightarrow} l'$ is enabled at location l when guard $g \in \mathcal{B}(Clk)$ holds and action $a \in Act$ occurs; any clock in $r \subseteq Clk$ will be reset to 0 once the transition has been taken and the new location is l'.

IOTA can be extended to capture *functional* variability by distinguishing between *may* and *must* transitions [183, 50]. May-transitions represent *allowed* behavior, while must-transitions capture the required behavior. This extension of IOTA are called *modal* IOTA (MIOTA). Formally, MIOTA are defined as follows:

Definition 3.2.4 (Modal I/O Timed Automaton)**.** A *modal I/O timed automaton* (MIOTA) $\mathcal{A}$ is a tuple $(Loc, \bar{l}, Clk, Act, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond})$ where

- Loc is a finite set of *locations*, and $\bar{l} \in Loc$ is an *initial location*
- Clk is a finite set of *clocks*
- $Act = Act^I \uplus Act^O \uplus \tau$ is a finite set of *actions* partitioned into *input* Act^I, *output* Act^O and *internal* Act^τ actions
- $Inv : Loc \to \mathcal{B}(Clk)$ assigns *invariants* to locations.
- $\hookrightarrow_{\Box} \subseteq Loc \times \mathcal{B}(Clk) \times Act \times 2^{Clk} \times Loc$ is the *must transition relation* describing required behavior
- $\hookrightarrow_{\Diamond} \subseteq Loc \times \mathcal{B}(Clk) \times Act \times 2^{Clk} \times Loc$ is the *may transition relation* describing allowed behavior. We require that $\hookrightarrow_{\Box} \subseteq \hookrightarrow_{\Diamond}$

We say an MIOTA

Definition 3.2.5 (Implementation)**.** We say that $\mathcal{A}$ is an *implementation* (*i.e.,* IOTA), if there is no variability, *i.e.,* $\hookrightarrow_{\Box} = \hookrightarrow_{\Diamond}$.

The operational semantics of MIOTA are then given by *modal* Timed Transition Systems (*i.e.,* MTTS), which likewise extend TTS with may/must distinction on transitions. The transitions of the MTTS inherit their modality from the modality of the edge that induces them in the MIOTA:

Definition 3.2.6 (Modal Timed Transition System)**.** A *modal timed transition system* (MTTS) M is a tuple $(S, \bar{s}, Act, \to_{\Box}, \to_{\Diamond}, \to_{\mathsf{d}})$ where

- S is an (infinite) set of states, $\overline{s} \in S$ is an initial state, and Act is a set of actions

- $\rightarrow_{\Box} \subseteq S \times Act \times S$ is the *must action* transition relation, $\rightarrow_{\Diamond} \subseteq S \times Act \times S$ is the *may action* transition relation, and $\rightarrow_{\mathsf{d}} \subseteq S \times \mathbb{R}_{\geq 0} \times S$ is the *delay* transition relation

Definition 3.2.7 (Operational Semantics of MIOTA)**.** The operational semantics of a modal timed I/O automaton $\mathcal{A} = (Loc, \bar{l}, Clk, Act, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond})$, denoted by $[\![\mathcal{A}]\!]$, is a *modal timed transition system (MTTS)* represented as a tuple $(S, \overline{s}, \Sigma, \rightarrow$ $)$ where

- $S = \{\langle l, v\rangle \in Loc \times \mathbb{R}_{\geq 0}^{Clk} \mid v \models Inv(l)\}$ is an infinite set of states

- $\overline{s} = \langle \bar{l}, \mathbf{0}\rangle$ is an initial state

- $\Sigma = Act \cup \mathbb{R}_{\geq 0}$ is the alphabet

- $\langle l, v\rangle \xrightarrow{a}_{\Box} \langle l', v'\rangle$ (*resp.* $\langle l, v\rangle \xrightarrow{a}_{\Diamond} \langle l', v'\rangle$) if there is a must (*resp.* may) action transition $l \overset{g,a,r}{\hookrightarrow}_{\Box} l'$ (*resp.* $l \overset{g,a,r}{\hookrightarrow}_{\Diamond} l'$) in $\mathcal{M}$ such that $v \models g$, $v' = v[r \mapsto 0]$ and $v' \models Inv(l')$

- $\langle l, v\rangle \xrightarrow{d}_{\mathsf{d}} \langle l, v + d\rangle$ for $d \in \mathbb{R}_{\geq 0}$ if $v + d \models Inv(l)$

We extend MIOTA to support *timing* variability by allowing natural number valued parameters in the clocks contraints. We call this extension Time Parametric Modal Specifications. Syntactically, a time-parametric modal specification looks similar to a MIOTA; it distinguishes *must* and *may* transition relations for functional variability, and also allows parametric clock constraints for timing variability:

Definition 3.2.8 (Time-Parametric Modal Specification)**.** A *time-parametric modal specification* (TPMS) $\mathcal{M}$ is a tuple $(Loc, \bar{l}, Clk, \Theta, Act, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond}, C(\Theta))$ where

- $Loc, \bar{l}$, Clk and Act are the same as in Definition 3.2.4
- Θ is a set of parameters that can take on values from $\mathbb{N}$.
- $Inv : Loc \rightarrow \mathcal{P}(Clk)$ assigns parametric clock constraints in the form of $x \leq c \pm \alpha$ or $x < c \pm \alpha$ to locations, where $c, \alpha \in \mathbb{N}$
- $\hookrightarrow_{\Box} \subseteq Loc \times \mathcal{P}(Clk) \times Act \times 2^{Clk} \times Loc$ is the *must transition relation* describing required behavior
- $\hookrightarrow_{\Diamond} \subseteq Loc \times \mathcal{P}(Clk) \times Act \times 2^{Clk} \times Loc$ is the *may transition relation* describing allowed behavior
- $C(\Theta)$ is a set of linear constraints on a finite set of non-negative integer parameters Θ that are used in $\mathcal{P}(Clk)$

We consider only *consistent* specifications where $\hookrightarrow_{\Box} \subseteq \hookrightarrow_{\Diamond}$, *i.e.,* a required transition should also be allowed. Definition 3.2.8 coincides with Definition 3.2.4 if $\hookrightarrow_{\Box} = \hookrightarrow_{\Diamond}$, $\mathcal{P}(Clk) = \mathcal{B}(Clk)$ and $\Theta = \emptyset$.

Intuitively, we can see that a time-parametric modal specification is a succint way to specify a set of MIOTA; one for each valid valuation of the parameters.

Example Figure 3.3 shows a time-parametric modal specification for an infusion pump, which has 4 locations: **detect**, **disabled**, **start** and **infusion**. The initial location **detect** is indicated with an incoming arrow. There is only one clock variable x. The actions include input ("bolus?", "off_pump?") and output ("alarm!", "pump_ison!", "pump_isoff!"). There are three parameters $\Theta =$

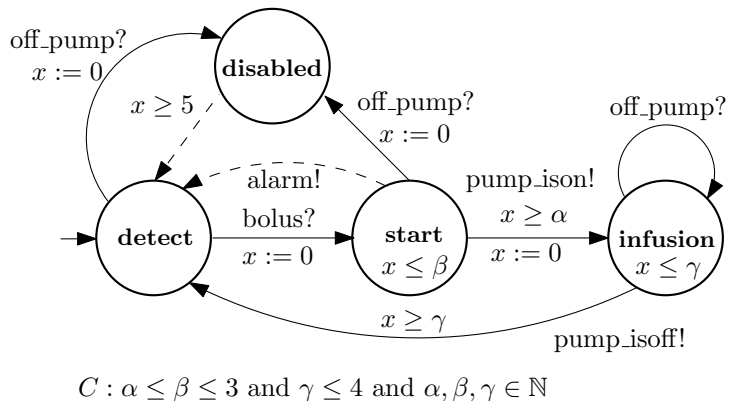


Figure 3.3: An example of time-parametric modal specification.

$\{\alpha, \beta, \gamma\}$, bounded by the linear constraints $\alpha \leq \beta \leq 3 \wedge \gamma \leq 4$ and $\alpha, \beta, \gamma \in \mathbb{N}$. The invariant on **detect** and **disabled** is true (omitted in the figure), while the invariants on **start** and **infusion** are the parametric clock constraints $x \leq \beta$ and $x \leq \gamma$, respectively. Must (*resp.* may) transitions are indicated by solid (*resp.* dashed) lines in the figure. For example, once a pump detects a "bolus?" request, it *must* move to the **start** location. If the pump is in **detect** or **start** it will be disabled if it receives a "off_pump" signal. Implementations are allowed to stay disabled or return to **detect** after 5 units of time has elapsed.

Then, the semantics of TPMS can be defined in terms of the semantics of MIOTA:

Definition 3.2.9 (Semantics of TPMS)**.** For a time-parametric modal specification $\mathcal{M} = (Loc, \bar{l}, Clk, Act, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond}, C(\Theta))$, its semantics, denoted by $[\![\mathcal{M}]\!]$, is a finite set of MTTSs such that there is a one-to-one mapping between each valid parameter assignment $f(\Theta)$, an MIOTA and its MTTS $M = (S, \overline{s}, Act, \rightarrow_{\Box}, \rightarrow_{\Diamond}, \rightarrow_{\mathsf{d}})$.

3.2.3 Modal Refinement

We say that a MIOTA satisfies a TPMS if the MIOTA *refines* the TPMS. Before we can formally define refinement between a MIOTA and a TPMS we need to define refinement between two MIOTA. We say that a MIOTA $\mathcal{M}_1$ refines another MITOA $\mathcal{M}_2$ if the operational semantic MTTS of $\mathcal{M}_1$ refines the operational semantic MTTS of $\mathcal{M}_2$:

Definition 3.2.10 (Modal Refinement of MTTSs)**.** Let $M_i = (S_i, \overline{s}_i, Act, \rightarrow_{\Box,i}, \rightarrow_{\Diamond,i}, \rightarrow_{\mathsf{d},i})$ for $i = 1, 2$ be two MTTSs. We say that M_1 *modally refines* M_2, denoted by $M_1 \preceq M_2$, iff there exists a binary relation $R \subseteq S_1 \times S_2$ containing $(\overline{s}_1, \overline{s}_2)$ such that for each $(s, t) \in R$ we have

- for all $(t, a, t') \in \rightarrow_{\Box,2}$ there is some $(s, a, s') \in \rightarrow_{\Box,1}$ with $(s', t') \in R$
- for all $(s, a, s') \in \rightarrow_{\Diamond,1}$ there is some $(t, a, t') \in \rightarrow_{\Diamond,2}$ with $(s', t') \in R$
- for all $(s, d, s') \in \rightarrow_{\mathsf{d},1}$ there is some $(t, d, t') \in \rightarrow_{\mathsf{d},2}$ with $(s', t') \in R$, and for all $(t, d, t') \in \rightarrow_{\mathsf{d},2}$ there is some $(s, d, s') \in \rightarrow_{\mathsf{d},1}$ with $(s', t') \in R$

Thus, $\mathcal{M}_1$ must exhibit all the required behavior of $\mathcal{M}_1$, and $\mathcal{M}_2$ must exhibit all the allowed behavior of $\mathcal{M}_1$. Since a TTS (Definition 3.2.7) can be considered as a special case of MTTSs where $\rightarrow_{\Box} = \rightarrow_{\Diamond}$, the above definition is also applicable for $A \preceq M$ where A is a TTS and M is a MTTS.

Definition 3.2.11 (TPMS Refinement)**.** Let $\mathcal{A}$ be a MIOTA and $\mathcal{M}$ be a TPMS over the same actions. We say that $\mathcal{A}$ refines $\mathcal{M}$, denoted by $\mathcal{A} \sqsubseteq \mathcal{M}$, if there exist a MTTS $M \in [\![\mathcal{M}]\!]$ such that $[\![\mathcal{A}]\!] \preceq M$. If $\mathcal{A}$ is an implementation, then we say that $\mathcal{A}$ *implements* $\mathcal{M}$.

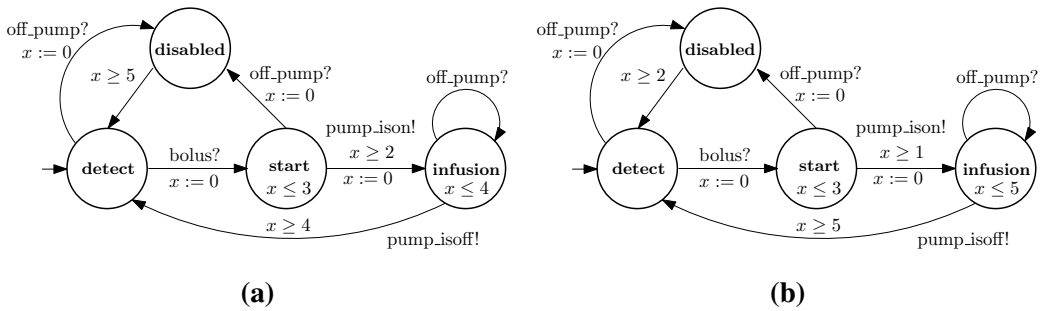


Figure 3.4: Two example MIOTAs.

A time-parametric modal specification $\mathcal{M}$ may admit a set of implementations with timing and functional variability. By fixing a parameter assignment for the specification, a MTTS $M \in [\![\mathcal{M}]\!]$ representing certain timing behavior is chosen; and by checking whether the operational semantics of $\mathcal{A}$ modally refines M, we compare their functional behavior. $\mathcal{A}$ is an implementation of $\mathcal{M}$ when both the timing and functional requirements are met. Later in Section 3.3.5, we present a symbolic method to check the refinement relation.

Example Consider the time-parametric modal specification shown in Figure 3.3. The MIOTA in Figure 3.4a is an implementation of the specification, where the may transition from **disabled** to **detect** is implemented and the parameters are fixed for $\alpha = 2, \beta = 3, \gamma = 4$. The MIOTA in Figure 3.4b is not a valid implementation for two reasons. First, both the specification and implementation can reach the location **infusion** with $x = 0$. The system in Figure 3.4b can then delay up to 5 timeunits while in location **infusion** (the invariant is $x \leq 5$), but the specification only allows for a delay of γ timeunits with $\gamma \leq 4$. Second, the transition from **disabled** to **detect** becomes disabled after only 2 units of time has elapsed.

Often we only care about the "internal" behavior of a component; we don't

care *how* a component reaches some externally visible behavior only that it does. Like many other types of modal specifications we provide a notion of *weak* refinement and implementation which hides the internal computation of a component. Before we can proceed to weak-refinement (and implementation) we need to define weak action transitions and weak delay transitions. Unlike normal delay transitions, weak-delay transitions have a (may-must) modality. We write $\overset{d}{\Rightarrow}_{\Box\mathsf{d}}$ for the set of must-delay transitions and $\overset{d}{\Rightarrow}_{\Diamond\mathsf{d}}$ for the set of weak-delay transitions. This distinction is required because what is externally observed as a delay transition may internally involve τ steps, which are either may or must.

Definition 3.2.12 (Weak Transitions)**.** The weak must (*resp.* may) action transition $\overset{a}{\Rightarrow}_{\Box}$ (*resp.* $\overset{a}{\Rightarrow}_{\Diamond}$) and weak must (*resp.* may-delay) delay $\overset{d}{\Rightarrow}_{\Box\mathsf{d}}$ (*resp.* $\overset{d}{\Rightarrow}_{\Diamond\mathsf{d}}$) are defined via the following rules:

$$\frac{s \xrightarrow{a}_{\Box} s' \quad a \in Act}{s \overset{a}{\Rightarrow}_{\Box} s'} \qquad \frac{s \xrightarrow{a}_{\Diamond} s' \quad a \in Act}{s \overset{a}{\Rightarrow}_{\Diamond} s'}$$

$$\frac{s \xrightarrow{\tau^n}_{\Box} s', \quad n \in \mathbb{N}}{s \overset{0}{\Rightarrow}_{\Box\mathsf{d}} s'} \qquad \frac{s \xrightarrow{\tau^n}_{\Diamond} s', \quad n \in \mathbb{N}}{s \overset{0}{\Rightarrow}_{\Diamond\mathsf{d}} s'}$$

$$\frac{s \xrightarrow{d}_{\Box} s'}{s \overset{d}{\Rightarrow}_{\Box\mathsf{d}} s'} \qquad \frac{s \xrightarrow{d}_{\Diamond} s'}{s \overset{d}{\Rightarrow}_{\Diamond\mathsf{d}} s'}$$

$$\frac{s \overset{d_1}{\Rightarrow}_{\Box\mathsf{d}} s', \; s' \overset{d_2}{\Rightarrow}_{\Box\mathsf{d}} s''}{s \overset{d_1+d_2}{\Longrightarrow}_{\Box\mathsf{d}} s''} \qquad \frac{s \overset{d_1}{\Rightarrow}_{\Diamond\mathsf{d}} s', \; s' \overset{d_2}{\Rightarrow}_{\Diamond\mathsf{d}} s''}{s \overset{d_1+d_2}{\Longrightarrow}_{\Diamond\mathsf{d}} s''}$$

In other words, any number of consecutive τ transitions are observed as a $0-$delay transition. If one or more of the τ transitions in the sequence are not *must*-transitions then then the delay transition is a "may" delay-transition.

Armed with a definition for weak-transitions we can now relax Definition 3.2.10 into a defintion for weak-refinement of MTTSs:

Definition 3.2.13 (Weak Modal Refinement of MTTSs)**.** Let $M_i = (S_i, \overline{s}_i, Act, \rightarrow_{\Box,i}, \rightarrow_{\Diamond,i}, \rightarrow_{\mathsf{d},i})$ for $i = 1, 2$ be two MTTSs. We say that M_1 *weakly modally refines* M_2, denoted by $M_1 \precsim M_2$, iff there exists a binary relation $R \subseteq S_1 \times S_2$ containing $(\overline{s}_1, \overline{s}_2)$ such that for each $(s, t) \in R$ we have

- for all $(t, a, t') \in \Rightarrow_{\Box,2}$ there is some $(s, a, s') \in \Rightarrow_{\Box,1}$ with $(s', t') \in R$
- for all $(s, a, s') \in \Rightarrow_{\Diamond,1}$ there is some $(t, a, t') \in \Rightarrow_{\Diamond,2}$ with $(s', t') \in R$
- for all $(s, d, s') \in \Rightarrow_{\mathsf{d}\Box,2}$ there is some $(t, d, t') \in \Rightarrow_{\mathsf{d}\Box,1}$ with $(s', t') \in R$
- for all $(s, d, s') \in \Rightarrow_{\mathsf{d}\Diamond,1}$ there is some $(t, d, t') \in \Rightarrow_{\mathsf{d}\Diamond,2}$ with $(s', t') \in R$

Note that the differences between strong (Definition 3.2.10) and weak refinement are the replacement of the transition relation with the weak version and an extra rule to account for the existence of may-delay transitions.

Definition 3.2.14 (Weak Implementation)**.** Let $\mathcal{A}$ be a MIOTA and $\mathcal{M}$ be a time-parametric modal specification over the same actions. We say that $\mathcal{A}$ is a *weak implementation* of $\mathcal{M}$, denoted by $\mathcal{A} \sqsubseteq\!\!\!\sim \mathcal{M}$, if there exist a MTTS $M \in [\![\mathcal{M}]\!]$ such that $[\![\mathcal{A}]\!] \precsim M$.

It is also clear that implementation implies weak implementation:

Lemma 3.2.1 (Implementation implies Weak Implementation)**.** Let $\mathcal{A}$ be a MIOTA and $\mathcal{M}$ be a TPMS. Then $\mathcal{A} \sqsubseteq \mathcal{M}$ implies $\mathcal{A} \sqsubseteq\!\!\!\sim \mathcal{M}$.

3.2.4 Property Preservation

Property preservation (*i.e.,* the satisfaction of certain property on a specification implies that all implementations also satisfy the property) is crucial for any specification theory because it allows designers to predict how systems will behave by

analyzing their specifications. In this section, we prove that the TPMS implementation relation preserves *state-reachability*, *quiescence-freedom*, and *timelock-freedom*.

Reachability

Let ϕ be a linear formulae over the locations and clocks of some specification $\mathcal{M}$. We say that ϕ is reachable in $\mathcal{M}$ if there is some path from $\mathcal{M}$'s initial state to a state satisfying ϕ. Reachability checking is crucial for safety analysis: If we can define a set of "bad" or "unsafe" states we want to ensure that those states cannot be reached in *any* implementation (*i.e.,* that it is not *may-reachability*). Conversely, we may also want to determine if all implementations are able to reach some useful state (*i.e.,* the state is *must-reachable*).

Before we can reason about reachability preservation we need to formally define *may* and *must* reachability:

Definition 3.2.15 (May-reachability)**.** Let $\mathcal{M}$ be a TPMS and ϕ be a formula over locations and clock values (*i.e.,* states). Then if there is some path from $\mathcal{M}$'s initial state to a state satisfying ϕ consisting of may-transitions in at least one MTTS $M \in [\![\mathcal{M}]\!]$ then we say that ϕ is may-reachable in $\mathcal{M}$ (written $\mathcal{M} \models_{\diamond} \mathsf{EF}\ \phi$).

Definition 3.2.16 (Must-reachability)**.** Let $\mathcal{M}$ be a TPMS and ϕ be a formula over locations and clock values (*i.e.,* states). Then if there is some path from $\mathcal{M}$'s initial state to a state satisfying ϕ consisting of must-transitions in all MTTS $M \in [\![\mathcal{M}]\!]$ then we say that ϕ is must-reachable in $\mathcal{M}$ (written $\mathcal{M} \models_{\Box} \mathsf{EF}\ \phi$).

Note that $\mathcal{M} \models_{\Box} \mathsf{EF}\ \phi$ implies $\mathcal{M} \models_{\diamond} \mathsf{EF}\ \phi$ because all must transitions are also may transitions.

When it comes to safety analysis we generally want to know that if the specification *cannot* reach some bad state then neither can any of its implementations. The TPMS refinement relation preserves negative may-reachability:

Lemma 3.2.2. Let A be a MTTS and M be a MTTS such that $A \preceq M$. Let ψ be a formula over the states of M. Suppose $M \not\models_{\diamond} \mathsf{EF}\ \psi$, then $A \not\models_{\diamond} \mathsf{EF}\ \psi$.

Proof. Because $M \not\models_{\diamond} \mathsf{EF}\ \psi$ we know that there is no path consisting of may or delay transitions in M from M's starting state through the states of M that passes through some state s such that $s \models \phi$. By definition of refinement of MTTS (Definition 3.2.10) we know that A has a subset of the may and delay transitions from M. We can conclude that there is no path consisting of may or delay transitions in A from A's starting state through the states of A that passes through some state s such that $s \models \phi$. □

Theorem 3.2.3. Let $\mathcal{A}$ be a MIOTA implementation of a time-parametric modal specification $\mathcal{M}$, *i.e.,* $\mathcal{A} \sqsubseteq \mathcal{M}$. Let ψ be a formula over the states of $\mathcal{M}$. Suppose $\mathcal{M} \not\models_{\diamond} \mathsf{EF}\ \psi$, then $\mathcal{A} \not\models_{\diamond} \mathsf{EF}\ \psi$.

Proof. Since $\mathcal{A} \sqsubseteq \mathcal{M}$, based on Definition 3.2.14, there must exist a MTTS $M \in [\![\mathcal{M}]\!]$ such that $[\![\mathcal{A}]\!] \preceq M$. Given that $\mathcal{M} \not\models_{\diamond} \mathsf{EF}\ \psi$, we know that $M \not\models_{\diamond} \mathsf{EF}\ \psi$ for every $M \in [\![\mathcal{M}]\!]$. Thus, based on Lemma 3.2.2, we have that $\mathcal{A} \not\models_{\diamond} \mathsf{EF}\ \psi$. □

Sometimes one wants to know if a state can be reached in all implementations. The implementation relation preserves positive must-reachability:

Lemma 3.2.4. Let A be a MTTS and M be a MTTS such that $A \preceq M$. Let ψ be a formula over the states of M. Suppose $M \models_{\Box} \mathsf{EF}\ \psi$, then $A \models_{\Box} \mathsf{EF}\ \psi$.

Proof. Because $M \models_{\Box} \mathsf{EF}\ \psi$ we know that there is a path π consisting of must and delay transitions in M from M's starting state through the states of M that passes through some state s such that $s \models \phi$. By definition of refinement of MTTS (Definition 3.2.10) we know that A has all the must and delay transitions of M. Therefore, we conclude that there is a path π consisting must and delay transitions in A from A's starting state through the states of A that passes through some state s such that $s \models \phi$. $\Box$

Theorem 3.2.5. Let $\mathcal{A}$ be a MIOTA implementation of a time-parametric modal specification $\mathcal{M}$, *i.e.,* $\mathcal{A} \sqsubseteq \mathcal{M}$. Let ψ be a formula over the states of $\mathcal{M}$. Suppose $\mathcal{M} \models_{\Box} \mathsf{EF}\ \psi$, then $\mathcal{A} \models_{\Box} \mathsf{EF}\ \psi$.

Proof. Since $\mathcal{A} \sqsubseteq \mathcal{M}$, based on Definition 3.2.14, there must exist a MTTS $M \in [\![\mathcal{M}]\!]$ such that $[\![\mathcal{A}]\!] \preceq M$. Given that $\mathcal{M} \models_{\Box} \mathsf{EF}\ \psi$, we know that $M \models_{\Box} \mathsf{EF}\ \psi$ for every $M \in [\![\mathcal{M}]\!]$. Thus, based on Lemma 3.2.4, we have that $\mathcal{A} \models_{\Box} \mathsf{EF}\ \psi$. $\Box$

Quiescence Freedom

Informally, a state is *quiescent* if it is "inactive", *i.e.,* a system is quiescent if it will not engage in any actions in the future. We can think of quiescence as a deadlock-like state where time is still allowed to pass. The implementation relation preserves negative reachability of quiescent states.

Definition 3.2.17 (Quiescent States)**.** Let s be some state of TPMS $\mathcal{M}$. We say that s is may-quiescent (written $s \models_{\Diamond} \Omega$) if all the paths starting at s consist of only delay transitions and may transitions.

Lemma 3.2.6. Let A be a MTTS and M be a MTTS such that $A \preceq M$. Let Ω be the quiescence property. Suppose $M \not\models_{\Diamond} \mathsf{EF}\ \Omega$, then $A \not\models_{\Diamond} \mathsf{EF}\ \Omega$.

Proof. Because $M \not\models_\diamond \mathsf{EF}\ \Omega$ we know that there is no reachable state s in M such that $s \models_\diamond \Omega$. There are two cases to consider:

- Case 1: There is some state s in A which is may-quiescent. Then by Theorem 3.2.3 we know it is not reachable.
- Case 2: There is no state which is may-quiescent in M. This means that for all states s in M there are paths starting at s consisting of both delays and must-transitions. By definition A must include all those transitions and therefore all those paths. We conclude that there are no may-quiescent states in A.

□

Theorem 3.2.7. Let $\mathcal{A}$ be a MIOTA implementation of a time-parametric modal specification $\mathcal{M}$, *i.e.,* $\mathcal{A} \sqsubseteq \mathcal{M}$. Let Ω be the quiescence property. Suppose $\mathcal{M} \not\models_\diamond \mathsf{EF}\ \Omega$, then $\mathcal{A} \not\models_\diamond \mathsf{EF}\ \Omega$.

Proof. Since $\mathcal{A} \sqsubseteq \mathcal{M}$, based on Definition 3.2.14, there must exist a MTTS $M \in [\![\mathcal{M}]\!]$ such that $[\![\mathcal{A}]\!] \preceq M$. Given that $\mathcal{M} \not\models_\diamond \mathsf{EF}\ \Omega$, we know that $M \not\models_\diamond \mathsf{EF}\ \Omega$ for every $M \in [\![\mathcal{M}]\!]$. Thus, based on Lemma 3.2.6, we have that $\mathcal{A} \not\models_\diamond \mathsf{EF}\ \Omega$. □

Timelock Freedom

A state is timelocked if time cannot progress. It is possible to express a TPMS where some implementation has a timelocked state. Like quiescence-freedom, the implementation relation preserves negative reachability of timelocked states.

Definition 3.2.18 (Timedlocked States)**.** Let s be some state of TPMS $\mathcal{M}$. We say that s is may-timelocked (written $s \models_\diamond \mathsf{EF}\ \Gamma$) if all the paths starting at s consist only of may transitions or the 0-delay transition.

Lemma 3.2.8. Let A be a MTTS and M be a MTTS such that $A \preceq M$. Let Γ be the timelock property. Suppose $M \not\models_{\diamond} \mathsf{EF}\ \Gamma$, then $A \not\models_{\diamond} \mathsf{EF}\ \Gamma$.

Proof. Because $M \not\models_{\diamond} \mathsf{EF}\ \Gamma$ we know that there is no reachable state s of M such that $s \models_{\diamond} \mathsf{EF}\ \Gamma$. There are two cases to consider:

- Case 1: There is some state s in A which is has may-timelocked. Then by Theorem 3.2.3 we know it is not reachable.
- Case 2: There is no state which has may-timelock in M. This means that for all states s in M there are paths starting at s consisting either of delay transitions such that $d \geq 0$ or must-transitions. By definition A must include all those transitions and therefore all those paths. We conclude that there are no may-timelocked states in A.

□

Theorem 3.2.9. Let $\mathcal{A}$ be a MIOTA implementation of a time-parametric modal specification $\mathcal{M}$, *i.e.,* $\mathcal{A} \sqsubseteq \mathcal{M}$. Let Γ be the timelock property. Suppose $\mathcal{M} \not\models_{\diamond} \mathsf{EF}\ \Gamma$, then $\mathcal{A} \not\models_{\diamond} \mathsf{EF}\ \Gamma$.

Proof. Since $\mathcal{A} \sqsubseteq \mathcal{M}$, based on Definition 3.2.14, there must exist a MTTS $M \in [\![\mathcal{M}]\!]$ such that $[\![\mathcal{A}]\!] \preceq M$. Given that $\mathcal{M} \not\models_{\diamond} \mathsf{EF}\ \Gamma$, we know that $M \not\models_{\diamond} \mathsf{EF}\ \Gamma$ for every $M \in [\![\mathcal{M}]\!]$. Thus, based on Lemma 3.2.8, we have that $\mathcal{A} \not\models_{\diamond} \mathsf{EF}\ \Gamma$. □

The theorems in this subsection also hold for weak implementation. The structure of the proofs do not change, one only has to reason over weak paths in the place of strong paths.

3.2.5 Compositional Reasoning

We now introduce the notion of *composition*, which is important for component-based design. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two time-parametric modal specifications. They are *composeable* iff they have disjoint sets of clocks and parameters, *i.e.,* $Clk_1 \cap Clk_2 = \emptyset$ and $\Theta_1 \cap \Theta_2 = \emptyset$, and their actions only overlap on complementary types: $(Act_1^I \cup Act_1^\tau) \cap (Act_2^I \cup Act_2^\tau) = \emptyset$ and $(Act_1^O \cup Act_1^\tau) \cap (Act_2^O \cup Act_2^\tau) = \emptyset$.

Definition 3.2.19 (Composition)**.** Given two *composeable* time-parametric modal specifications $\mathcal{M}_i = (Loc_i, \bar{l}_i, Clk_i, Act_i, Inv_i, \hookrightarrow_{\Box,i}, \hookrightarrow_{\Diamond,i}, C_i(\Theta_i))$ for $i = 1, 2$. Their *composition* product, denoted by $\mathcal{M}_1 \| \mathcal{M}_2$, yields a specification $(Loc_1 \times Loc_2, (\bar{l}_1, \bar{l}_2), Clk_1 \cup Clk_2, Act, Inv, \hookrightarrow_\Box, \hookrightarrow_\Diamond, C(\Theta))$ such that

- $Act = Act^I \uplus Act^O \uplus Act^\tau$ where $Act^I = (Act_1^I \setminus Act_2^O) \cup (Act_2^I \setminus Act_1^O)$, $Act^O = (Act_1^O \setminus Act_2^I) \cup (Act_2^O \setminus Act_1^I)$

- $Inv(l_1, l_2) = Inv_1(l_1) \wedge Inv_2(l_2)$

- $\hookrightarrow_\Box$ and $\hookrightarrow_\Diamond$ are defined by the following rules (interchangeable for $\mathcal{M}_1$ and $\mathcal{M}_2$):

$$\frac{(l_1, g_1, a!, r_1, l_1') \in \hookrightarrow_{\gamma,1} \quad (l_2, g_2, a?, r_2, l_2') \in \hookrightarrow_{\gamma,2}}{\big((l_1, l_2), g_1 \wedge g_2, \tau, r_1 \cup r_2, (l_1', l_2')\big) \in \hookrightarrow_\gamma} \textit{ (synchronizing)}$$

$$\frac{(l_1, g_1, a, r_1, l_1') \in \hookrightarrow_{\gamma,1} \quad a \notin Act_2 \vee a = \tau}{\big((l_1, l_2), g_1, a, r_1, (l_1', l_2)\big) \in \hookrightarrow_\gamma} \textit{ (interleaving)}$$

  where $\gamma \in \{\Box, \Diamond\}$: if $\hookrightarrow_{\gamma,1} = \hookrightarrow_{\Box,1}$ and $\hookrightarrow_{\gamma,2} = \hookrightarrow_{\Box,2}$ in the synchronizing rule, or $\hookrightarrow_{\gamma,1} = \hookrightarrow_{\Box,1}$ in the interleaving rule, then $\hookrightarrow_\gamma = \hookrightarrow_\Box$; otherwise, $\hookrightarrow_\gamma = \hookrightarrow_\Diamond$.

- $\Theta = \Theta_1 \cup \Theta_2$ and $C(\Theta) = C_1(\Theta_1) \wedge C_2(\Theta_2)$.

We also define composition at the semantic level for MTTS:

Definition 3.2.20 (Composition of MTTS)**.** Given two MTTS $M_i = (S_i, \overline{s}_i, Act_i, \rightarrow^i_\Box, \rightarrow^i_\Diamond, \rightarrow^i_{\mathsf{d}})$ for $i = 1, 2$. Their *composition* product, denoted by $\mathcal{M}_1 \| \mathcal{M}_2$, yields a specification $(S_1 \times S_2, (\overline{s}_1, \overline{s}_2), Act_1 \cup Act_2, \rightarrow_\Box, \rightarrow_\Diamond, \rightarrow_{\mathsf{d}})$ such that

- $\rightarrow_\Box$, $\rightarrow_\Diamond$ and $\rightarrow_{\mathsf{d}}$ are defined by the following rules (interchangeable for M_1 and M_2):

$$\frac{(s_1, a!, s_1') \in \rightarrow^1_\gamma \quad (s_2, a?, s_2') \in \rightarrow^2_\gamma}{\big((s_1, s_2)), \tau, (s_1', s_2')\big) \in \rightarrow_\gamma} \textit{ (action synchronizing)}$$

$$\frac{(s_1, d, s_1 + d) \in \rightarrow^1_{\mathsf{d}} \quad (s_2, d, s_2 + d) \in \rightarrow^2_{\mathsf{d}}}{\big((s_1, s_2)), d, (s_1, s_2) + d\big) \in \rightarrow_{\mathsf{d}}} \textit{ (delay synchronizing)}$$

$$\frac{(s_1, a, s_1') \in \rightarrow^1_\gamma \quad a \notin Act_2 \vee a = \tau}{\big((s_1, s_2), a, (s_1', s_2)\big) \in \rightarrow_\gamma} \textit{ (interleaving)}$$

  where $\gamma \in \{\Box, \Diamond\}$: if $\rightarrow^1_\gamma = \rightarrow^1_\Box$ and $\rightarrow^2_\gamma = \rightarrow^2_\Box$ in the action synchronizing rule, or $\rightarrow^1_\gamma = \rightarrow^1_\Box$ in the interleaving rule, then $\rightarrow_\gamma = \rightarrow_\Box$; otherwise, $\rightarrow_\gamma = \rightarrow_\Diamond$.

Since a MIOTA can be considered as a special case of time-parametric modal specifications, the above definition is also applicable for the composition of two MIOTAs (*resp.* a MIOTA and a TPMS), which yields a product MIOTA (*resp.* TPMS). Compositional refinement is important for component based reasoning because it enables top-down reasoning with component specifications: Let $\mathcal{X}$ and $\mathcal{Y}$ be two TPMS. Compositional refinement means that if $\mathcal{X}_1 \sqsubseteq \mathcal{X}$ then $\mathcal{X}_1 \| \mathcal{Y} \sqsubseteq \mathcal{X} \| \mathcal{Y}$. TPMS refinement is compositional. We conclude this section by proving the compositionality of TPMS refinement and that compositionality preserves reachability properties.

First we show that composition of syntax is equivalent to composition of MTTS:

Lemma 3.2.10 (Equivalence of Syntactic and Semantic Composition)**.** Let $\mathcal{X}_1$ and $\mathcal{X}_2$ be two MIOTA. Then $[\![\mathcal{X}_1]\!] \| [\![\mathcal{X}_2]\!] = [\![\mathcal{X} \| \mathcal{X}_2]\!]$.

Proof. It suffices to show that the transitions of $[\![\mathcal{X}_1]\!] \| [\![\mathcal{X}_2]\!]$ are subsets of the transitions of $[\![\mathcal{X} \| \mathcal{X}_2]\!]$ and vice-versa. We prove both directions of the equality.

Case: Left to right . There are two subcases to consider, one for the action transitions and one for the delay transitions. (We do not need to distinguish between may and must. The arguments the follow can be instantiated to apply to either may or must transitions by substituting $\Diamond$ or $\Box$ for γ).

Case: action transitions . Assume towards a contradiction that there is some $((s_1, s_2), a, (s_1', s_2'))$ in the action transitions of $[\![\mathcal{X}_1]\!] \| [\![\mathcal{X}_2]\!]$ but not in the action transitions of $[\![\mathcal{X}_1 \| \mathcal{X}_2]\!]$. W.o.l.o.g. we only consider the action synchronizing case. By Definition 3.2.20 there must be some $(s_1, a, s_1') \in \rightarrow_\gamma^1$ and $(s_2, a, s_2') \in \rightarrow_\gamma^2$. But by Definition 3.2.7 there must be some transition $(l_1, g_1, a, r_1, l_1') \in \hookrightarrow_{\gamma,1}$ and $(l_2, g_2, a, r_2, l_2') \in \hookrightarrow_{\gamma,2}$ where $s_1 \models g_1 \wedge s_2 \models g_2$ and s_1' is the same as s_1 except with location l_1' and all clocks in r_1 reset and s_2 is the same as s_2 except with location l_2' and all clocks in r_2 reset. Now if we follow Definition 3.2.5 it must be the case that $((l_1, l_2), g_1 \wedge g_2, a, r_1 \cup r_2, (l_1', l_2'))$ is a transition in $\mathcal{X}_1 \| \mathcal{X}_2$. We can apply Definition 3.2.7 again to see that $((s_1, s_2), a, (s_1', s_2'))$ is an action transition of $[\![\mathcal{X}_1 \| \mathcal{X}_1]\!]$.

Case: delay transitions . Assume towards an contradiction that $[\![\mathcal{X}]\!]\|[\![\mathcal{Y}]\!]$ has a delay transition $((s_1, s_2), d, (s_1, s_2) + d)$ not in $[\![\mathcal{X}_1\|\mathcal{X}_2]\!]$. Then by Definition 3.2.20 there is a $(s_1, d, s_1 + d)$ in the transitions of $[\![\mathcal{X}_1]\!]$ and a $(s_2, d, s_2 + d)$ in the transitions of $[\![\mathcal{X}_2]\!]$ and there is no $((s_1, s_2), d, (s_1, s_2) + d)$ in the transitions of $[\![\mathcal{X}_1\|\mathcal{X}_2]\!]$. Let l_1 be the location associated with s_1 and l_2 be the location associated with s_2. Then it must be the case that $s_1 + d \models Inv(l_1)$ and $s_2 + d \models Inv(l_2)$. Since $Inv(s_1, s_2) = Inv(s_1) \wedge Inv(s_2)$ by Definition 3.2.19 it must be the case that $(s_1, s_2) + d \models Inv(l_1, l_2)$. By Definition 3.2.7 $((s_1, s_2), d, (s_1, s_2) + d)$ is a transition of $[\![\mathcal{X}_1\|\mathcal{X}_2]\!]$.

Case: Right to left There are two subcases to consider, one for the action transitions and one for the delay transitions.

Case: action transitions Assume towards a contradiction that there is some $((s_1, s_2), a, (s'_1, s'_2))$ in the action transitions of $[\![\mathcal{X}_1\|\mathcal{X}_2]\!]$ but not in the action transitions of $[\![\mathcal{X}_1]\!]\|[\![\mathcal{X}_2]\!]$. By Definition 3.2.7 if $((s_1, s_2), a, (s'_1, s'_2))$ is a transition of $[\![\mathcal{X}_1\|\mathcal{X}_2]\!]$ there is some transition $((l_1, l_2), g_1 \wedge g_2, a, r_1 \cup r_2, (l'_1, l'_2))$ in $\mathcal{X}_1\|\mathcal{X}_2$ for appropriate l_1, l_2, g_1, g_2, r_1 and r_2. By Definition 3.2.19 if $((l_1, l_2), g_1 \wedge g_2, a, r_1 \cup r_2, (l'_1, l'_2))$ is a transition of $\mathcal{X}_1\|\mathcal{X}_2$ then (l_1, g_1, a, r_1, l'_1) is a transition of $\mathcal{X}_1$ and (l_2, g_2, a, r_2, l'_2) is a transition of $\mathcal{X}_2$. This means that (s_1, a, s'_1) is a transition of $[\![\mathcal{X}_1]\!]$ and (s_2, a, s'_2) is a transition of $[\![\mathcal{X}_2]\!]$. We can apply the action synchronizing rule of Definition 3.2.20 to show that $((s_1, s_2), a, (s'_1, s'_2))$ is a transition of $[\![X_1]\!]\|[\![X_2]\!]$.

Case: delay transitions Assume towards an contradiction that there is some

$((s_1, s_2), d, (s_1, s_2) + d)$ in the transitions of $[\![\mathcal{X} \| \mathcal{Y}]\!]$ but not in $[\![\mathcal{X}_1]\!] \| [\![\mathcal{X}_2]\!]$. Let (l_1, l_2) be associated with the state (s_1, s_2). Observe that by Definition 3.2.7 if $((s_1, s_2), d, (s_1, s_2) + d)$ is a transition of $[\![\mathcal{X} \| \mathcal{Y}]\!]$ then it is because $(s_1, s_2) + d \models Inv(l_1, l_2)$. We know then that by Definition 3.2.19 that it must be the case that $(s_1) + d \models Inv(l_1)$ and $(s_2) + d \models Inv(l_2)$. This means that by Definition 3.2.7 $(s_1, d, s_1 + d)$ is a transition of $[\![\mathcal{X}_1]\!]$ and $(s_2, d, s_2 + d)$ is a transition of $[\![\mathcal{X}_2]\!]$. We can now apply Definition 3.2.20 to show that $((s_1, s_2), d, (s_1, s_2) + d)$ is a transition of $[\![\mathcal{X}_1]\!] \| [\![\mathcal{X}_2]\!]$.

□

Before we proceed with the proof of compositionality we introduce a "convenience lemma" to help us relate specifications that have been specified with different syntax (*e.g.,* location names, clock variable names, graph structure) but have the same behavior (*i.e.,* they are equivalent). This convenience lemma will simplify our proof of compositionality.

Lemma 3.2.11 (Syntactic Equivalence)**.** Let $\mathcal{X}_1$ be an MIOTA, $\mathcal{X}$ be a TPMS and $\mathcal{X}_1 \sqsubseteq_{\sim} \mathcal{X}$. Then by definition there is some $\mathcal{X}_2 \sqsubseteq_{\sim} \mathcal{X}$ such that $\mathcal{X}_2$ is defined over the same locations, and clock constraints as $\mathcal{X}$ and $[\![\mathcal{X}_2]\!] \precsim [\![\mathcal{X}_1]\!]$ and $[\![\mathcal{X}_1]\!] \precsim [\![\mathcal{X}_2]\!]$. We say that the two specifications are equivalent (written $\mathcal{X}_1 \sim \mathcal{X}_2$).

Proof. The conclusion follows directly from the definition of TPMS and MIOTA refinement. (Definitions 3.2.14, 3.2.10 and 3.2.13). □

Lemma 3.2.11 says that for any MIOTA $\mathcal{X}_1$ that refines (*resp.* implements) a TPMS $\mathcal{X}$, we can find some other MIOTA $\mathcal{X}_2$ created by syntactic manipulation

of $\mathcal{X}$ (*i.e.,* turning may edges into must edges, fixing parameter values, *etc.*), that has the same behavior as $\mathcal{X}_1$

Theorem 3.2.12 (Compositionality of Refinement)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be two TPMSs and let $\mathcal{X}_1$ and $\mathcal{Y}_1$ be two MIOTA such that $\mathcal{X}_1 \sqsubseteq \mathcal{X}$ and $\mathcal{Y}_1 \sqsubseteq \mathcal{Y}$. Then $\mathcal{X}_1 \| \mathcal{Y}_1 \sqsubseteq \mathcal{X} \| \mathcal{Y}$.

Proof. Assume towards a contradiction that it is not the case that $\mathcal{X}_1 \| \mathcal{Y}_1 \sqsubseteq \mathcal{X} \| \mathcal{Y}$. There are three cases to consider: 1) $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ is missing a required must-transition, 2) $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ has an extra may-transition, 3) $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ has delay transitions not allowed by any valid parametric substitution:

Case 1 $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ is missing a required must-transition. Then either $[\![\mathcal{A}']\!]$ or $[\![\mathcal{B}']\!]$ is missing a required must-transition. But $[\![\mathcal{X}_1]\!]$ or $[\![\mathcal{Y}_1]\!]$ can't be missing any required must transition because $\mathcal{X}_1 \sqsubseteq \mathcal{X}$ and $\mathcal{Y}_1 \sqsubseteq \mathcal{Y}$.

Case 2 $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ has an extra may-transition. Then either $[\![\mathcal{X}_1]\!]$ or $[\![\mathcal{Y}_1]\!]$ has an extra may-transition. But $[\![\mathcal{X}_1]\!]$ or $[\![\mathcal{Y}_1]\!]$ can't have an extra may-transition because $\mathcal{X}_1 \sqsubseteq \mathcal{X}$ and $\mathcal{Y}_1 \sqsubseteq \mathcal{Y}$.

Case 3 $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ has delay transitions not allowed by substituiting valid values for parameters:

By Lemma 3.2.11 there is some $\mathcal{X}_2$ and $\mathcal{Y}_2$ s.t. $\mathcal{X}_1 \sim \mathcal{X}_2$ and $\mathcal{Y}_1 \sim \mathcal{Y}_2$ and both $\mathcal{X}_2 \sqsubseteq_{\approx} \mathcal{X}$ and $\mathcal{Y}_2 \sqsubseteq_{\approx} \mathcal{Y}$. By Lemma 3.2.10 we have that $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!] = [\![\mathcal{X}_1]\!] \| [\![\mathcal{Y}_1]\!]$. We can see by Definition of MTTS composition that if $[\![\mathcal{X}_1 \| \mathcal{Y}_1]\!]$ has different delays than permitted then both $[\![\mathcal{X}_1]\!]$ and $[\![\mathcal{Y}_1]\!]$ have different delays allowed by taking the invariants of either $\mathcal{X}$ or $\mathcal{Y}$ and substituting valid values for parameters. But because $\mathcal{X}_1 \sim \mathcal{X}_2$ and $\mathcal{Y}_1 \sim \mathcal{Y}_2$ and that

$\mathcal{X}_2$ and $\mathcal{Y}_2$ are specifications created by substituting valid values for parameters we know that the delays of $\mathcal{X}_1$ and $\mathcal{Y}_1$ are precisely those allowed by substituting valid values for parameters.

□

Theorem 3.2.13. Let $\mathcal{X}$ and $\mathcal{Y}$ be either TPMSs or MIOTAs. Let ϕ be some reachability property (including timelock and quiescence freedom) over the states of $\mathcal{X}$ and/or $\mathcal{Y}$. Let ψ be some reachability property (not including timelock and quiescence freedom) over the states of $\mathcal{X}$ and/or $\mathcal{Y}$. Then for all $\mathcal{Y}_1$ s.t. $\mathcal{Y}_1 \sqsubseteq \mathcal{Y}$, if $\mathcal{X}\|\mathcal{Y} \not\models_{\diamond} \mathsf{EF}\ \phi$ then $\mathcal{X}\|\mathcal{Y}_1 \not\models_{\diamond} \mathsf{EF}\ \phi$. Likewise, if $\mathcal{X}\|\mathcal{Y} \models_{\square} \mathsf{EF}\ \psi$, then $\mathcal{X}\|\mathcal{Y}_1 \models_{\square} \mathsf{EF}\ \psi$.

Proof. The conclusion follows directly from Theorems 3.2.9 (preservation of timelock freedom) 3.2.7 Quiescence freedom, 3.2.5 (preservation of must-reachability), and 3.2.3 (preservation of negative may-reachability), and 3.2.12 (compositionality of refinement). □

3.3 Symbolic Semantics and Verification

In this section we propose some methods for checking reachability properties and refinement of TPMSs. As it turns out, it is simple to reduce may-reachbility of TPMSs to reachability checking on Parametric Timed Automata (PTA). This reduction is advantageous, because there are existing tools and techniques for checking reachability on PTA. As far as we know, must-reachability and refinement checking cannot be checked efficiently via a reduction to an existing problem for which there are existing tools or techniques. For each problem we devise some novel extensions of existing algorithms.

First, we show how to reduce TPMS may-reachability checking to PTA reachability checking and give an overview of existing tools. Second, we show how may-reachability of TPMS can be encoded as a recursive Horn-clause problem (See [87] for a practical description of the recursive horn-clause problem). Encoding may-reachability as a horn-clause problem enables the use of modern "fully symbolic" SAT-based model-checking algorithms. These two techniques each offer different potential advantages which we discuss in their respective sections. Next, we show how parametric zone-graphs can encode the state-space of a TPMS and how the parametric zone-graph can be used as the basis of an algorithm for must-reachability checking (Sections 3.3.4 and 3.3.3). Lastly, we show how existing algorithms for checking the modal-refinement of MIOTAs can be extended to efficiently check refinement of TPMSs (Section 3.3.5).

3.3.1 Reduction to PTA Parameter Synthesis

Parametric Timed Automata (PTA) [6] take classical Timed Automata and extend them to allow parameters in clock-constraints. The original conception of PTAs allow the parameters to take on values from the non-negative reals ($\mathbb{R}^+$). There are also versions that restrict the parameters to the natural numbers or require that the parameters take on values that satisfy some linear formula (as in TPMS) [10]. Much of the existing literature on PTAs focus on the *parameter synthesis* problem, *i.e.,* what values of the parameters make a some (*e.g.,* a reachability) property true.

In general, the parameter synthesis problem is undecidable for the more general forms of PTAs (*e.g.,*, the ones that don't restrict the parameter values to some finite set) [6]. However, once we restrict ourselves to PTAs that don't admit an infinite number of parameters there are a number of techniques available [14, 13]

and high quality tools [10, 11, 12, 135] that decide the parameter synthesis problem. The prime practical advantage of reducing may-reachability to PTA parameter synthesis is that we can take advantage of mature tools and techniques.

The reduction of may-reachability of TPMSs to parameter synthesis of PTAs is straightforward. First we define the restricted class of PTAs which are the target of our reduction:

Definition 3.3.1 (Parametric Timed Automata)**.** A Parametric Timed Automata (PTA) is a tuple $(Loc, \bar{l}, Clk, Inv, \hookrightarrow, C(\Theta))$ where

- $Loc, \bar{l}, Clk$ are the same as in Definition 3.2.4
- $Inv : Loc \rightarrow \mathcal{P}(Clk)$ assigns downwards closed parametric clock constraints in the form of $x \leq c \pm \alpha$ or $x < c \pm \alpha$ to locations
- $\hookrightarrow \subseteq Loc \times \mathcal{P}(Clk) \times Act \times 2^{Clk} \times Loc$ is the *transition relation*.
- $C(\Theta)$ is a set of linear constraints on a finite set of non-negative integer parameters Θ that are used in $\mathcal{P}(Clk)$

Observe that Definition 3.3.1 is the same as for TPMS except there is no may/-must distinction on transitions. Thus, we can create a PTA to represent the may-behavior of a TPMS by treating the may transitions of the TPMS as the transitions of the PTA. Checking may-reachibility is answered by checking whether there exists a parameter valuation such that the target state is reachable in the PTA: Let $\mathcal{X}$ be a TPMS and $\mathcal{X}'$ be a PTA created by removing the must-transitions of $\mathcal{X}$. Let ϕ be a simple state formula over the locations and variables of $\mathcal{X}$. Then $\mathcal{X} \models_{\diamond} \mathsf{EF}\ \phi$ if there exists some parameter valuation $p \models C(\Theta)$ s.t. $\mathcal{X}' \models \mathsf{EF}\ \phi$.

3.3.2 May Reachability as a Recursive Horn-Clause Problem

Another way to check may-reachability of TPMS is by encoding and solving a recursive horn-clause problem. Note, that this method also decides the parameter synthesis problem of PTAs: As far as we know the technique presented here also represents a new algorithm to decide the parameter synthesis problem of PTAs. The central idea of this technique is to encode instances of the may-reachability problem as a set of recursive Horn-clauses so they can be solved by recent symbolic model-checking techniques such as Interpolation Model Checking (IMC) [138] and generalized versions of the IC3/Property Directed Reachability algorithm (PDR) [86, 43, 42].

One potential advantage of these techniques is that they are fully symbolic: They avoid unrolling the explicit state transition graph of the model by instead manipulating characteristic formula representing the reachable set and transition relation. Consequently, these techniques have the potential to verify reachability properities of TPMS with large discrete state spaces much faster (and utilizing less memory) than the zone-graph technique of Section 3.3.4. Another advantage is that, unlike traditional model-checking, these techniques can generate certificates explaining why a reachability property is not satisfied. The certificate takes the form of a formula that represents an inductive invariant over the transition relation of the model and implies the contradiction of the reachability property. The certificate could then be used as evidence in an assurance case argument or other type of regulatory submission.

In this section we describe a simple extension to the Horn-Clause encoding for reachability of Timed Automata given by Hoder *et al.* in [86]. In order to make the presentation simpler, we show the encoding for a modified (but se-

mantically equivalent) definition of TPMS: We redefine a TPMS $\mathcal{M}$ as a tuple $(l, Init, c, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond}, \Theta, C(\Theta))$ where

- $l : \mathbb{D}^n$ is a vector of n variables, ranging over some finite domain $\mathbb{D}$.
- $Init : \mathbb{D}^n \rightarrow \mathbb{B}$ is a predicate over $\mathbb{D}^n$ (*i.e.,* the variables of l) that returns true if the argument satisfies the initial states of the TPMS.
- $\Theta : \mathbb{N}^p$ is a vector of p of 0 or non-negative integer parameters.
- $c : \mathbb{R}^m$ is a vector of m real-valued clock variables.
- $Inv : \mathbb{D}^n \times \mathbb{R}^m \times \mathbb{N}^p \rightarrow \mathbb{B}$ is a predicate over c, l, *and* Θ that returns true if the valuation of c satisfies the invariant constraint associated with l.
- $\hookrightarrow_{\Box} \subseteq (\mathbb{D}^n \rightarrow \mathbb{B}) \times (\mathbb{R}^m \rightarrow \mathbb{B}) \times 2^{Clk} \times (\mathbb{D}^n \rightarrow \mathbb{B})$ is the *must*-transition relation describing required behavior.
- $\hookrightarrow_{\Diamond} \subseteq (\mathbb{D}^n \rightarrow \mathbb{B}) \times (\mathbb{R}^m \rightarrow \mathbb{B}) \times 2^{Clk} \times (\mathbb{D}^n \rightarrow \mathbb{B})$ is the *may*-transition relation describing allowed behavior.
- $C : \mathbb{N}^p \rightarrow \mathbb{B}$ is the linear constraint over the parameters in Θ.

This modified definition, treats each location as a valuation of the discrete variables which in turn allows for a compact representation of the specification's transition relation using characteristic formula over the variable valuations. The transitions are specified as guarded commands: Characteristic formula over the location and clock valuations are used to indicate both the states the transitions are enabled in and the constraints on the values of the successor states (the primed

variables). Additionally, the actions associated with transitions are dropped because they are not relevant for the reachability problem (we only consider the τ actions as all other actions are disabled in a closed system).

Extending Hoder *et al.*'s timed-automata encoding, the semantics of the *may*-transitions of the TPMS can be directly encoded with the following system of universally quantified recursive Horn-clauses. Let $\Theta \in \mathbb{N}^p$, $l, l' \in \mathbb{D}^n$, $d \in \mathbb{R}^+$, and $c, c' \in \mathbb{R}^m$. Then:

$$\forall_{\Theta,\, l} \; Init(l) \wedge C(\Theta) \rightarrow R(l, \mathbf{0}, \Theta) \tag{3.1}$$

$$\forall_{\Theta,\, l, l',\, c, c'} \; R(l, c, \Theta) \wedge T(l, c, l', c', \Theta) \rightarrow R(l', c', \Theta) \tag{3.2}$$

$$\forall_{\Theta,\, l, l',\, c, c'} \bigvee_{(g_l, g_c, r, g') \in \hookrightarrow_\diamond} \left(g_l(l) \wedge g_c(c, \Theta) \wedge g'(l) \wedge \left(\bigwedge_{c_i \in r} c'_i = 0 \right) \right) \rightarrow T(l, c, l', c', \Theta) \tag{3.3}$$

$$\forall_{\Theta,\, l,\, c, c',\, d} \; Inv(l, c', \Theta) \wedge c' = c + d \rightarrow T(l, c, l, c', \Theta) \tag{3.4}$$

where $R : \mathbb{D}^n \times \mathbb{R}^m \times \mathbb{N}^p \rightarrow \mathbb{B}$ is an uninterpreted predicate representing the set of reachable states and $T : \mathbb{D}^n \times \mathbb{R}^m \times \mathbb{D}^n \times \mathbb{R}^m \times \mathbb{N}^p \rightarrow \mathbb{B}$ is an uninterpreted predicate that represents the transition relation. Equation 3.1 encodes the set of reachable states: Any valuation of location and parameter variables that satisfies the inital state and parametric constraint predicates is in the reachable set. Equa-

tion 3.2 specifies that any state reachable in a single step from the reachable set is also in the reachable set. Equation 3.3 encodes the *may*-action transitions of the TPMS: If the valuation of the location and clock variables satisfy the transition guards then that transition is enabled. Equation 3.4 encodes the delay transition; clocks are allowed to advance as long as the invariant predicate Inv is satisfied.

May-reachability using the Horn encoding can be checked by asserting Equations 3.1-3.4 to a suitable solver (*e.g.,* Z3 [57]) and then querying the following formula:

$$\exists_{\Theta \in \mathbb{N}^p} \exists_{l \in \mathbb{D}^n} \exists_{c \in \mathbb{R}^m} \Phi(l, c) \wedge R(l, c, \Theta) \tag{3.5}$$

where $\Phi : \mathbb{D}^n \times \mathbb{R}^m \rightarrow \mathbb{B}$ is a predicate specifing the set of states we want to check for *may*-reachability. If equations 3.1-3.4 and 3.5 are satisfiable together, then states satisfying Φ are *may*-reachable. If they are unsatisfiable then no states specified by Φ are *may*-reachable.

Our Horn clause encoding of *may*-reachability for TPMS simply treats the parameters as discrete variables that never change. In fact, the only difference between the our encoding for *may*-reachability of TPMS and the encoding of Timed Automata reachability given by Hoder *et al.* in [86] is the addition of variables representing the parameter valuations and the constraints over the valuation of those variables. Thus, if a Horn-clause solver is able to decide reachability for Hoder *et al.*'s Timed Automata encoding (such as the solver described in [86] and available in the Z3 SMT solver) it will decide *may*-reachability of TPMS using the above encoding.

3.3.3 Symbolic Semantics: Parametric Zone-Graphs

Some techniques (*e.g.,* [92]) for analysis of parametric timed systems extend the classical notion of *zones* and *zone-graphs* for timed automata [33] to parametric zones and zone-graphs. A *parametric zone* is in the form of a parametric clock constraint (defined in Section 3.2.1), representing the maximal set of clock valuations satisfying any *instance* of the parametric clock constraint. Let v be a clock valuation and D be a parametric zone, we define $v \in D$ iff v satisfies the parametric clock constraint of D, *i.e.,* $v \models D$. Sometimes (as in must-reachability checking) we want to know if $v \in D$ for all possible parametric assignments. We define $v \in D$ for all possible parametric assignments iff $v \models_{\Box} D$ (Likewise, we can extend the notion to sets of valuations constrained by a formula ϕ: $\phi \models_{\Box} D$ iff ϕ satisfies D for every valid parameter valuation). Given two parametric zones D_1 and D_2, if $v \in D_1$ implies $v \in D_2$, then zone D_1 is included in D_2, denoted by $D_1 \subseteq D_2$. We define $D^{\uparrow} = \{v + d \mid v \in D, d \in \mathbb{R}_{\geq 0}\}$ for the zone progression, and $r(D) = \{v[r \mapsto 0] \mid v \in D\}$ for the clock reset of zones. A *parametric zone-graph* is a graph where each node consists of a location and a parametric zone. We define the symbolic semantics of time-parametric modal specifications based on parametric zone-graph as follows.

Definition 3.3.2 (Specification's Symbolic Semantics)**.** The symbolic semantics of a time-parametric modal specification $\mathcal{M} = (Loc, \bar{l}, Clk, Act, Inv, \hookrightarrow_{\Box}, \hookrightarrow_{\Diamond}, C(\Theta))$, denoted by $[\![\mathcal{M}]\!]_{\mathsf{z}}$, is a *parametric zone-graph* $(S, \bar{s}, Act, \rightsquigarrow_{\Box}, \rightsquigarrow_{\Diamond}, \rightsquigarrow_{\mathsf{d}}, C(\Theta))$ where

- $S = \{\langle l, D\rangle \in Loc \times \mathcal{P}(Clk) \mid D \subseteq Inv(l)\}$ is a finite set of *symbolic states*, and $\bar{s} = \langle \bar{l}, D_0\rangle$ is the *initial state*

- symbolic *must action* transition: $\langle l, D\rangle \overset{a}{\rightsquigarrow}_{\Box} \langle l', r(D \wedge g) \wedge Inv(l')\rangle$ if there is a must transition $l \overset{g,a,r}{\hookrightarrow}_{\Box} l'$ in $\mathcal{M}$ and $g \models_{\Box} D$ and $r(D \wedge g) \models_{\Box} Inv(l')$

- symbolic *may action* transition: $\langle l, D\rangle \overset{a}{\rightsquigarrow}_{\Diamond} \langle l', r(D \wedge g) \wedge Inv(l')\rangle$ if there is a may transition $l \overset{g,a,r}{\hookrightarrow}_{\Diamond} l'$ in $\mathcal{M}$

- symbolic *delay* transition: $\langle l, D\rangle \rightsquigarrow_{\mathsf{d}} \langle l, D^{\uparrow} \wedge Inv(l)\rangle$

A symbolic state $\langle l, D\rangle$ in $[\![\mathcal{M}]\!]_{\mathsf{z}}$ corresponds to a set of states in the semantics of $\mathcal{M}$. A symbolic transition $\langle l, D\rangle \rightsquigarrow_{\gamma} \langle l', D'\rangle$ with $\gamma \in \{\Box, \Diamond, \mathsf{d}\}$ implies that, for every $v' \in D'$, there must exist at least one MTTS $M \in [\![\mathcal{M}]\!]$ which has a transition $\langle l, v\rangle \rightarrow_{\gamma} \langle l', v'\rangle$ for some $v \in D$. Indeed, we show that the symbolic semantics given in Definition 3.3.2 is a correct and full characterization of the operation semantics given in Definition 3.2.9 as follows.

Theorem 3.3.1. Let $\mathcal{M}$ be a time-parametric modal specification, $[\![\mathcal{M}]\!]_{\mathsf{z}}$ be its symbolic semantics and $[\![\mathcal{M}]\!]$ be its operational semantics.

- *(Soundness)* if the initial symbolic state $\langle \bar{l}, D_{\mathbf{0}}\rangle$ in $[\![\mathcal{M}]\!]_{\mathsf{z}}$ must (*resp.* may) lead to a target state $\langle l_f, D_f\rangle$, then for all $v_f \in D_f$, state $\langle l_f, v_f\rangle$ must (*resp.* may) be reachable from the initial state $\langle \bar{l}, \mathbf{0}\rangle$ in some $M \in [\![\mathcal{M}]\!]$

- *(Completeness)* if, in any $M \in [\![\mathcal{M}]\!]$, a target state $\langle l_f, v_f\rangle$ must (*resp.* may) be reachable from the initial state $\langle \bar{l}, \mathbf{0}\rangle$, then state $\langle \bar{l}, D_{\mathbf{0}}\rangle$ in $[\![\mathcal{M}]\!]_{\mathsf{z}}$ must (*resp.* may) lead to $\langle l_f, D_f\rangle$ for some D_f such that $v_f \in D_f$

Proof. We will prove by induction on the length of paths. Without loss of generality, we assume that all paths are expressed in the form of alternating (must or may) action transitions and delay transitions, *i.e.,* $\cdots \langle l_{i-1}, v_{i-1}\rangle \xrightarrow{a} \langle l_i, v_i\rangle \xrightarrow{d} \langle l_{i+1}, v_{i+1}\rangle \cdots$ for $a \in Act$ and $d \in \mathbb{R}_{\geq 0}$.

(Soundness) Assume $\langle \bar{l}, D_{\mathbf{0}} \rangle \rightsquigarrow^* \langle l_n, D_n \rangle \overset{\sigma}{\rightsquigarrow} \langle l_{n+1}, D_{n+1} \rangle$, where $\rightsquigarrow^*$ represents a succession of transitions. By induction, we have $\langle \bar{l}, \mathbf{0} \rangle \rightarrow^* \langle l_n, v_n \rangle$ for all $v_n \in D_n$. We need to prove for all $v_{n+1} \in D_{n+1}$, there is a transition $\langle l_n, v_n \rangle \xrightarrow{\sigma} \langle l_{n+1}, v_{n+1} \rangle$. There are two cases, since σ can be an action or a delay.

- Suppose $\langle l_n, D_n \rangle \overset{a}{\rightsquigarrow}_{\diamond} \langle l_{n+1}, D_{n+1} \rangle$ for $a \in Act$. Based on Definition 3.3.2, we have $l_n \overset{g,a,r}{\hookrightarrow}_{\diamond} l_{n+1}$ and $D_{n+1} = r(D_n \wedge g) \wedge Inv(l_{n+1})$. By Definition 3.2.9, there is a transition $\langle l_n, v_n \rangle \xrightarrow{a}_{\diamond} \langle l_{n+1}, v_{n+1} \rangle$ in some $M \in [\![\mathcal{M}]\!]$ such that $v_n \in g$. Thus, for all $v_{n+1} \in D_{n+1}$, there is a $v_n \in D_n$ such that $v_n \in g$, $v_{n+1} \in Inv(l_{n+1})$ and $v_{n+1} = v_n[r \mapsto 0]$.

- Suppose $\langle l_n, D_n \rangle \overset{a}{\rightsquigarrow}_{\Box} \langle l_{n+1}, D_{n+1} \rangle$ for $a \in Act$. Based on Definition 3.3.2, we have $l_n \overset{g,a,r}{\hookrightarrow}_{\Box} l_{n+1}$ and $D_{n+1} = r(D_n \wedge g) \wedge Inv(l_{n+1})$ and $g \models_{\Box} D_n$ and $r(D_n \wedge g) \models_{\Box} Inv(l_{n+1})$. By Definition 3.2.9, there is a transition $\langle l_n, v_n \rangle \xrightarrow{a}_{\Box} \langle l_{n+1}, v_{n+1} \rangle$ in all $M \in [\![\mathcal{M}]\!]$ such that $v_n \in g$. Thus, for all $v_{n+1} \in D_{n+1}$, there is a $v_n \in D_n$ such that $v_n \in g$, $v_{n+1} \in Inv(l_{n+1})$ and $v_{n+1} = v_n[r \mapsto 0]$.

- Suppose $\langle l_n, D_n \rangle \overset{d}{\rightsquigarrow}_{\mathsf{d}} \langle l_{n+1}, D_{n+1} \rangle$ for $d \in \mathbb{R}_{\geq 0}$. From Definition 3.3.2, we have $l_n = l_{n+1}$ and $D_{n+1} = D_n^{\uparrow} \wedge Inv(l_n)$. Due to the definition of zone progression, we have $D_{n+1} = \{v_n + d \mid v_n \in D_n, d \in \mathbb{R}_{\geq 0}$ and $v_n + d \in Inv(l_n)\}$. Based on Definition 3.2.9, we have $\langle l_n, v_n \rangle \xrightarrow{d}_{\mathsf{d}} \langle l_n, v_n + d \rangle$ if $v_n + d \in Inv(l_n)$. Thus, for all $v_{n+1} \in D_{n+1}$, there is a $v_n \in D_n$ such that $v_{n+1} = v_n + d$ and $v_n, v_{n+1} \in Inv(l_n)$.

(Completeness) Assume $\langle \bar{l}, \mathbf{0} \rangle \rightarrow^* \langle l_n, v_n \rangle \xrightarrow{\sigma} \langle l_{n+1}, v_{n+1} \rangle$. The induction step gives $\langle \bar{l}, D_{\mathbf{0}} \rangle \rightsquigarrow^* \langle l_n, D_n \rangle$ and $v_n \in D_n$. We need to prove that $\langle l_n, D_n \rangle \overset{\sigma}{\rightsquigarrow} \langle l_{n+1}, D_{n+1} \rangle$ for some D_{n+1} and $v_{n+1} \in D_{n+1}$. There are two cases:

- Suppose $\langle l_n, v_n\rangle \xrightarrow{a}_{\diamond} \langle l_{n+1}, v_{n+1}\rangle$ for $a \in Act$ in any $M \in [\![\mathcal{M}]\!]$. Based on Definition 3.2.9, there is a transition $l_n \overset{g,a,r}{\hookrightarrow}_{\diamond} l_{n+1}$ in $\mathcal{M}$ and $v_n \in g$, $v_{n+1} = v_n[r \mapsto 0]$ and $v_{n+1} \in Inv(l_{n+1})$. By Definition 3.3.2, we have $\langle l_n, D_n\rangle \overset{a}{\rightsquigarrow}_{\diamond} \langle l_{n+1}, D_{n+1}\rangle$ and $D_{n+1} = r(D_n \wedge g) \wedge Inv(l_{n+1})$. Thus, $v_{n+1} \in D_{n+1}$.

- Suppose $\langle l_n, v_n\rangle \xrightarrow{a}_{\Box} \langle l_{n+1}, v_{n+1}\rangle$ for $a \in Act$ for all $M \in [\![\mathcal{M}]\!]$. Based on Definition 3.2.9, there is a transition $l_n \overset{g,a,r}{\hookrightarrow}_{\gamma} l_{n+1}$ in $\mathcal{M}$ and $v_n \models_{\Box} g$, $v_{n+1} = v_n[r \mapsto 0]$ and $v_{n+1} \models_{\Box} Inv(l_{n+1})$. By Definition 3.3.2, we have $\langle l_n, D_n\rangle \overset{a}{\rightsquigarrow}_{\Box} \langle l_{n+1}, D_{n+1}\rangle$ and $D_{n+1} = r(D_n \wedge g) \wedge Inv(l_{n+1})$ with $g \models_{\Box} D_n$ and $r(D_n \wedge g) \models_{\Box} Inv(l_{n+1})$. Thus, $v_{n+1} \in D_{n+1}$.

- Suppose $\langle l_n, v_n\rangle \xrightarrow{d}_{\mathsf{d}} \langle l_{n+1}, v_{n+1}\rangle$ for $d \in \mathbb{R}_{\geq 0}$. Then we have $l_n = l_{n+1}$, $v_{n+1} = v_n + d$ and $v_n, v_{n+1} \in Inv(l_n)$. From Definition 3.3.2, we get $\langle l_n, D_n\rangle \overset{d}{\rightsquigarrow}_{\mathsf{d}} \langle l_{n+1}, D_{n+1}\rangle$ and $D_{n+1} = D_n^{\uparrow} \wedge Inv(l_n) = \{v_n + d \mid v_n \in D_n, d \in \mathbb{R}_{\geq 0}$ and $v_n + d \in Inv(l_n)\}$. Thus, $v_{n+1} \in D_{n+1}$.

$\Box$

The clock differences in a (parametric) zone-graph may increase unboundedly due to transitions which reset some clocks but not others, which can result in an infinite zone-graph. We can apply techniques such as *zone normalization* to normalize zones and guarantee a finite zone graph. See [33] for an indepth discussion on how these clock differences can increase unboundedly and how to implement zone normalization techniques.

Example Figure 3.5 illustrates the parametric zone-graph induced from the time-parametric modal specification shown in Figure 3.3. There are 7 symbolic states

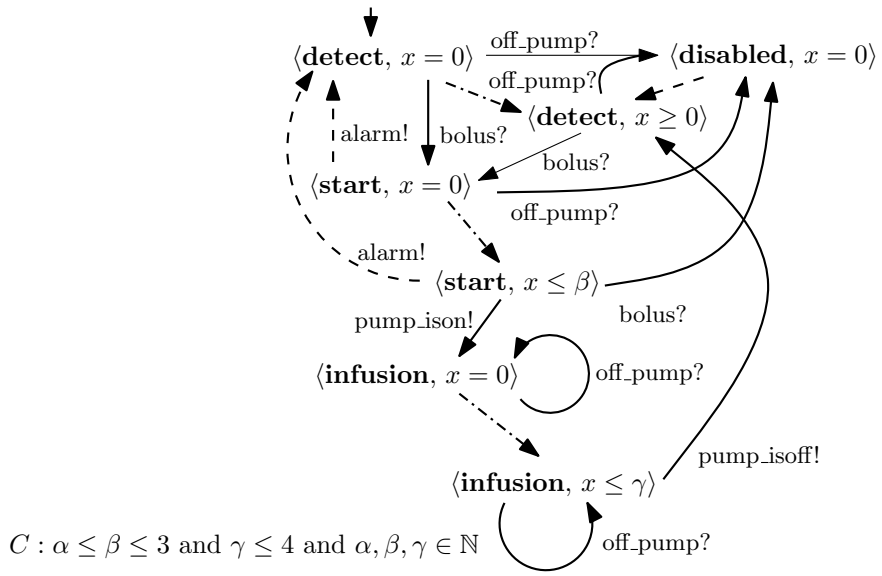


Figure 3.5: Parametric zone-graph of the specification in Figure 3.3.

in the zone-graph; for example, $\langle$**infusion**$, x \leq \gamma\rangle$ is a state associated with a location **infusion** and a parametric zone $x \leq \gamma$, which is bounded by the linear constraint: $\gamma \leq 4$.

In Figure 3.5, *solid* lines represent symbolic must action transitions, *e.g.,* there is a must transition labeled with action "bolus?" from state $\langle$**detect**$, x = 0\rangle$ to $\langle$**start**$, x = 0\rangle$; *dashed* lines are for the symbolic may action transitions, *e.g.,* state $\langle$**start**$, x \leq \beta\rangle$ may loop back to $\langle$**detect**$, x = 0\rangle$ with action "alarm!"; and *dash dotted* lines are for symbolic delay transitions, *e.g.,* state $\langle$**detect**$, x = 0\rangle$ evolves to $\langle$**detect**$, x \geq 0\rangle$ via zone progression.

3.3.4 Zone-Graph Symbolic Reachability Analysis

Reachability analysis lies at the core of many verification problems, *e.g.,* we can verify safety properties by checking whether some *bad* states are reachable. In-

Algorithm 1 Must (*resp.* may) reachability analysis on parametric zone-graphs

Precondition: G is a a parametric zone-graph whose initial state is $\langle \bar{l}, D_{\mathbf{0}} \rangle$

Postcondition: returns "YES", if $\langle l_f, \phi_f \rangle$ is reachable from $\langle \bar{l}, D_{\mathbf{0}} \rangle$; "NO", otherwise

```
function REACHABILITYCHECK(G, ⟨l_f, φ_f⟩)
    seen ← ∅, wait ← {⟨l̄, D_0⟩}
    while wait ≠ ∅ do
        ⟨l, D⟩ ← pop(wait)
        if l = l_f and D ∩ φ_f ≠ ∅ for all (resp. some) valid f(Θ) then
            return "YES"
        end if
        if D ⊈ D′ for all ⟨l, D′⟩ ∈ seen then
            seen ← seen ∪ ⟨l, D⟩
            for ⟨l′, D′⟩ such that (⟨l, D⟩, ⟨l′, D′⟩) ∈ ⇝_d and ⇝_□ (resp. ⇝_◇) do
                wait ← wait ∪ ⟨l′, D′⟩
            end for
        end if
    end while
    return "NO"
end function
```

spired by the zone-graph based reachability algorithm in [33], we propose a symbolic algorithm for the reachability analysis of parametric zone-graphs, which is useful for the verification of time-parametric modal specifications.

As illustrated in Algorithm 1, we check whether a target state $\langle l_f, \phi_f \rangle$ is reachable from the initial state $\langle \bar{l}, D_{\mathbf{0}} \rangle$ by exploring the state-space of parametric zone-graph [1] on-the-fly. Note that we twist Algorithm 1 for both the *must* and *may* reachability analysis. The algorithm maintains two sets of states: PASSED for those having been traversed and WAIT for those to be considered. Starting with

[1] We assume a normalized zone-graph with finite transition relations.

the initial state $\langle \bar{l}, D_{\mathbf{0}} \rangle$, the algorithm processes each element $\langle l, D \rangle$ of WAIT till the set becomes empty. Note that a transition is a must transition ($\rightsquigarrow_{\Box}$) only if it is enabled for *all* parameter assignments. If $l = l_f$ and $D \cap \phi_f \neq \emptyset$ for all valid parameter assignment $f(\Theta)$, then the target *must* be reachable. This can be checked by querying an SMT solver with an assertion of the following form:

$$\exists_{\Theta} \left[C(\Theta) \wedge \neg \exists_{Clk} \left(D \wedge \phi_f \right) \right]$$

If the above is satisfiable, then there exists an assignment to parameters Θ that satisfy $\mathcal{P}(Clk)$ such that there is no possible valuation of the clocks Clk that intersect with the target valuation ϕ_f. On the other hand, we say that $\langle l_f, \phi_f \rangle$ *may* be reachable if $D \cap \phi_f \neq \emptyset$ is only true for some $f(\Theta)$, in the sense that the target may (not) be reachable for some implementation of the time-parametric modal specification. If $\langle l, D \rangle$ does not hit the target and has not been traversed, then the algorithm adds $\langle l, D \rangle$ to PASSED and all its successor states to WAIT. The algorithm terminates when WAIT is empty, and outputs that the target state $\langle l_f, \phi_f \rangle$ is not reachable.

The termination of Algorithm 1 is guaranteed, because the parametric zone-graph G has a finite set of symbolic states and finite transition relations, *i.e.,* the size of WAIT is finite. The correctness of Algorithm 1 is given by Theorem 3.3.1.

Example Consider the parametric zone-graph shown in Figure 3.5. A target $\langle \textbf{infusion}, x \leq 0 \rangle$ *must* be reachable, because there is a path

$$\langle \textbf{detect}, x = 0 \rangle \rightarrow \langle \textbf{start}, x = 0 \rangle \rightarrow \langle \textbf{start}, x \leq \beta \rangle \rightarrow \langle \textbf{infusion}, x = 0 \rangle$$

and, for any valid parameter values satisfying the constraint: $0 \leq \alpha \leq \beta \leq 3 \wedge 0 \leq \gamma \leq 4$, we always have that $x = 0 \cap x \leq \gamma \neq \emptyset$. Suppose the target is

$\langle$**infusion**$, x = 3\rangle$, then it *may* be reachable, because $(x \leq \gamma) \cap (x = 3) \neq \emptyset$ is true for some parameter assignments, *e.g.,* $\gamma \geq 3$, but false for others, *e.g.,* $\gamma < 3$.

3.3.5 Symbolic Modal Refinement Checking

Recall from Section 3.2.3 that we verify whether a MIOTA is an implementation of a TPMS via a modal refinement check. We can directly apply the definition of TPMS refinement (Definition 3.2.14) by reducing the problem to the timed refinement check of MIOTA, where standard zone-graph based algorithms (*e.g.,* [183]) are available: Given a time-parametric modal specification $\mathcal{M}$ and potential implementation $\mathcal{I}$, we first solve the linear constraints $C(\Theta)$ and obtain a finite set of parameter assignments. For each parameter assignment f, we substitute all the occurrences of parameters Θ with values $f(\Theta)$, resolving the timing variability of the specification and leaving a set of concrete specifications $[\![\mathcal{M}]\!]$. $[\![\mathcal{I}]\!] \sqsubseteq\!\!\!\sim \mathcal{M}$ iff $\mathcal{I} \precsim S$ for some $S \in [\![\mathcal{M}]\!]$.

Unfortunately, brute-force checking refinement of $\mathcal{I}$ vs. each possible parametric concretization of $\mathcal{M}$ can be too slow for practical applications. If one multiplies all the constants in the specifications by some factor (*e.g.,* to increase the resolution of the time units considered), the number of possible parametric concretizations increases exponentially (*i.e.,* it is "scaling-sensitive"). Furthermore, for a given implementation $\mathcal{I}$ and specification $\mathcal{M}$ only a few (perhaps only one) of the parametric concretizations of $\mathcal{M}$ witnesses $\mathcal{I} \sqsubseteq\!\!\!\sim \mathcal{M}$. We want to avoid checking as many irrelevent concretizations as possible because the known procedures for checking the timed refinement of modal timed automata can be expensive[2].

[2]The zone and region techniques described [183, 182, 48] are EXPTIME.

The rest of Section 3.3.5 is devoted to a TPMS implementation checking algorithm that avoids brute-force checking of each possible parametric concretization. Our algorithm attempts to quickly narrow the parameter search space by leveraging information learned from repeatedly applying a modified version of Weise *et al.*'s refinement checking algorithm [183]. First we describe Carsten *et al.*'s algorithm and import the relevant theorems from [183]. Then we we give a modified version of Carsten *et al.*'s original algorithm and show how it can be used as part of a procedure that avoids searching irrelevant parameter valuations. Finally, we prove the correctness of our algorithm.

Weise's Algorithm for Checking Refinement Between MIOTAs

Algorithm 2 is psuedo-code of an algorithm for checking *timed-bisimulation* originally presented in [183] and extended for checking *timed-refinement* in [182]. First, the algorithm constructs the syntactic product of the two specifications $\mathcal{P}$ and $\mathcal{Q}$ (Definition 3.3.3). Then the algorithm unrolls a Backward Stable (BS) zone-graph (see Definition 3.3.4) of the syntactic product. The BS zone-graph of the product is a symbolic representation of an *overapproximation* of the refinement relation between the input specifications. Finally, the algorithm iteratively removes all the non-closed zones until all remaining zones in the zone-graph are *closed* (*i.e.,* each state in each zone can match the required action and delay transitions, see Definition 3.3.5). If the initial zone is removed then it is not the case that $\mathcal{P} \preceq \mathcal{Q}$.

Definition 3.3.3 (Syntactic Product of MIOTA)**.** The syntactic product of two MIOTA $\mathcal{P} = (Loc_{\mathcal{P}}, \bar{l}_{\mathcal{P}}, Clk_{\mathcal{P}}, Act_{\mathcal{P}}, Inv_{\mathcal{P}}, \hookrightarrow_{\Box,\mathcal{P}}, \hookrightarrow_{\Diamond,\mathcal{P}})$ and
$\mathcal{Q} = (Loc_{\mathcal{Q}}, \bar{l}_{\mathcal{Q}}, Clk_{\mathcal{Q}}, Act_{\mathcal{Q}}, Inv_{\mathcal{Q}}, \hookrightarrow_{\Box,\mathcal{Q}}, \hookrightarrow_{\Diamond,\mathcal{Q}})$ is

$\mathcal{PQ} = (Loc_{\mathcal{P}} \times Loc_{\mathcal{P}}, (\bar{l}_{\mathcal{P}}, \bar{l}_{\mathcal{Q}}), Clk_{\mathcal{P}} \cup Clk_{\mathcal{Q}} \cup \{t\}, Act_{\mathcal{P}} \cup Act_{\mathcal{Q}}, Inv_{\mathcal{PQ}}, \hookrightarrow_{\Box,\mathcal{PQ}}$ $, \hookrightarrow_{\Diamond,\mathcal{PQ}})$ where:

- $Inv_{\mathcal{PQ}}((l_{\mathcal{P}}, l_{\mathcal{Q}})) = Inv_{\mathcal{P}}(l_{\mathcal{P}}) \wedge Inv_{\mathcal{Q}}(l_{\mathcal{Q}})$.
- $\hookrightarrow_{\Box,\mathcal{PQ}}$ and $\hookrightarrow_{\Diamond,\mathcal{PQ}}$ are defined by the following rules:

$$\frac{(l_1, g_1, a, r_1, l_1') \in \hookrightarrow_{\gamma,\mathcal{P}} \quad (l_2, g_2, a, r_2, l_2') \in \hookrightarrow_{\gamma,\mathcal{Q}}}{\big((l_1, l_2), g_1 \wedge g_2, a, r_1 \cup r_2, (l_1', l_2')\big) \in \hookrightarrow_{\gamma,\mathcal{PQ}}} \textit{ (synchronizing)}$$

$$\frac{(l_1, g_1, \tau, r_1, l_1') \in \hookrightarrow_{\gamma,\mathcal{X}} \quad \mathcal{X} \in \{\mathcal{P}, \mathcal{Q}\}}{\big((l_1, l_2), g_1, \tau, r_1 \cup t, (l_1', l_2)\big) \in \hookrightarrow_{\gamma,\mathcal{PQ}}} \textit{ (interleaving)}$$

- t is a fresh clock.

Definition 3.3.4 (Backward Stable Zone Graph)**.** A zone graph is called backward stable if each state in each zone is reachable by some state in each predecessor via an action transition: $\langle l, D\rangle \overset{a}{\rightsquigarrow} \langle l', D'\rangle$ implies that $\forall_{v' \in D'}\ \exists_{v \in D^{\uparrow}}$ s.t. $\langle l, v\rangle \overset{a}{\rightarrow} \langle l', v'\rangle$ is in the corresponding MTTS.

Definition 3.3.5 (Product Zone Closure)**.** All the points in a zone $\langle l_{pq}, D_{pq}\rangle$ from the zone-graph of MIOTA $\mathcal{PQ}$ are closed with respect to MIOTA specifications $\mathcal{P}$ and $\mathcal{Q}$ if each point can match the delays of $\mathcal{P}$ & $\mathcal{Q}$, match the *may*-actions of $\mathcal{P}$, and match the *must*-actions of $\mathcal{Q}$. More formally, all the points of $\langle l_{pq}, D_{pq}\rangle$ are closed w.r.t. $\mathcal{P}$ and $\mathcal{Q}$ iff the following formula holds (we abuse notation and treat zones and clock constraints as sets of valuations or points that satisfy those

Algorithm 2 On-the-fly version of Weise's zone algorithm to check whether a Timed Modal Specification $P \preceq Q$.

Precondition: Two MIOTA P & Q

Postcondition: returns TRUE if $P \preceq Q$, FALSE otherwise

```
1:  function RefCheck(P, Q)
2:      PQ ← SyntacticProduct(P, Q)
3:      seen ← ∅,  check ← EmptyStack()
4:      Push(check, ⟨l̄_pq, 0⟩)
5:      while NotEmpty(check) do
6:          ⟨l_pq, D_pq⟩ ← Pop(check)
7:          if Relevant(⟨l_pq, D_pq⟩) then
8:              if NotClosed(⟨l_pq, D_pq⟩) then
9:                  if ⟨l_pq, D_pq⟩ = ⟨l̄_pq, 0⟩ then
10:                     return False
11:                 end if
12:                 Remove(⟨l_pq, D_pq⟩)
13:                 PushAll(check, Pred(⟨l_pq, D_pq⟩)
14:             end if
15:         end if
16:         if ⟨l_pq, D_pq⟩ ∉ seen then
17:             seen ← seen ∪ ⟨l_pq, D_pq⟩
18:             PushAll(check, Succ(⟨l_pq, D_pq⟩)
19:         end if
20:     end while
21:     return True
22: end function
```

constraints):

$$D^{\uparrow}_{pq} \cap Inv(l_p) \subseteq D^{\uparrow}_{pq} \cap Inv(l_{pq}) \tag{3.6}$$

$$\wedge \; D^{\uparrow}_{pq} \cap Inv(l_q) \subseteq D^{\uparrow}_{pq} \cap Inv(l_{pq}) \tag{3.7}$$

$$\wedge \bigwedge_{a \in \textsc{May}(Act_P \cup Act_Q)} D^{\uparrow}_{pq} \cap Inv(l_{pq}) \cap \mathrm{G}_{PQ}(a, l_{pq}) \subseteq D^{\uparrow}_{pq} \cap Inv(l_{pq}) \cap \mathrm{G}_{Q}(a, l_{q}) \tag{3.8}$$

$$\wedge \bigwedge_{a \in \textsc{Must}(Act_P \cup Act_Q)} D^{\uparrow}_{pq} \cap Inv(l_{pq}) \cap \mathrm{G}_{Q}(a, l_{q}) \subseteq D^{\uparrow}_{pq} \cap Inv(l_{pq}) \cap \mathrm{G}_{PQ}(a, l_{pq}) \tag{3.9}$$

Where MAY (resp. MUST): $2^{Act_{\mathcal{P}} \cup Act_{\mathcal{Q}}} \rightarrow 2^{Act_{\mathcal{P}} \cup Act_{\mathcal{Q}}}$ is a helper function that filters out the set of actions associated with *may* (resp. *must*)-transitions and $\mathrm{G}_{\mathcal{X}} : Act_{\mathcal{X}} \times Loc_{\mathcal{X}}$ is a helper function that gives the union of all the guards of the edges associated with the specified action and location of specification $\mathcal{X}$ (the null set is given if there is no such edge for the specified location).

Our presentation of Weise's algorithm has one important difference compared to what was originally described in [183]. In the original presentation, instead of removing non-closed zones from the zone graph (line 12 of Algorithm 2), only the non-closed *regions* of a zone were removed (Weise called this reducing a zone). Does simply removing the zone affect the correctness of the algorithm when applied to MIOTA? As it turns out, if the two specifications are *action determined* (Definition 3.3.6), then the product zone graph exactely represents the refinement relation if the the implementation is valid.

Definition 3.3.6 (Action Determined MIOTA)**.** Let $\mathcal{X}$ be some MIOTA specification and Let R be the greatest refinement relation satisfying Definition 3.2.10

over $[\![\mathcal{X}]\!]$. We say that some MIOTA $\mathcal{X}$ is action determined if $(s', s'') \in R$ when $(s, a, s'), (s, a, s'') \in \rightarrow_{\Box,\mathcal{X}}$ or $(s, a, s'), (s, a, s'') \in \rightarrow_{\Diamond,\mathcal{X}}$.

Definition 3.3.6 simply states that, for a given state, the specification cannot have more than one transition with the same action-label if those transitions lead to inequivalent states.

Lemma 3.3.2 (Product ZG Closure)**.** Let $\mathcal{P}$ and $\mathcal{Q}$ be MIOTA. Let both $\mathcal{P}$ and $\mathcal{Q}$ be action determined. If $[\![\mathcal{P}]\!] \preceq [\![\mathcal{Q}]\!]$ then the product zone-graph of $\mathcal{P}$ and $\mathcal{Q}$ is closed.

Proof. Supposed towards a contradiction that $\mathcal{P} \preceq \mathcal{Q}$ but there is some zone $\langle l'_{pq}, D_{pq}\rangle'$ in the zone-graph of the product that is not closed. There are two possibilities: $\langle l'_{pq}, D'_{pq}\rangle$ is the initial zone or it is some successor zone. We examine each case:

1. $\langle l'_{pq}, D'_{pq}\rangle$ is the initial zone. If $\langle l'_{pq}, D'_{pq}\rangle$ is the initial zone then it is not the case that $\mathcal{P} \preceq \mathcal{Q}$.

2. $\langle l'_{pq}, D'_{pq}\rangle$ is some succesor zone. Then either its predecessor $\langle l_{pq}, D_{pq}\rangle$ is not fully closed, or the states represented by the predecessor are not action determined. Why? if the predecessor is closed, then there must be some other transition $\langle l_{pq}, D_{pq}\rangle \rightarrow \langle l''_{pq}, D''_{pq}\rangle$ in the product zone graph labeled with the same action and $\langle l''_{pq}, D''_{pq}\rangle$. If $\langle l''_{pq}, D''_{pq}\rangle$ is closed but $\langle l'_{pq}, D'_{pq}\rangle$ isn't, then $(\langle l''_{pq}, D''_{pq}\rangle, \langle l'_{pq}, D'_{pq}\rangle) \notin R$ and therefore aren't action determined.

The argument above can be inductively applied backwards until one reaches the initial zone. $\Box$

Since the product zone graph must represent the refinement relation for action determined specifications, simply removing unclosed zones (as opposed to reducing) from the product graph is safe. This ability will be essential when we extend Weise's algorithm to check refinement of TPMSs. We don't believe the restriction to action determined TPMSs much a hardship: developers will likely want to avoid action non-deterministic specifications for medical devices if possible.

Algorithm for Checking Refinement Between TPMSs

Algorithm 3 Scale-insensitive algorithm to check whether TPMS $\mathcal{P}$ is an implementation of $\mathcal{Q}$.

Precondition: Two TPMS $\mathcal{P}$ & $\mathcal{Q}$.

Postcondition: Returns TRUE if $\mathcal{P} \sqsubseteq_{\sim} \mathcal{Q}$, FALSE otherwise.

```
function RefCheck(P, Q)
    Ω ← C(Θ)_Q
    while TRUE do
        (R, Ω') ← GuessRefCheck(P, Q, Ω)
        if R = TRUE then
            return TRUE
        else
            if Ω' = TRUE then
                return FALSE
            else
                Ω ← Ω ∧ ¬Ω'
            end if
        end if
    end while
end function
```

Our algorithm (Algorithm 3) takes a modified version of Weise's algorithm and wraps it in a parameter search loop. The loop maintains a formula, Ω, that constrains the possible valuations of parameters. On the first iteration Ω is simply the parametric constraint from the TPMS $\mathcal{Q}$. On each iteration Ω is passed to the sub-routine GUESSREFCHECK (Algorithm 4) that implements a modified version of Weise's refinement check. If GUESSREFCHECK "guesses" parameters that allow it to build a closed BS-zone graph containing the initial states then it returns true. If GUESSREFCHECK did not guess the correct parameters then Ω is narrowed to exclude those valuations on the next iteration. Eventually, if GUESSREFCHECK uses a complete guessing procedure, either witnessing parameter valuations will be found (and the algorithm returns TRUE) or it will not be able to guess any more parameters and it will terminate with FALSE.

How can we make an effective guess procedure? Weise's original algorithm checks zone closure directly by using operations on zones. Indeed, the formula in Definition 3.3.5 can be calculated by using only the future ($D^{\uparrow}$) and intersection ($D_1 \wedge D_2$) zone operations which enables the use of efficient data structures to represent zones such as DBMs. Another possibility is to represent the closure property on zones using the zones' characteristic formula and use an SMT solver:

$$
\begin{aligned}
& f(\Theta) \wedge \Omega \wedge \forall_{Clk_{PQ} \to \mathbb{R}_{\geq 0}} \Big[\big(D^{\uparrow}_{pq} \wedge Inv(l_p)\big) \to \big(D^{\uparrow}_{pq} \wedge Inv(l_{pq})\big) \\
& \wedge \quad \big(D^{\uparrow}_{pq} \wedge Inv(l_q)\big) \to \big(D^{\uparrow}_{pq} \wedge Inv(l_{pq})\big) \\
& \wedge \quad \bigwedge_{a \in \textsc{May}(Act_P \cup Act_Q)} \big(D^{\uparrow}_{pq} \wedge Inv(l_{pq}) \wedge \mathbf{G}_{PQ}(a, l_{pq})\big) \\
& \qquad \to \big(D^{\uparrow}_{pq} \wedge Inv(l_{pq}) \wedge \mathbf{G}_{Q}(a, l_{q})\big) \\
& \wedge \quad \bigwedge_{a \in \textsc{Must}(Act_P \cup Act_Q)} \big(D^{\uparrow}_{pq} \wedge Inv(l_{pq}) \wedge \mathbf{G}_{Q}(a, l_{q})\big) \\
& \qquad \to \big(D^{\uparrow}_{pq} \wedge Inv(l_{pq}) \wedge \mathbf{G}_{PQ}(a, l_{pq})\big) \Big]
\end{aligned}
$$

The above formula is satsifiable if the subformula contained by the quantifier (*i.e.,* the closure property) is *valid* with parameters assigned values satisfying $f(\Theta) \wedge \Omega$. Thus, any model returned by the SMT solver is a set of parameter valuations that make the zone closed.

Algorithm 3 eventually terminates assuming the oracle we use for guessing parametric assignments terminates: First, Weise's algorithm (Algorithm 4) terminates because the parametric zone-graph it explores is finite and it can only expand and remove a zone at most once. The outer guessing loop terminates (Algorithm 3) because the number of possible parameter valuations is finite and any valuation of the parameters is tried at most once.

Algorithm 3 decides modal refinement for action-deterministic TPMS given Lemma 3.3.2 and assuming Weise's orginal algorithm is correct. First, it is sound. If it returns true then by definition it has found a parameter assignment that makes the initial zone of the product zone graph closed. If it returns false, then my

Algorithm 4 Version of Weise's zone-based refinement checking algorithm modified to assist a parameter search.

Precondition: Two TPMS $\mathcal{P}$ & $\mathcal{Q}$ and a boolean formula Ω indicating invalid parameter valuations.

Postcondition: returns (TRUE, $f(\Theta)$) if $\mathcal{P} \sqsubseteq_{\sim} \mathcal{Q}$ for the given parameter valuation $f(\Theta)$, (FALSE, $f(\Theta)$) otherwise. If $\mathcal{P} \sqsubseteq \mathcal{Q}$ then $f(\Theta)$ are the supporting witness parameter valuations. If $\neg(\mathcal{P} \sqsubseteq_{\sim} \mathcal{Q})$ then $f(\Theta)$ are parameter valuations that don't support $\mathcal{P} \sqsubseteq_{\sim} \mathcal{Q}$.

```
function GUESSREFCHECK(P, Q, Ω)
    PQ ← SYNTACTICPRODUCT(P, Q)
    f(Θ) ← TRUEFORMULA()
    seen ← ∅,  check ← EMPTYSTACK()
    PUSH(check, ⟨l̄_pq, 0⟩)
    while NOTEMPTY(check) do
        ⟨l_pq, D_pq⟩ ← POP(check)
        if RELEVANT(⟨l_pq, D_pq⟩) then
            V ←CLOSEDPARAM(⟨l_pq, D_pq, f(Θ) ∧ Ω⟩)
            if V = FALSE then
                if ⟨l_pq, D_pq⟩ = ⟨l̄_pq, 0⟩ then
                    return (FALSE, f(Θ))
                end if
                REMOVE(⟨l_pq, D_pq⟩)
                PUSHALL(check, PRED(⟨l_pq, D_pq⟩)
            else
                f(Θ) ← f(Θ) ∧ V
            end if
        end if
        if ⟨l_pq, D_pq⟩ ∉ seen then
            seen ← seen ∪ ⟨l_pq, D_pq⟩
            PUSHALL(check, SUCC(⟨l_pq, D_pq⟩)
        end if
    end while
    return (TRUE, f(Θ))
end function
```

definition and Lemma 3.3.2 it could not find any parametric assignment that makes the initial zone closed.

3.4 Related Work

In [119] Larsen *et al.* first introduced what they called a *modal* process logic. Their modal process essentially consists of a LTS where each transition has both an action label and a *may/must modality*. The may-must modality constrains the behaviors exhibited by potential implementations: If a specification contains a must-transition then all implementations of that specification must have an equivalent transition. If a specification has a may-transition then an implementation is allowed to have an equivalent transition.

Recently there has been renewed interest in modal specifications as a formalism for software product lines [121, 66, 31]. In software product lines a *product line architect* designs a product line by working "top-down": They start with abstract models of the software components and then incrementally refine the abstract model into concrete implementations by either removing may transitions or turning may into must transitions. Much of the research efforts studying modal specifications for product lines has focused on creating a "complete" set of alebraic operators for the formalism in questions. These operators usually include *parallel composition*, *conjunction*, *disjunction* and *quotient*.

Parallel composition allows designers to combine two specification into a larger system where the two sub-specifications interact with each other. Conjunction allows designers to create a new specification that has the intersection of behaviors from the operands while disjunction is the union. Quotient gives the

"difference" between two specifications and is useful for synthesis.

There have been a number of interesting extensions to the basic idea of modal specifications. *Disjunctive* modal transition systems [31, 64] allow for specifiers to group may transitions into a disjunction. Then any valid refinement of the specification must have a transition equivalent to *at least one* transition in the disjunctive group. At some level, we have applied a similar idea in TPMS, where a parametric clock constraint induces a number of possible MTTS.

Quantitative Modal Transition Systems [29, 120] add ranges of numerical weights to the transitions of the specification. The weights can be used to denote a number of quantitative metrics, for example cost. One interesting class of quantitative model transition systems are timed modal transition systems. Here, instead of annotating edges with integer weights, an existing timed formalism, such as timed-automata ([5]), is extended with a notion of may/must modality.

The first appearance of a timed modal specification in the literature can be seen in [49] and Jens Godskesen's PhD dissertation [68]. While they give a process algebraic syntax for their specification the semantics of an implementation are fundamentally the same as a Timed Automata with location invariants. In this early work refinement of the timing behavior is allowed in a very restricted sense: A refinement must spend less than or equal time in a given state as the specification.

Later, [182] directly extends timed automata by allowing edges in the automata be either may or must. This means that while specifiers can express functional variability over possible refinements they cannot express timing variability separately: The semantics of refinement are given by applying the modality of the edges to the infinite transition system in the operational semantics of the au-

tomata. This means that if a specification has a must edge that is enabled between time t_1 and t_2, then any refinement must have an equivalent edge enabled for the same time duration: All refinements must preserve the time-non-determinism of the specification's must edges.

[35] establish the decidability of refinement and consistency (*i.e.,* does this specification have a refinement?) of timed modal specifications. The timed formalism Bertrand *et al.* studied coincides with the original formulation of timed automata which had accepting states (but no location invariants).

[56] extends timed-automata with may/must modality on the edges but with the extra restriction that all edges labeled with *input* events are must edges and all edges with output events are may edges. While this is a significant restriction it allows for game-theoretic characterization of their semantics which in turn reveals elegant algorithms for the various algebraic operators including quotient.

[34] studied another restricted form of timed modal specification where edges in event-clock automata are extended with modality. Event-clock automata are a restricted form of timed automata where there is a clock associated with each edge label. Whenever an edge is taken, its associated clock is reset.

In [32] Benes *et al.* proposes a unique modal-specification for real-time systems where timing and functional variability are seperate. Instead of using clocks and guards over clock values to model time, Benes *et al.* annotate the edges of the labeled transition system with durations. The durations indicate the range of time the source state of the edge is allowed to wait before an action is taken. Timing variability is achieved by indicating whether the durations are *controllable* or *uncontrollable*. Controllable durations admit implementations that use a duration equal or less than the duration. Implementations must keep the same durations for

all controllable durations. While [32] lets designers specify functional and timing variability separately, the formalism of [32] does not offer any parallel composition operator.

Each of the timed specification theories described bove either doesn't allow for timing variability separate from functional variability or lacks a parallel composition operator (unlike TPMS). The specification theories of [182, 35, 56] and [34] don't separate timing from functional variability. This means that any implementation of their specifications must preserve all the timing behavior (including non-determinism) of the specification. While [68] separates timing variability from functional variability its notion of refinement only allows for "faster" implementations (TPMS is more flexible and allows the designer to write specifications that allow, for example, not just faster but also more precise timing behavior). As far as we know only the specification theory of [32] explicitly allows for timing variability separate from functional variability. Unfortunately its lack of parallel composition makes it unsuitable for reasoning about on-demand systems.

Chapter 4

The On-Demand Systems Description Language

The design of the language used to specify the applications in an on-demand ecosystem will impact the viability of that ecosystem. The language will have a large role in determining what applications developers can express and verify, what applications the regulators will admit to the ecosystem, and what devices and algorithms users can combine into a functioning system.

In this chapter we propose the On-Demand Systems Description Language (ODSDL). The ODSDL would be used by application developers to program applications and to specify what devices that application needs to operate correctly. Device manufacturers would use the ODSDL to specify the behavior and capabilities of their device. This chapter is organized as follows: In Section 4.1 we lay down requirements that any language (or language framework) intended for on-demand systems must satisfy. In Section 4.2, we give a high-level overview ODSDL features and how those features help satisfy the previously described re-

quirements. In Section 4.3 we provide a sketch of the language itself and illustrate how it would be used to program an example application. In Section 4.4 we give denotational semantics for ODSDL programs and prove important determinism properties.

4.1 Key Language Requirements

A language intended to program on-demand applications (*i.e.,* define on-demand systems) must enable *predictability*, *compositional flexibility*, provide adequate amounts of *extensibility* and be *portable*. If not, the number and types of systems available for end-users to instantiate will be limited. If the number and types of systems are limited, then the value of the on-demand approach is practically diminished. In this section we will describe what we mean by predictability, compositional flexibility, and existensibility and how each impacts the systems ultimately available to end-users. For portability, we follow the generally accepted meaning (*e.g.,* as used in [144] or [81]).

Predictability

When an application developer creates an application they are effectively defining a set of systems users can create. Developers will not be able to test every combination of allowed devices and their application. Instead, developers will have to **predict** the possible behavior of these systems by analyzing the application itself (*i.e.,* treat the application as a model of the systems it defines). The detail at which system behavior can be predicted will affect what types of systems developers can get admitted to the ecosystem: The safety assesment of low(er)-criticality appli-

cations (*e.g.,* smart-alarms [170, 107, 109]) will likely not depend on the detailed behavior of the system while complex systems of higher-criticality (*e.g.,* closed-loop control of PCA) will require a detailed understanding of all the ways the system's software and devices can interact with each other and the environment (*i.e.,* patient).

The language used to define an on-demand system should include features that allow the developer to control the behavior of the system's software components and how those software components interact with each other and connected devices. Futhermore, the language must allow developers to model individual device behavior and extrapolate from the behavior models to the behavior of real systems.

Compositional Flexibility

The compositional flexibility of an on-demand ecosystem is measured by its ability of the end-users to (safely) use different combinations of devices with the same application. Compositional flexibility gives end-users more options in terms of how they allocate (*e.g.,* assign to patients) and procure devices, which in turn can help drive down costs [1].

The compositional flexibility of an on-demand ecosystem will be affected by the ecosystem's standard device behavior language and the ecosystem's notion of application/device compatibility. Superficially, compositional flexibility and predictability seem to be at odds: Strict definitions of compatibility limit the number of devices that can be used with a given application which in turn makes predicting overall system behavior "easier". Conversely, permissive definitions of compatibility mean more devices can be used with a given application but then

the application developer must account for more possible behaviors. Ideally, the ecosystem should standardize on a language that lets application developers write permissive requirements when possible, and strict requirements when necessary.

Extensibility

As discussed in the intro to this dissertation, creating large complex standards is practically difficult. Instead of developing a large complex standard that defines the behavior of each device type, a better approach would be to use a standardized *extensible language* to describe device behavior. Ideally, the language would be relatively simple yet expressive enough for application developers and device manufacturers to specify the behavior of new devices at whatever level of detail is needed. The main benefit of using a standard extensible language is less standardization effort. The ecosystem stakeholders only needs to come to a consensus on the language itself. There would no need for the ecosystem consortium to decide the standard behavior for each device type. If a device manufacturer wants to produce a new device, there is no need for them to conform to a standard for that type.

Extensibility frees the manufacturer to add new capability to the ecosystem (via devices with new behavior). Likewise, application developers would not be restricted rely one the behaviors defined by any standard. If they want to develop a new application that requires new behavior out of a device they are free to do so: They can partner with a device manufacturer to ensure that a device with that behavior is available in the ecosystem. Then, if that application is successful, other device manufacturers may decide to market devices that also offer that behavior. In this way, a standard extensible language for device behavior would let

the ecosystem grow organically with a minimum of standardization effort.

4.2 Key Language Features

The ODSDL addresses predictability, compositional flexibility, and extensibility by incorporating (and tweaking) a number of previously studied concepts into a single language. These concepts include the ability to specify the *logical architecture* of a system, write programs with logical execution time semantics, and use a type of modal specification to specify device behavior. While there are three concepts and three key language requirements there is not a direct mapping between these concepts and the requirements. Instead, each of these concepts helps the ODSDL meet all of the requirements.

Logical Architecture Abstraction

Inspired by the AADL [65] and MetaH [175], the ODSDL gives application developers an abstraction by which they specify a system's logical architecture. A logical architecture specification is a description of a system's functional components (*e.g.,* software modules, medical devices, *etc.*), how information flows between those components, and abstractions that capture the "non-functional" properties (*e.g.,* timing) of those components. The logical architecture abstraction allows application developers to express different systems without specifying particular hardware or hardware configurations which is necessary for application portability.

Logical Execution Time Semantics & Input Determinism

The ODSDL extends Logical Execution Time (LET) semantics [80, 79] to systems of (device/software) components connected via publish/subscribe [63] communications. In LET semantics each communication or computation is specified with a delay d, or *logical execution time*. Each time a computation or communication activates, it reads its inputs and does its computation in zero time, then writes its outputs in zero time after d timeunits have elapsed [110]. LET semantics support the portabilty of applications by abstracting away timing variabilities in the execution of an induced by a platform's underlying scheduler, current system load, or the speed of the processor (or network).

In addition to having LET timing semantics, the value-semantics (*i.e.,* the semantics of the computations themselves) in the ODSDL are deterministic: For a given state and input, an ODSDL task will always compute the same result. Deterministic value-semantics, combined with the LET semantics, ensure that ODSDL applications are *input-stream deterministic*: For a given stream of (timed) input events (*e.g.,* sensor readings from devices), an application will always generate the same stream of (timed) outputs (*e.g.,* actuator commands). Input stream determinism directly supports predictability because if developers verify & validate the application using testing, they have assurance that the system will behave the same in the field (at least for the tested inputs). Furthermore, input-sequence determinism can aid automatic and exhaustive verification techniques: For example, a model-checker would not have to explore as many possible system executions or event interleavings [67].

Modal Specifications for Device Behavior

The ODSDL bases its device behavior specification sub-language on Time Parametric Modal Specifications (TPMS) (see Chapter 3) and uses *weak-implementation* (see Definition 3.2.14) to define compatibility between applications and devices. TPMS (and weak-implementation) directly supports predictability, compositional-flexibility, and extensibility.

TPMS supports predictability by allowing the application developer to explicitly specify device behavior requirements and then directly analyze the composition of the application, device requirements specifications, suitable environment model for safety: First, the developer would implement the application software, specify its logical architecture, and declare the device behavior requirements. A TIOA (a TPMS with no modal variabilty) model of the application software can then be extracted from the implementation and composed with the device requirements TPMS and environment models. That composition can be fed into a model-checker or other tool to verify that is satisfies the correct safety properties. If the platforms use the weak-implementation relation to check for compatibility then Theorems 3.2.3, 3.2.5, and 3.2.13 ensure us that all instantiated systems will satisfy those same safety properties.

TPMS allow for compositional flexibility because they allow developers to directly control how permissive their specifications are by distinguishing between *required* and *allowed* behavior. As their name suggests, required behaviors must be present in all devices that satisfy a specification. Conversely, a device cannot satisfy a specification if it includes more behavior that what is allowed. Application developers can increase the permissiveness (and hence make their application compatible with a wider range of devices) of their device requirements specifi-

cations by adding allowed behavior or reducing required behavior. Furthermore, TPMS allow developers to start with permissive specifications and then iteratively refine them (*e.g.,* following the definition of weak-refinement) until they arrive at a specififcation for which they can prove overall system safety.

TPMS support extensibility and enables small standards. If TPMS are used by application developers and device manufacturers to describe behavior, the ecosystem consortium will not have to standardize behavior for different device types. As long as the consortium standardizes the syntax and the meaning of the action-labels (*i.e.,* the physical phenomenon each action-label represents) then the platforms will be able to automatically determine if a device's behavior meets the needs of an application. Furthermore, if the ecosystem consortium chooses to standardize basic behavior for each device type, they can do so with a "loose" specification which may be easier to achieve consensus on. Then, if necessary, different stakeholders can refine the specification as needed via modal-refinement (analogous to a programmer extending an abstraction).

4.3 Core Language Elements

The ODSDL provides three core language elements: *system declarations*, *module declarations* and *device declarations*. All three are needed to define a complete ODSDL application. Device declarations would also be used standalone by device manufacturers to specify their device's behavior.

In this section we will illustrate the use of each of these elements by showing how they would be used to implement a PCA safety interlock, *i.e.,* an application that automatically disables a PCA pump if the patient is at risk of overseda-

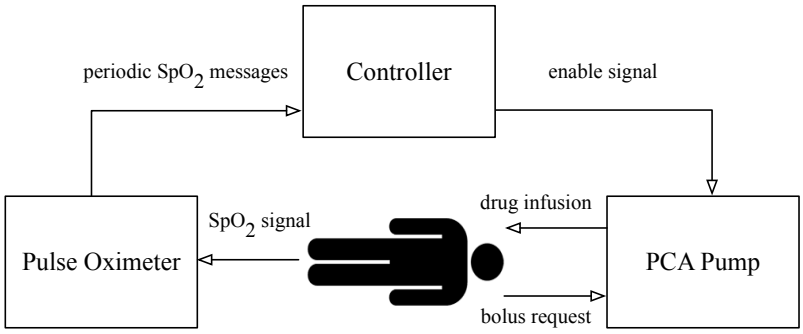


Figure 4.1: Logical architecture of a closed-loop PCA safety interlock.

tion (Figure 4.1). The application uses two medical devices: a pulseoximeter and PCA pump. The pulseoximeter periodically samples the patient's blood oxygenation (SpO_2) and then transmits those values to the controller. The controller is a software component that decides, based on received SpO_2 values, whether or not the patient is at risk for oversedation. If the controller deems the patient at risk, then it will send a 'disable' message to the pump which will cause the pump to ignore future bolus requests from the patient.

The PCA safety interlock example will help us show why the different ODSDL language features are needed and how they are used to define an on-demand system. First, The PCA safety interlock is a real-time system and the application developer must ensure timely delivery of the message which will in turn depend on them ensuring timing characteristics of the whole control loop (sensor sampling delays, network transmission, and controller computation). Second, the safety of of the PCA safety interlock is also clearly dependent how how the devices themselves behave but tolerates some variability in behavior: For example, its perfectly safe to use a pump that disables itself autonomously as long as it disables itself

when told to by the controller. Third, both devices ahve input/output interactions with both the environment and network which lets us demonstrate how to define device behavior in terms of basic physical signal and data-types.

4.3.1 System Declarations

$$
\begin{array}{rcl}
\langle\text{system declaration}\rangle & \models & \texttt{system}\ \langle\text{identifier}\rangle\{\ \langle\text{body}\rangle\ \} \\
\langle\text{body}\rangle & \models & \langle\text{components}\rangle\ \langle\text{interactions}\rangle \\
\langle\text{components}\rangle & \models & \langle\text{device}\rangle\texttt{;}\ \langle\text{components}\rangle\ \mid\ \langle\text{module}\rangle\texttt{;}\ \langle\text{components}\rangle\ \mid\ \epsilon \\
\langle\text{device}\rangle & \models & \texttt{device}\ \langle\text{identifier}\rangle\ \texttt{:}\langle\text{type identifier}\rangle \\
\langle\text{module}\rangle & \models & \texttt{module}\ \langle\text{identifier}\rangle\ \texttt{:}\langle\text{type identifier}\rangle \\
\langle\text{interactions}\rangle & \models & \texttt{interactions}\ \{\ \langle\text{interaction body}\rangle\ \}\ \mid\ \epsilon \\
\langle\text{interaction body}\rangle & \models & \langle\text{interaction}\rangle\texttt{;}\langle\text{interaction body}\rangle\ \mid\ \epsilon \\
\langle\text{interaction}\rangle & \models & \langle\text{identifier}\rangle\texttt{.}\langle\text{identifier}\rangle\ \texttt{flowsto}\ \langle\text{identifier}\rangle\texttt{.}\langle\text{identifier}\rangle\ \langle\text{qos clause}\rangle \\
\langle\text{qos clause}\rangle & \models & \texttt{delay}\ \langle\text{number}\rangle\ \mid\ \epsilon
\end{array}
$$

Figure 4.2: Grammar for ODSDL system declarations

ODSDL system declarations are used by application developers to specify the logical architecture of the system managed by their application: They specify the types of devices their application needs, the software modules that are used, and the possible flows of data and commands between devices and modules. The grammar for system declarations is included in Figure 4.2.

System declarations always have two parts. In the first, needed devices and software modules are declared. The declaration always associates an *instance* with a *type*. This allows systems to use more than one of the same device or mod-

```
1 system ClosedLoopPCA {
2        device pca : PCAPump;
3        device po : PulseOximeter;
4        module controller : PCAController;
5        interactions{
6            controller.disable_cmd flowsto pca.disable delay 100;
7            po.spo2 flowsto controller.spo2_in delay 100;
8        }
9 }
```

Figure 4.3: Top level specification for the closed-loop PCA system

ule type. The module or device type name is always a reference to an ODSDL (device or module) declaration of the same name. The second part of the declaration is called the *interactions* block. Interactions are *dataflows* that specify the publish/subscribe relationships that carry data and commands between module and device ports. Each flow is always declared by identifying the source module instance/port and destination module instance/port. Each input port of a device or module can have **at most one** flow coming into it (this restriction simplifies the input-deterministic semantics of the system). There is no restriction on the numbe of flows leaving an output port. Flows have a single QoS parameter called *delay*. The delay sets how long the platform will take to propagate the data or command from the source to the destination in milliseconds. The delay is **not** a deadline: it is an isochronal parameter meaning the message will arrive exactely after the delay has elapsed.

The system declaration for the example closed-loop PCA system is shown in Figure 4.3. The declaration lists the sub-components of the system and how they will exchange data and command . Lines 2 & 3 declare that the system

will make use of two external devices; one with type `PCAPump` the other with type `PulseOximeter`. Line 4 declares that one application module will be used (of type `PCAController`). These modules are software units will execute on the MAP. The types for both modules and devices are both references to the appropriate module or device declarations.

The closed-loop PCA system has two dataflows: The pulse-oximeter will periodically publish sampled SpO_2 values to the controller, which may cause the controller to issue a command to disable the pump. Note that the interactions block only consists of interactions between the devices and modules and not the devices and the physical environment (*e.g.,* the SpO_2 signal, infusion rate, and bolus event from Figure 4.1). Instead, device interactions with the physical environment will be specifed in the device declarations themselves. Each dataflow in the closed-loop PCA system has a `delay` of 100 which means that if the pulse-oximeter sends a message at time t it will arrive at the input port of the controller at $t + 100$ (likewise and command issued by the controller will arrive at the PCA pump 100 milliseconds later.).

4.3.2 Module Declarations

Modules are the software units that implement the control/coordination algorithms of an ODSDL application. Modules consist of input/output ports, state variables, and a task.

The ports of a module are its logical interface to the rest of the system. Each port is specified as either *input* (*i.e.,* a destination for a dataflow) or an *output* (source of a dataflow). The datatype of all ports is implicitly a long (*i.e.,* a signed, 64-bit twos-complement integer). Each port is annotated with a `minsep` parame-

$$
\begin{array}{rcl}
\langle\text{module}\rangle & \models & \texttt{module}\ \{\ \langle\text{module body}\rangle\ \} \\
\langle\text{module body}\rangle & \models & \langle\text{signature}\rangle\ \langle\text{globals}\rangle\ \langle\text{task spec}\rangle \\
\langle\text{signature}\rangle & \models & \texttt{net input}\ \langle\text{port declaration}\rangle\ \langle\text{signature}\rangle \\
 & & \mid\ \texttt{net output}\ \langle\text{port declaration}\rangle\ \langle\text{signature}\rangle \\
\langle\text{port declaration}\rangle & \models & \langle\text{identifier}\rangle\ \langle\text{identifier}\rangle\ \langle\text{port timing}\rangle \\
\langle\text{port timing}\rangle & \models & \texttt{minsep}\ n_1\ \texttt{maxsep}\ n_2\ \mid\ \epsilon \\
\langle\text{vars}\rangle & \models & \texttt{vars}\ \{\ \langle\text{var list}\rangle\ \} \\
\langle\text{var list}\rangle & \models & \langle\text{var}\rangle\texttt{,}\ \langle\text{var list}\rangle\ \mid\ \epsilon \\
\langle\text{var}\rangle & \models & \langle\text{identifier}\rangle\texttt{:}\langle\text{identifier}\rangle \\
\langle\text{task spec}\rangle & \models & \texttt{task activated}\ \langle\text{trigger}\rangle \\
 & & \texttt{delay}\ n\ \{\ \langle\text{task body}\rangle\ \} \\
\langle\text{trigger}\rangle & \models & \texttt{periodically}\ n\ \mid\ \texttt{by port}\ \langle\text{identifier}\rangle \\
\langle\text{task body}\rangle & \models & \langle\text{locals}\rangle\ \langle\text{stmt}\rangle \\
\langle\text{stmt}\rangle & \models & \langle\text{identifier}\rangle\ \texttt{:=}\ \langle\text{axepr}\rangle\ \mid\ \langle\text{stmt}\rangle\texttt{;}\langle\text{stmt}\rangle\ \mid\ \texttt{if}\ \langle\text{bexpr}\rangle\ \texttt{then}\ \langle\text{stmt}\rangle\ \texttt{else}\ \langle\text{stmt}\rangle \\
 & & \mid\ \texttt{while}\ \langle\text{bexpr}\rangle\ \texttt{do}\ \langle\text{stmt}\rangle \\
\langle\text{send}\rangle & \models & \texttt{send}\texttt{(}\langle\text{identifier}\rangle\texttt{,}\langle\text{aexpr}\rangle\texttt{)} \\
\langle\text{aexpr}\rangle & \models & n\ \mid\ \langle\text{identifier}\rangle\ \mid\ \langle\text{aexpr}\rangle\texttt{+}\langle\text{aexpr}\rangle\ \mid\ \langle\text{aexpr}\rangle\texttt{-}\langle\text{aexpr}\rangle\ \mid\ \langle\text{aexpr}\rangle\texttt{*}\langle\text{aexpr}\rangle \\
\langle\text{bexpr}\rangle & \models & \texttt{true}\ \mid\ \texttt{false}\ \mid\ \langle\text{aexp}\rangle\ =\ \langle\text{aexp}\rangle\ \mid\ \langle\text{aexp}\rangle\ \leq\ \langle\text{aexp}\rangle \\
 & & \mid\ \langle\text{aexp}\rangle\ \texttt{||}\ \langle\text{aexp}\rangle\ \mid\ \langle\text{aexp}\rangle\ \texttt{\&\&}\ \langle\text{aexp}\rangle
\end{array}
$$

Figure 4.4: Grammar for ODSDL module descriptions

ter which declares the minimum separation between message arrivals on that port in milliseconds. Minimum separation informs the platform scheduler on how often it might need to dispatch a task (if the module's task is dispatched whenever a new value arrives on the port). Input ports contain the last value received on that port. Tasks can read (but not write) input ports as if the input ports are variables.

Each module has a single task which can be programmed to read data from

input ports, read and update state variables, and send data out the output ports. The programmer has two options for how the task is dispatched: The module can be dispatched periodically by the platform's scheduler or when a new value arrives on a particular port. Dispatch can be triggered by only one port. Programmers must specify a `delay` (in milliseconds) for each module task. The delay must be less than the task's period. A task's period is either specified directly (in the case of periodically dispatched tasks) or the task inherits its period from the `minsep` of the port that triggers it. Tasks can send data using the `send` operator. Tasks are restricted from sending more than once out of the same port in the same execution: the last sent value is what will be used. This restriction makes both resource allocation easier for the plaform[1].

Module tasks have LET semantics: When they are dispatched their execution is "instantaneous" but all `send` events they generate will not happen until after they task delay has elapsed. If a module receives values on a port at the same time a task is to be dispatched, the port values are updated before the task is dispatched. The LET semantics combined with the input port write/task dispatch ordering ensures that modules are input deterministic: For a given stream of inputs on the input ports, the module will always produce the same output stream on its output ports.

Figure 4.5 shows the ODSDL module declaration for the `PCAController`. The controller receives SpO_2 measurements on its input port and generates disable commands on its output port. The module has a single state variable, `last_spo2`

[1]If we dropped this restriction then we would have to do static analysis of the task itself to figure out how many times it could send out a port in the worst case. Furthermore, sending more than once out of the same port at the same time instant means we would have to decide which value the receiving port accepts.

```
1 module PCAController {
2   net input "spo2_in" Integer minsep 200 maxsep 700
3   net output "disable_cmd" Event minsep 100 maxsep inf
4   vars { }
5   task activated periodically 200  delay 100{
6       if(spo2_in < 90){
7         send("disable_cmd", "")
8       }
9       else{ }
10     }
11   }
12 }
```

Figure 4.5: PCA Controller module declaration

which it uses to record the most recently received SpO_2 reading. The process of this module is very simple; it is activated periodically and samples the most recently received value on the `spo2_in` port which it copies into the `last_spo2` state variable. If the SpO_2 value is below some threshold it will send a disable signal to the PCA pump. If a task activation is scheduled for the same instant as when `spo2_in` receives an update the port will always be updated before task activation. Because the task's `delay` is 100, if it is dispatched at time t and if it decides to send the disable command, the command will be sent at time $t + 100$.

4.3.3 Device Declarations

Device declarations in the ODSDL have two uses. First, they are used by application developers to specify the required and allowed behavior of the devices that are used with their application. Second, device manufacturers use device declarations to model the behavior of their devices. If the certification authority certifies

```
⟨device declaration⟩ ⊨ device ⟨identifier⟩{ ⟨body⟩ }
⟨body⟩ ⊨ ⟨signature⟩ ⟨clocks⟩ ⟨parameters⟩ ⟨constraints⟩ ⟨vars⟩ ⟨invariants⟩ ⟨init⟩
         ⟨behavior⟩
⟨signature⟩ ⊨ net input ⟨port declaration⟩ ⟨signature⟩
            | net output ⟨port declaration⟩ ⟨signature⟩
            | sensor ⟨identifier⟩
            | actuator ⟨identifier⟩ | ϵ
⟨port declaration⟩ ⊨ ⟨identifier⟩ datatype ⟨identifier⟩ ⟨timing⟩
⟨timing⟩ ⊨ minsep ⟨number⟩ maxsep ⟨number⟩ | ϵ
⟨clocks⟩ ⊨ clocks { ⟨identifier list⟩ }
⟨invariants⟩ ⊨ invariants { ⟨inv list⟩ }
⟨inv list⟩ ⊨ (⟨expr⟩,⟨expr⟩),⟨inv list⟩ | ϵ
⟨parameters⟩ ⊨ parameters { ⟨identifier list⟩ }
⟨constraints⟩ ⊨ constraints { ⟨expr⟩ }
⟨vars⟩ ⊨ vars { ⟨var list⟩ }
⟨var list⟩ ⊨ ⟨var⟩, ⟨var list⟩ | ϵ
⟨var⟩ ⊨ ⟨identifier⟩:(⟨number⟩ ⟨number⟩) | ⟨identifier⟩:(⟨string list⟩)
⟨init⟩ ⊨ init ( ⟨expr⟩ )
⟨behavior⟩ ⊨ behavior ( ⟨behaviors⟩ )
⟨behaviors⟩ ⊨ ⟨guarded update⟩, ⟨behaviors⟩ | ϵ
⟨guarded update⟩ ⊨ when ⟨expr⟩ andsync ⟨action⟩ then ⟨modality⟩ do ⟨cexpr⟩
⟨action⟩ ⊨ ⟨identifier⟩! | ⟨identifier⟩? | ⟨identifier⟩!⟨identifier⟩
          | ⟨identifier⟩?⟨identifier⟩
⟨modality⟩ ⊨ may | must
```

Figure 4.6: Grammar for ODSDL device declarations

that the device implements the modeled behavior, the device declaration would be embedded into the device itself. Then, each time the device is connected to a platform it would register its device declaration with the platform where it can be used for device/application compatibility checking. Device declarations are a syntactic sugar for TPMS introduced in Chapter 3. The grammar for device declarations is given in Figure 4.6.

Like module declarations, device declarations begin with a declaration of ports. Unlike module declarations, device declarations have four types of ports: `net input`, `net output`, `sensor`, and `actuator`. The `net input` and `net output` port types are the same as in module declarations: They declare the logical interface the device exposes to the application over the network. The `sensor` and `actuator` port types declare the device's logical interface to its physical environment. Sensors represent inputs from the environment while actuators are treated as outputs. Unlike the network ports, sensors and actuators must be specified with a *physical type*. The physical types give the interpretation of a signal on one of those ports. The physical types, their interpretation, units, and allowed value ranges would be standardized by the ecosystem consortium.

The (required & allowed) behavior of devices are modeled using a guarded command language version of TPMSs which allows developers to expression both timing and functional variability in satisfying devices. Parameters, constraints on parameters, clocks, and invariants are the same as in TPMSs. Device declarations can have state variables and a valuation of the state variables correspond to a location in TPMS. Programmers write transitions using guarded commands. Programmers express each transition as a predicate (guard) over device state (clocks & state variables), synchronization actions over the device's ports, may/must modal-

ity, and updates to the device state: When the guard is true the transition is enabled and the updates happen in "0-time".

Synchronization with ports is expressed using standard CSP [85] synchronization syntax with data: `p ! v` denotes sending value (or value of variable) v out port p. `p ? v` means receiving a value over port p into variable v. TPMS do not explicity sending data as part of synchronization. ODSDL desugars into TPMS by creating a special action label for each value that can be sent out of each port (*i.e.,* `p ! 1` in a device declaration would translate into $p_1!$ in the corresponding TPMS).

We chose not to restrict the language in a way to prevent modeling errors such as deadlocks. For example, it is possible to specify a device that is not ready to receive a signal from the application when the application sends it, which will result in a deadlock. This is a tradeoff: Fewer restrictions make it easier for the programmer to express certain types of behavior, but the programmer is responsible for ensuring that their specifications don't deadlock when composed. In practice this may not be a significant issue as the model checker we developed for TPMS can automatically check for deadlock freedom.

Figures 4.7 and 4.8 show the device requirements specification of the closed-loop PCA application. The `PulseOximeter` samples SpO_2 from the environment every 500 to 600 milliseconds then it emits the sampled data out of its `spo2` port. Note how the constraints on the parameters allow for timing variability: This specification allows for devices that are imprecise (*e.g.,* non-deterministically sample between every 500 to 600 milliseconds) or precise (*e.g.,* samples every t milliseconds where t is fixed and constrained by $500 \leq t \leq 600$).

The `PCAPump` can receive a "disable" command from the network and a bolus

```
1 device PCAPump {
2   net input "disable" Event minsep 10 maxsep infinity
3   sensor "breq" physicaltype BolusRequest
4   actuator "rate" physicaltype FlowRateMlHour
5   clocks{ x }
6   parameters{ alpha, beta, gamma }
7   constraints{ alpha <= beta & beta <= 3 & gamma <= 4 }
8   vars{
9     loc : ("idle", "starting", "disabled", "infusing")
10  }
11  invariants{
12    (loc = "starting", x <= beta),
13    (loc = "infusing", x <= gamma)
14  }
15  init{ loc = "idle", FlowRateMlHour = 0 }
16  behavior {
17    when (loc = "idle") andsync ("breq") then must do (loc := "starting", x
        := 0),
18    when (loc = "idle") andsync ("disable") then must do (loc := "disabled",
        x := 0),
19    when (loc = "starting" & alpha <= x) andsync ("rate" ! 10)  then must do
        (loc := "infusing", x := 0),
20    when (loc = "starting") andsync ("disable") then must do (loc := "
        disabled", x := 0),
21    when (loc = "starting") andsync tau then may do (loc := "disabled", x :=
        0),
22    when (loc = "infusing" & gamma = x) andsync ("rate" ! 0)  then must do (
        loc := "idle"),
23    when (loc = "disabled") andsync ("disable") then must do (loc := "
        disabled", x := 0),
24    when (loc = "disabled") andsync tau then may do (loc := "idle")
25  }
26 }
```

Figure 4.7: Requirements on PCA pump interfacing and behavior

```
1 device PulseOximeter {
2   net output "spo2" Integer
3   sensor "pspo2"  physicaltype PercentBloodOxygenSaturation
4   clocks{ p }
5   parameters{ alpha, beta }
6   constraints{ 500 <= alpha & alpha <= beta & beta <= 600 }
7   vars{
8     loc : ("sensing", "sending"),
9     lastval : Integer[0 .. 100]
10  }
11  invariants{
12    (loc = "sensing", x <= beta),
13    (loc = "sending", x <= 0)
14  }
15  init{ loc = "sensing" & lastval = 0 }
16  behavior {
17    when (loc = "sensing" & x >= alpha) andsync ("pspo2" ? lastval) then must
          do (loc := "sending", x := 0),
18    when (loc = "sensing") andsync ("spo2" ! lastval) then must do (loc := "
         sensing", x := 0)
19  }
20 }
```

Figure 4.8: Requirements on Pulse-Oximeter interfacing and behavior

request from its environment while it has an "infusionrate" port that outputs to the environment. Note how the behavior specification gives the interpretation of these commands. When the pump receives a "disable" command it sets the infusion rate to 0 and moves into a state where it ignores subsequent bolus requests (line 24). When the pump gets a bolus request it implements the bolus by increasing the infusion rate for a bounded amount of time then it sets the rate back to 0. Note how the `PCAPump` declaration expresses *functional* variability: The behavior on line 27 is a *may* behavior. The pump may decide to abort a bolus for some internal

reason (*e.g.,* due to some technical alert). However, if a pump does not exhibit this behavior then it might still satisfy this declaration.

4.4 Semantics

In this section we give the semantics of ODSDL *programs* and prove that ODSDL programs are input-deterministic. By program, we mean the parts of an ODSDL specification whose execution is controlled by the platform, *i.e.,* the software modules and the dataflows. Logically, the boundry of an ODSDL program is at the input/output ports of devices: The output ports of the devices are the inputs to the program, while the input ports of the devices are the outputs of the program.

4.4.1 Preliminaries

We define the semantics of ODSDL modules and dataflows denotationally over streams of time-stamped event sets. Definition 4.4.1 gives the three types of events a module or dataflow can create and/or react to: Modules and devices can send values out of their output ports, modules and devices can receive values on input ports. A dataflow can get send events on its input which are converted to receive events on its output. Only modules can get schedule events on their inputs, which denote the dispatch of the module task.

Definition 4.4.1 (Event)**.** An event is one of $\mathsf{send}\,(m, p, v)$, $\mathsf{recv}\,(m, p, v)$, $\mathsf{sched}\,(m)$ or $\mathsf{update}\,(m, x, n)$.

- $\mathsf{send}\,(m, p, v)$ denotes module m sending value v out port p.
- $\mathsf{recv}\,(m, p, v)$ denotes module m receiving value v on port p.

- $\mathsf{sched}\,(m)$ denotes the dispatch of module m's task.

A stream is an infinite sequence. Our semantics require two types of streams (event-streams and state-streams) so we first give a generic definition of streams (Definition 4.4.2).

Definition 4.4.2 (A-Stream)**.** An A stream is an infinite sequence of elements from the set A. A-streams are co-inductively defined: If $a \in A$ and s is an A-stream then $a :: s$ is an A-stream. An A-stream can be destructed using the head (hd) and tail (tl) functions:

- $\mathsf{hd}\,(a :: s) = a$.
- $\mathsf{tl}\,(a :: s) = s$.

Timed-event streams are an infinite sequence of event sets each associated with a timestamp (Definition 4.4.6). Each element of the stream is a tuple that denotes what events happen when. For example the tuple $(\{\mathsf{recv}\,(m, p_1, 1)\,, \mathsf{recv}\,(m, p_2, 2)\}, t)$ means that module m receives 1 on port p_1 and 2 on port p_2 at time t. Because the events that happen at time t are collected into a set, there is no ordering condition between events; they all happen "simultaneously" at t. Note that the well-formedness condition ensures that time increases discretely and uniformly.

Definition 4.4.3 (Timed-Event Stream)**.** A timed-event stream is a stream where each element is a tuple (E, t) where E is a set of events that occur at time t. Timed-Event Streams must be well-formed. A stream s is well-formed if both:

1. $\mathsf{hd}\,(s) = (E, t)$ and $\mathsf{hd}\,(\mathsf{tl}\,(s)) = (E', t + 1)$.

2. $\mathsf{tl}\,(s)$ is well-formed.

If s is a timed-event stream then $s\,(t)$ gives the set of events that happen at time t: Let $\mathsf{hd}\,(s) = (E, t_i)$. If $t_i = t$ then $s\,(t) = E$ else $s\,(t) = \mathsf{tl}\,(s)\,(t)$.

The composition process used to define program semantics requires us to "merge" two or more event streams into a single stream. Definition 4.4.4 gives a precise definition of a function, merge, that takes two or more streams and merges them. Informally, two or more streams are merged by constructing a new stream whose event-set at each timestamp is the union of all the event-sets from the argument streams at that same timestamp.

Definition 4.4.4 (Stream Merge)**.** $\mathsf{merge} : ES \times ES \rightarrow ES$ is a function that takes two timed-event streams s_1 and s_2 and gives a new timed-event stream s_3 where $s_3\,(t) = s_1\,(t) \cup s_2\,(t)$. Formally merge is defined co-inductively: Let $(E_1, t) = \mathsf{hd}\,(s_1)$ and $(E_2, t) = \mathsf{hd}\,(s_2)$. Then $\mathsf{merge}\,(s_1, s_2) = (E_1 \cup E_2, t) :: \mathsf{merge}\,(\mathsf{tl}\,(s_1), \mathsf{tl}\,(s_2))$. Note that merge is only defined for streams that are well-formed and have the same initial timestamp. For convenience we often want to merge more than two streams at once. The above definition can be generalized to $n > 2$ arguments by iteratively merging pairs:

$$\mathsf{merge}\,(s_1, \ldots, s_n) = \mathsf{merge}\,(\ldots \mathsf{merge}\,(\ldots, s_{n-1}), s_n)$$

ODSDL modules carry state. We call a snapshot of the module's state a configuration (Definition 4.4.5). Module variables include both programmer defined variables and the module's input ports (the input ports hold the last value they received).

Definition 4.4.5 (Module Configuration)**.** The configuration of a module is given by Ω, which is a map of variables to values. If v is a variable, then $\Omega(v)$ is the value of v in Ω.

Our semantics will relate module configurations to time. We do this via timed-state streams (Definition 4.4.6).

Definition 4.4.6 (Timed-State Stream)**.** A timed-state stream is a stream where each element is a tuple (Ω, t) where Ω is a module configuration at time t. Timed-State Streams must also be well-formed. A stream s is well-formed if both:

1. $\mathsf{hd}(s) = (\Omega, t)$ and $\mathsf{hd}(\mathsf{tl}(s)) = (\Omega', t+1)$
2. $\mathsf{tl}(s)$ is well-formed.

4.4.2 Tasks

The a module's task in the ODSDL is written in a language based on the simple language IMP [185]. Unsuprisingly, the operational semantics of ODSDL tasks largely mirrors the semantics of IMP a student would learn in their undegraduate programming languages course (the operational semantics of ODSDL tasks are given in Figure 4.9). The main difference from IMP is the addition of the `send` construct. By definition, a module can't send more than one value out the same output at the same time. Rules *Send1* and *Send2* ensure that only the last value passed to a `send` call is actually sent: The configuration of a task is given by the triple $\langle S, \Omega, \delta \rangle$, where S is a program statement, Ω maps program variables to values and δ is the set of $\mathsf{send}(m, p, v)$ events corresponding to the values the module will send as a result of its task's execution.

Definition 4.4.7 (Task Evaluation)**.** If a module m's task final configuration when evaluated in state Ω is $\langle S, \Omega', \delta \rangle$, then $\mathsf{eval}\,(m, \Omega) = (\delta, \Omega')$. The rules for task evaluation are given in Figure 4.9. By convention a is used to denote an arithmatic expression and b is used to denote a boolean expression.

4.4.3 Modules, Dataflows and Programs

Each dataflow is an ODSDL program is a simple stream processing process that converts input send events into output receive events timeshifted by the delay of that dataflow:

Definition 4.4.8 (Semantics of Dataflows)**.** For a dataflow connector $c = i.j$ **flowsto** $k.l$ **delay** d its semantics are defined by the greatest binary relation R_c between c's input timed-event streams and its output timed-event streams such that if $(s_{in}, s_{out}) \in R_c$ then $(E, t) = \mathsf{hd}\,(s_{in})$, $(E', t + d) = \mathsf{hd}\,(s_{out})$ and $E' = \{\mathsf{recv}\,(k, l, v) \mid \mathsf{send}\,(i, j, v) \in E\}$ and $(\mathsf{tl}\,(s_{in}), \mathsf{tl}\,(s_{out})) \in R_c$.

Dataflows are clearly input-deterministic:

Lemma 4.4.1 (Dataflows are Input Deterministic)**.** For a dataflow conntector c let R_c be defined as in Definition 4.4.8. If $(s_{in}, s^1_{out}) \in R_c$ and $(s_{in}, s^2_{out}) \in R_c$ then $s^1_{out} = s^2_{out}$.

Proof. We proceed by *co-induction* on the streams. Let $(E_{in}, t) = \mathsf{hd}\,(s_{in})$. Then there is only one possibilty for $\mathsf{hd}\,(s_{out})$ which is $(E', t + d)$ with $E' = \{\mathsf{recv}\,(k, l, v) \mid \mathsf{send}\,(i, j, v) \in E\}$ (when c connects $i.j$ to $k.l$). We conclude the proof by applying the *co-induction* hypothesis to show that if

$$\left((E_{in}, t) :: \mathsf{tl}\,(s_{in}), (E', t) :: \mathsf{tl}\left(s^1_{out}\right)\right) \in R_c$$

$$\text{Lookup}\ \frac{\Omega\,(x) = n}{\langle x, \Omega, \delta\rangle \rightarrow \langle n, \Omega, \delta\rangle}$$

$$\text{Sum}\ \frac{n_3 = n_1 + n_2}{\langle n_1 + n_2, \Omega, \delta\rangle \rightarrow \langle n_3, \Omega, \delta\rangle}$$

$$\text{Add1}\ \frac{\langle a_1, \Omega, \delta\rangle \rightarrow \langle a_1', \Omega, \delta\rangle}{\langle a_1 + a_2, \Omega, \delta\rangle \rightarrow \langle a_1' + a_2, \Omega, \delta\rangle} \quad \text{Add2}\ \frac{\langle a_2, \Omega, \delta\rangle \rightarrow \langle a_2', \Omega, \delta\rangle}{\langle n + a_2, \Omega, \delta\rangle \rightarrow \langle n + a_2', \Omega, \delta\rangle}$$

$$\text{Sequence1}\ \frac{\langle s_0, \Omega, \delta\rangle \rightarrow \langle s_0', \Omega, \delta\rangle}{\langle s_0; s_1, \Omega, \delta\rangle \rightarrow \langle s_0'; s_1, \Omega, \delta\rangle} \quad \text{Sequence2}\ \frac{}{\langle \{\}; s_1, \Omega, \delta\rangle \rightarrow \langle s_1, \Omega, \delta\rangle}$$

$$\text{Cond}\ \frac{\langle b, \Omega, \delta\rangle \rightarrow \langle b', \Omega, \delta\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ s_0\ \texttt{else}\ s_1, \Omega, \delta\rangle \rightarrow \langle \texttt{if}\ b'\ \texttt{then}\ s_0\ \texttt{else}\ s_1, \Omega, \delta\rangle}$$

$$\text{CondTrue}\ \frac{}{\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ s_0\ \texttt{else}\ s_1, \Omega, \delta\rangle \rightarrow \langle s_0, \Omega, \delta\rangle}$$

$$\text{CondFalse}\ \frac{}{\langle \texttt{if}\ \mathit{false}\ \texttt{then}\ s_0\ \texttt{else}\ s_1, \Omega, \delta\rangle \rightarrow \langle s_1, \Omega, \delta\rangle}$$

$$\text{While}\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ s, \Omega, \delta\rangle \rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (s;\ \texttt{while}\ b\ \texttt{do}\ s)\ \texttt{else}\ \{\}, \Omega, \delta\rangle}$$

$$\text{Assign1}\ \frac{}{\langle x := n', \Omega, \delta\rangle \rightarrow \langle \{\}, \Omega[x \mapsto n'], \delta\rangle}$$

$$\text{Assign2}\ \frac{\langle a, \Omega, \delta\rangle \rightarrow \langle a', \Omega, \delta\rangle}{\langle x := a, \Omega, \delta\rangle \rightarrow \langle x := a', \Omega, \delta\rangle}$$

$$\text{Send1}\ \frac{U = \{e \mid e = \mathsf{send}\,(o, p, n) \wedge e \in \delta\}}{\langle \texttt{send}\,(n', p)\,, \Omega, \delta\rangle \rightarrow \langle \{\}, \Omega, (\delta - U) \cup \mathsf{send}\,(o_2, m.p, n')\rangle}$$

$$\text{Send2}\ \frac{\langle a, \Omega, \delta\rangle \rightarrow \langle a', \Omega, \delta\rangle}{\langle \texttt{send}\,(a, p)\,, \Omega, \delta\rangle \rightarrow \langle \texttt{send}\,(a', p)\,, \Omega, \delta\rangle}$$

Figure 4.9: Task Execution Semantics

and

$$\left((E_{in}, t) :: \mathsf{tl}\left(s_{in}\right), (E', t) :: \mathsf{tl}\left(s^2_{out}\right)\right) \in R_c$$

then $(E', t) :: \mathsf{tl}\left(s^1_{out}\right) = (E', t) :: \mathsf{tl}\left(s^2_{out}\right)$ □

Each module is also a stream processing process (Definition 4.4.9). Each module converts receive events on its input ports into send events on its output ports. Note that at any time instant, modules execute tasks in a configuration where all the input port updates for that time instant has been applied. Like dataflows, the outputs resulting from a task computation are timeshifted by the module's delay.

Definition 4.4.9 (Semantics of Modules)**.** Let the input timed-event stream s_{in} of a module m be the merge of the input timed-event streams of its ports. The semantics of a module m with a task-delay of d is defined by the greatest ternary relation R_m over the module's input timed-event streams, timed-state streams, and output timed-event streams satisfying:

- Let $(E_{in}, t) = \mathsf{hd}\left(s_{in}\right)$, $(\Omega, t) = \mathsf{hd}\left(s_{\Omega}\right)$ and $(E_{out}, t + d) = \mathsf{hd}\left(s_{out}\right)$.
- If $(s_{in}, s_{\Omega}, s_{out}) \in R_m$ then either:
  1. If $\mathsf{sched}\left(m\right) \in E_{in}$ then $(E_{out}, \Omega') = \mathsf{eval}\left(T_m, \mathsf{update}\left(E_{in}, \Omega\right)\right)$ and $\mathsf{hd}\left(\mathsf{tl}\left(s_{\Omega}\right)\right) = (\Omega', t + 1)$ and $\left(\mathsf{tl}\left(s_{in}\right), \mathsf{tl}\left(s_{\Omega}, \mathsf{tl}\left(s_{out}\right)\right)\right) \in R_m$.
  2. If m's task is triggered by port p and $\mathsf{recv}\left(m, p, v\right) \in E_{in}$ then $(E_{out}, \Omega') = \mathsf{eval}\left(T_m, \mathsf{update}\left(E_{in}, \Omega\right)\right)$ and $\mathsf{hd}\left(\mathsf{tl}\left(s_{\Omega}\right)\right) = (\Omega', t + 1)$ and $\left(\mathsf{tl}\left(s_{in}\right), \mathsf{tl}\left(s_{\Omega}, \mathsf{tl}\left(s_{out}\right)\right)\right) \in R_m$.

3. If $\mathsf{sched}\,(m) \notin E_{in}$ and m's task is not triggered by any $\mathsf{recv}\,(m, p, v) \in E_{in}$ then $E_{out} = \emptyset$ and $\Omega' = \mathsf{update}\,(E_{in}, \Omega)$ and $\mathsf{hd}\,(\mathsf{tl}\,(s_\Omega)) = (\Omega', t+1)$ and $(\mathsf{tl}\,(s_{in})\,, \mathsf{tl}\,(s_\Omega, \mathsf{tl}\,(s_{out}))) \in R_m$.

For a given start state, modules are input-deterministic:

Lemma 4.4.2 (Modules are Input/State Deterministic)**.** For a module m let R_m be defined as in Definition 4.4.9. If $(s_{in}, s^1_\Omega, s^1_{out}) \in R_m$ and $(s_{in}, s^2_\Omega, s^2_{out}) \in R_m$ and $\mathsf{hd}\,(s^1_\Omega) = \mathsf{hd}\,(s^2_\Omega)$ then $s^1_{out} = s^2_{out}$ and $\mathsf{tl}\,(s^1_\Omega) = \mathsf{tl}\,(s^2_\Omega)$.

Proof. We proceed by *co-induction* on the streams. Let $(E_{in}, t) = \mathsf{hd}\,(s_{in})$ and assume that $\mathsf{hd}\,(s^1_\Omega) = \mathsf{hd}\,(s^2_\Omega) = (\Omega, t)$. There are three cases to consider:

Case 1 ($\mathsf{sched}\,(m) \in E_{in}$)**.** Since $\mathsf{hd}\,(s^1_\Omega) = \mathsf{hd}\,(s^2_\Omega)$, $\mathsf{hd}\,(s^1_{out}) = \mathsf{hd}\,(s^2_{out}) = (E_{out}, t+d)$ with $(E_{out}, \Omega') = \mathsf{eval}\,(T_m, \mathsf{update}\,(E_{in}, \Omega))$ (because task evaluation is deterministic relative to state and $\mathsf{update}\,(E_{in}, \Omega)$ is unique). We also see that $\mathsf{hd}\,(\mathsf{tl}\,(s^1_\Omega)) = \mathsf{hd}\,(\mathsf{tl}\,(s^2_\Omega)) = (\Omega', t+1)$ because $\mathsf{update}\,(E_{in}, \Omega)$ is unique. Application of the *co-inductive hypothesis* lets us conclude that for

$$\left((E_{in}, t) :: \mathsf{tl}\,(s_{in})\,, (\Omega, t) :: (\Omega', t+1) :: \mathsf{tl}\left(\mathsf{tl}\left(s^1_\Omega\right)\right), (E_{out}, t+d) :: s^1_{out}\right) \in R_m$$

and

$$\left((E_{in}, t) :: \mathsf{tl}\,(s_{in})\,, (\Omega, t) :: (\Omega', t+1) :: \mathsf{tl}\left(\mathsf{tl}\left(s^2_\Omega\right)\right), (E_{out}, t+d) :: s^2_{out}\right) \in R_m$$

it is the case that $(E_{out}, t+d) :: s^1_{out} = (E_{out}, t+d) :: s^2_{out}$ and $(\Omega', t+1) :: \mathsf{tl}\,(\mathsf{tl}\,(s^1_\Omega)) = (\Omega', t+1) :: \mathsf{tl}\,(\mathsf{tl}\,(s^2_\Omega))$.

Case 2 (m's task is triggered by port p and $\mathsf{recv}\,(m, p, v) \in E_{in}$)**.** Same as Case 1.

Case 3 (sched $(m) \notin E_{in}$ and m's task is not triggered by any recv $(m, p, v) \in E_{in}$)**.** The proof proceeds as in the first two cases except with hd $(s^1_{out}) =$ hd $(s^2_{out}) = (\emptyset, t + d)$.

□

The semantics of ODSDL programs are given by relating the inputs/outputs of devices, modules and dataflows according to how they are interconnected in the interactions block:

Definition 4.4.10 (Semantics of Programs)**.** Let x be an ODSDL program with the set of device specifications S, set of module instances M and set of dataflow connectors C. Let t be the moment in time p is started. Then the semantics of p are given by the greatest binary relation R_x over timed-event streams such that $(s_{in}, s_{out}) \in R_p$ when:

- Each module $m \in M$'s input stream s^m_{in} is the merge of all the output streams of the connectors c such that $c = i.j$ **flowsto** $m.p$ and with m's task scheduling stream s^m_{sched}. If m's task is activated periodically with period p, then $\forall_{n \in \mathbb{N}}$ sched $(m) \in s^m_{sched}(tn)$. Otherwise $\forall_{n \geq t}\ \emptyset = s^m_{sched}(n)$.

- The head of each module $m \in M$'s state-stream is (Ω, t) where Ω maps each variable to its initial value.

- Each connector $c \in C$'s input stream is equal to the output stream of module (or device) m if $c = m.j$ **flowsto** $l.k$.

- s_{in} is the merge of all the input-streams of connectors $c \in C$ such that $s.j$ **flowsto** $m.k$ such that $s \in S$ and $m \in M$.

- s_{out} is the merge of all the output-streams of connectors $c \in C$ such that $m.j$ **flowsto** $s.k$ such that $s \in S$ and $m \in M$.

Because dataflows and modules are input-deterministic, so are programs:

Theorem 4.4.3 (Programs are Input Deterministic)**.** For a program p let R_p be defined as in Definition 4.4.10. If $(s_{in}, s^1_{out}) \in R_p$ and $(s_{in}, s^2_{out}) \in R_p$ then $s^1_{out} = s^2_{out}$.

Proof. Because the each of the modules $m \in M$ have their configurations initialized to only one possible value all modules will map each input-stream to a unique output-stream (Lemma 4.4.2), and connectors map each input-stream to a unique output-stream (4.4.3) it follows from the Definition of R_p that $s^1_{out} = s^2_{out}$. □

4.4.4 Devices

The semantics of device specificiations are given in terms of Time Parametric Modal Specifications (Chapter 3). However, we do not need to go into details to describe how the devices relate to the timed-input and output event streams used here. Each device specification is fully captured by a *digital* TPMS of its behavior. Each digital TPMS models a *set* of input-output timed automata (IOTA) with digital semantics. We require each IOTA to be deadlock and timelock free, thus each IOTA induces a set of infinite timed input and output words.

Definition 4.4.11 (Timed Word)**.** A timed word is a (possibly infinite) sequence of alternating actions and delays $d \geq 0$ where $d \in \mathbb{R}$. We say that a timed-word is digital if it is always the case that $d \in \mathbb{N}$. For example:

$$\xrightarrow{0} a \xrightarrow{5} b \xrightarrow{2} \ldots$$

Is a digital timed-word.

If a word w contains only delays and actions from a IOTA's input alphabet we say w is a timed-input word. Likewise if w contains only actions from the output alphabet it is a timed-output word.

Each IOTA from an ODSDL device specification is input-enabled by construction so we can convert the timed-event output stream of an ODSDL program into an infinite timed word over the specification's inputs. Likewise we can convert the timed-words over the outputs of a IOTA into an input timed-event stream for the ODSDL program.

Definition 4.4.12 (Stream Semantics of Devices)**.** Let $\mathcal{X}$ be the TPMS corresponding to device specification D and let $[\![\mathcal{X}]\!]$ be the set of IOTAs implied by $\mathcal{X}$. Let W be the union of all the timed words of all the IOTA in $[\![\mathcal{X}]\!]$. The the stream semantics of device specification D is given by the greatest relation R_D that satisfies the following: $(s_{in}, s_{out}) \in R_D$ if s_{in} is the timed-event stream corresponding to the input-projection of some infinite word $w_i \in W$ and s_{out} is the timed-event stream corresponding to the output projection of w_i.

4.5 Related Work

Programs executing as part of a larger cyber-physical system typically have real-time requirements: in order to be correct they must perform the correct computations at the correct time. This presents major challenges for program portability and program verification. Real-time programs are typically not portable because their timing is dependent on the processor and how the underlying operating system allocates (*i.e.,* schedules) processor time to different programs. If a real-time

program is run on a different processor or operating system it will have different timing characteristics which in turn could cause the program to violate its timing requirements. Even if a program meets its timing requirements (*e.g.,* always completes before its deadline) execution non-determinism (*e.g.,* caused by operating scheduling decisions or speculative effects in the processor) can make it difficult for developers to reproduce bugs or to relate testing results back to assurance claims. Difficulties with portability and verifiability of real-time code drive up the development costs for real-time software systems. Furthermore, the platform approach to on-demand systems requires portable real-time programs.

In the 1990's DARPA invested in the MetaH [177, 176] program to address problems surrounding real-time program portability and verifiability. MetaH resulted in a set of tools and programming constructs for the development of distributed real-time software systems. MetaH allowed developers to construct software systems out of components that communicate through ports. Each component may have a number of periodic or sporadic real-time tasks that do the actual processing. Tasks in MetaH can read or write to their container component's ports. MetaH addresses time-determinism by assuming that tasks only share data via ports, and that the results of a computation are only "visible" at a task's deadline (regardless when the computation actually finished). This ensures that computations are time-deterministic if all the tasks of the software system are schedulable on the given platform. MetaH inspired the Architecture Analysis & Design Langauge (AADL) SAE standard [65] and many of MetaH's features have found their way into AADL.

In 2001 Henzinger *et al.* rediscovered many of MetaH's determinism features and included them in the Giotto programming language [80, 79]. In fact, it is

fairly accurate to think of Giotto as equivalent to the periodic subset of MetaH. There are, however, two major differences betweem Giotto and the periodic subset of MetaH. First, Henzinger *et al.* provide formal operational semantics for Giotto programs. Second, Giotto is not coupled to an underlying scheduler (MetaH assumed a form of rate-monotonic scheduling).

[58] describe the PTIDES deterministic programming model for distributed real-time embedded software. PTIDES semantics are based on discrete event simulation: Programs in PTIDES consist of actor components which can communicate with each other via ports and channels that link those ports. Whenever a PTIDES actor sends a message it is timestamped. The timestamp is used by the underlying PTIDES exection platform to ensure that each actor only processes input messages in timestamp order [59, 189, 188]. Because receipt of a message triggers actor computation, PTIDES supports aperiodic execution. Periodic execution can be achieved with a specialized clock actor that generates messages at period boundaries. Because PTIDES programs consist of communicating actors (as opposed to a collection of tasks) new schedulability techniques needed to be devised to asses the schedulability of PTIDES programs. Christos Stergio established the decidability of PTIDES scheduling in his PhD dissertation [169]. He showed that in general schedulability is no harder than the reachability problem for timed automata, and if only a subset of PTIDES is considered it is equivalent to earliest deadline first schedulability analysis.

For on-demand systems we want a programming model that is deterministic, has precise formal semantics, is easy (*i.e.,* efficient) for paltforms to schedule, provides familiar programming constructs and allows for both periodic and aperiodic computation. The programming model of MetaH (and AADL) don't yet

have a complete and formal semantics. Giotto only allows periodic computation. While PTIDES has discrete event semantics these are never precisely given. For example it is unclear what should happen if an actor receives two events with the same timestamp. Furthermore, PTIDES does not precisely couple *model-time* with physical time: Actuators must receive their messages prior to the expiration of the timestamp, but when the actuator acts on the message is undefined (presumably it happens after, but not too long after, the timestamp). The actor construct of PTIDES is also different from what most programmers are used to (*i.e.,* tasks). Furthermore, the schedulability analysis of PTIDES programs can be quite complicated (both operationally and in terms of time-complexity) which is not suitable for a platform that must perform schedulability analysis on-demand.

Chapter 5

A Prototype Medical Application Platform

In this Chapter we describe the Medical Device Coordination Framework (MDCF) and the MIDdleware Assurance Substrate (MIDAS). The combination of the MDCF and MIDAS results in a prototype MAP implementation designed to demonstrate the feasibility of key technologies essential to the viability of the MAP concept. As discussed in Chapter 2, a MAP could host any number of safety- or privacy critical applications. Ideally, the MAP certification criteria in a real on-demand ecosystem would require that all MAPs satsify the following "*high-assurance*" features:

1. Enforce a high degree of (potentially complete) of isolation and separation between independent applications and system components.

2. Employee sound techniques to secure sensitive (i.e., private data) and determine the authenticity of devices and applications.

3. Utilize fault-tolerant (or fault resistant) hardware and algorithms.

4. Be verified & validated (possiiblly with formal, machine-checked, proofs) to the highest level of assurance[1].

While all the above are important for a real MAP, this chapter does not focus on how each and every possible *high assurance* feature could be implemented and incorporated into the MDCF/MIDAS. Indeed, many useful high assurance features (such as those listed above) have been studied extensively in the literature and in some cases are available in commercial products (see the related work Section 5.6.1). Instead, this chapter focuses on describing the platform capabilities that are both new and necessary to make *our* vision of a MAP ecosystem work.

A key challenge for any MAP is to provide determinism in a dynamic, open, and distributed environment. Most distributed systems with strict determinism requirements are designed assuming the system itself will be static (*i.e.,* the tasks and data-flows that comprise the "digital" portion of the system are fixed from the factory). Consequently, most technology developed to enable determinism have not been designed to work for systems that are dynamic, open, and distributed. For example, the networking technology typically used in high assurance real-time systems (*e.g.,* the Time Triggered Architecture [113], Time Triggered Ethernet, FlexRay [53], *etc.*) can provide the needed timing guarantees and inter-component isolation but must be configured "offline". Likewise, while there has been a number of promising distributed deterministic programming models developed (*e.g.,* PTIDES) their execution strategies require complex procedures (both operationally and computationally) for *schedulability analysis* which makes on-line admission control difficult.

[1] *e.g.,* the criteria laid out for DO-178C Level A or EAL7

This chapter presents three contributions towards enabling strict determinism in an open, dynamic, and distributed system. The first is the MIDAS. MIDAS combines Software Defined Networking (SDN) [136] with a publish/subscribe communications abstraction. MIDAS enables programmers to specify timing constraints on logical flows of data between producers and consumers. MIDAS then enforces the timing constraints and specified isolation by reconfiguring the underlying network packet switching infrastructure on the fly. The second contribution is a *determinizing scheduler* that implements the DLT semantics of ODSDL programs. The resource demands of the execution strategy employeed by the determinizing scheduler can be reduced to a traditional task-set specifciation which enables the use of relatively simple schedulability analysis techniques. The third contribution is a collection of minor modifications to existing real-time scheduling techniques. These minor modifications are necessary for us to use classic real-time scheduling theory with real COTS operating systems and networking hardware.

This Chapter is organized as follows. Section 5.1 gives a functional overview of the software architecture of the MDCF/MIDAS. Section 5.2 gives a detailed overview of the MIDAS Channel-Service. The MIDAS Channel-Service provides the hard real-time publish/subscribe communications primitives leveraged by the *determinizing scheduler* to implement the DLT semantics in a distributed system. Section 5.3 describes the overall operation of that determinizing scheduler and how it achieves the LET semantics. Section 5.4 describes a variety of practical scheduling techniques used by the MDCF/MIDAS to ensure that tasks and messages always meet their deadlines. Section 5.5 an evaluation and performance assesment of the MDCF/MIDAS' ability to guarantee deterministic application

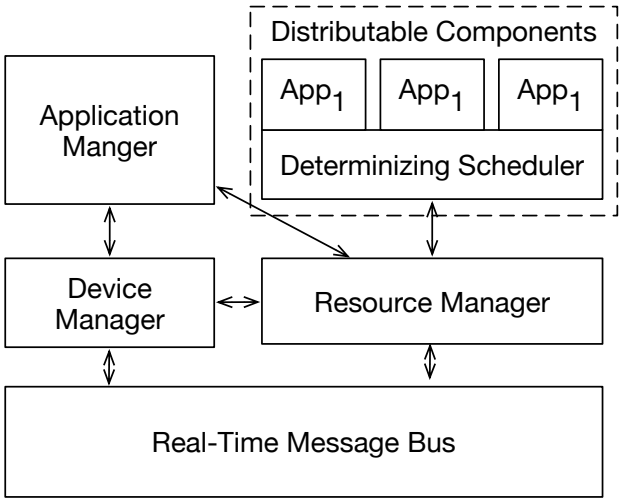


Figure 5.1: MDCF Software Architecture.

execution.

5.1 MDCF Architecture and Functional Overview

The MDCF/MIDAS is a set of integrated services intended to run on top of a traditional real-time operating system. These services work with each other, the underlying hardware and operating system to manage the lifecycle of MAP applications and provide a predictable environement for those applications to execute in. Figure 5.1 shows the software architecture of the MDCF/MIDAS. The MDCF/MIDAS is comprised of the MIDAS Real-Time Message Bus, a Device Manager, an Application Manager, and a Resource Manager that supports the Determinizing Scheduler.

5.1.1 Real-Time Message Bus

The Real-Time Message Bus (RTMB) provides a publish/subscribe messaging service and is used a number of ways by the MDCF/MIDAS components to communicate with other. Devices use the RTMB to communicate with the MDCF via combination of public (*i.e., atrium*) and private topics (*e.g.,* announce the device's presence, engage in authentication, *etc.*). The Application Manager & Resource Manager use the RTMB as the underlying transport medium for ODSDL dataflows. The RTMB leverages software defined networking (SDN), and in particular the OpenFlow protocol, to guarantee the timeliness of any time-critical messages. The design and operation of the RTMB will be elaborated in Section 5.2.

5.1.2 Device Manager

The Device Manager manages the *association lifecycle* of devices used with MDCF applications. Each stage of the association life-cycle of a device defines to what extent it is available to be used by an application. Figure 5.2 is an illustration of the MDCF device connection protocol and shows the relationship between each stage (state) of the lifecycle. Transitions between states are either annotated with CSP [85, 84] style communications across channels implemented with the RTMB or locally detected events.

When a device is first plugged into the network it is in the `Disconnected` state. Once the device detects it has been plugged in, it annouces itself to the MDCF Device Manager and begins the process of authentication (state `Authenticating`). Once the device has been successfully authenticated (*i.e.,*

the MDCF has validated the device's certification credentials) the device transmits its behavior specification and becomes "associated", or available for with an application (`Associated`). If the MDCF or RTMB detects a disruption in the underlying network transport the device is considered `Lost` until the transport medium has recovered.

After the device has transmitted its specification, but before the device can be used with an application, the Device Manager will direct the RTMB to create topics to carry any medical data streams the device may output. In general, a single unique topic is created for each output port of the device's specification (see Figure 5.3). If the RTMB or Resource Manager cannot allocate the resources to create the necessary topics the association process will halt.

5.1.3 Application Manager

The Application Manager controls the lifecycle of MDCF applications (see Figure 5.4). When a user wants to start an MDCF application they use the MDCF console to select the desired application and the set of devices they wish to couple with the application. The Application Manager first checks whether the application and selected devices are compatible. If so, the MDCF works with the Resource Manager and RTMB to perform admission control, *i.e.,* it checks if it can ensure that the application's QoS requirements are always met given the application's resource needs and the current demand on the platform's resources. If the application QoS requirements can be met the platform reserves the necessary resources, instantiates the application and then binds the application to the selected devices (Figure 5.4). The binding process involves subscribing the application to the topics corresponding to the devices publish ports and the devices to topics carrying

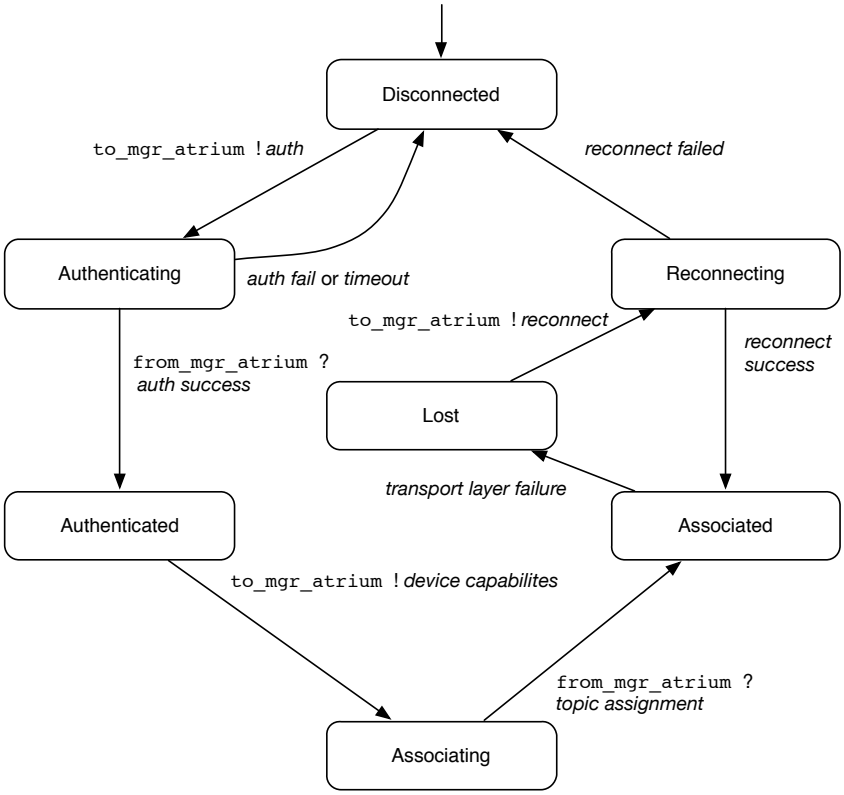


Figure 5.2: Device connection protocol

application to device messages. Figure 5.5 shows how the dataflow declaration from the pulse-oximeter alarm is realized when an instance of the application is bound to a `PulseOximeter` with UID 1.

5.1.4 Resource Manager

The Resource Manager ensures that every running application's timing requirements will always be met and that they maintain their DLT semantics. The resource manager is comprised of two sub-systems:

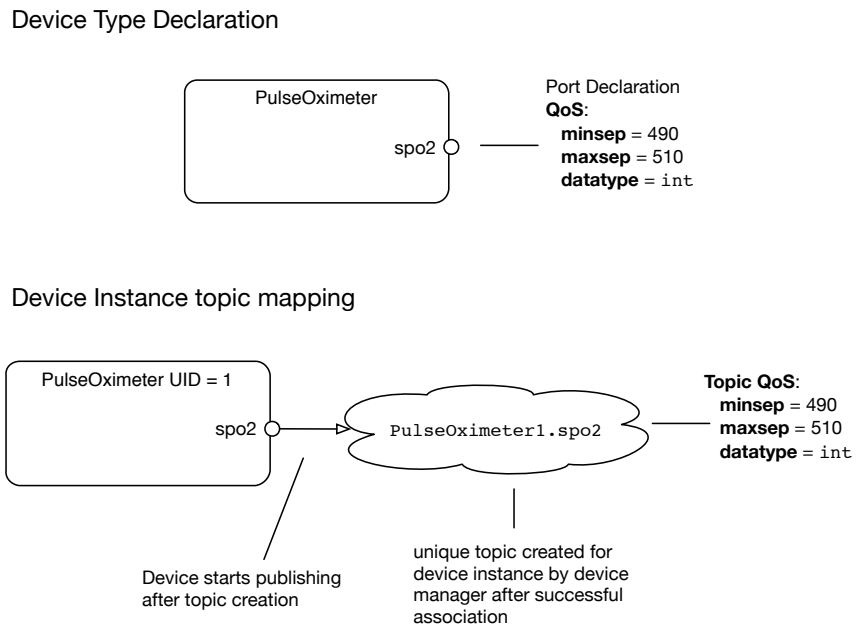


Figure 5.3: Relationship between a device's logical data ports tand RTMB topics.

1. A low-level scheduler that makes sure all tasks and messages complete *prior* to their deadlines.

2. A high-level scheduler, which we call the *determinizing scheduler*, that manipulates when messages and the output of tasks become visible to implement the DLT semantics of ODSDL programs.

When a user requests the MDCF to start an application the Resource Manager does admission control. First, it transforms the application's timing specification into a resource specification (*e.g.,* task set). Second, it generates a *resource configuraton* (*e.g.,* prioritization) for the resource specification. Third, it checks the resource configuration for feasibility (*i.e.,* whether all timing requirements will

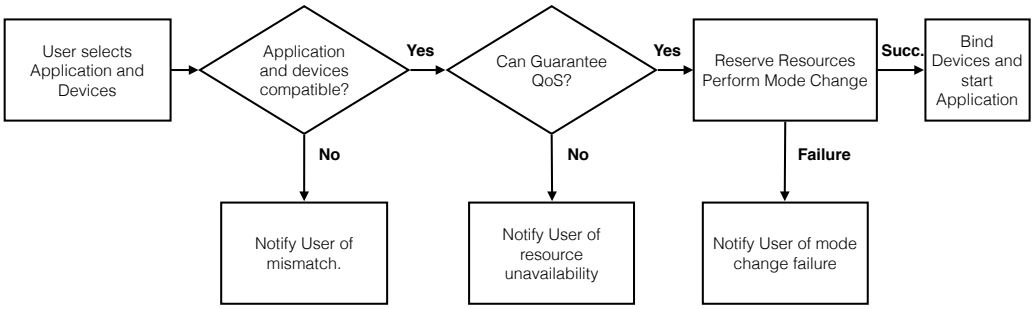


Figure 5.4: Platform system instantiation process

be satisfied). If the configuration deemed feasible the resource configuration is applied and the application is started.

After the application has started, the determinizing scheduler takes control of application task execution and dispatch. The operation of the determinizing scheduler is elaborated in Section 5.3 and the algorithms and scheduling techniques used by the MDCF prototype are elaborated in Section 5.4.

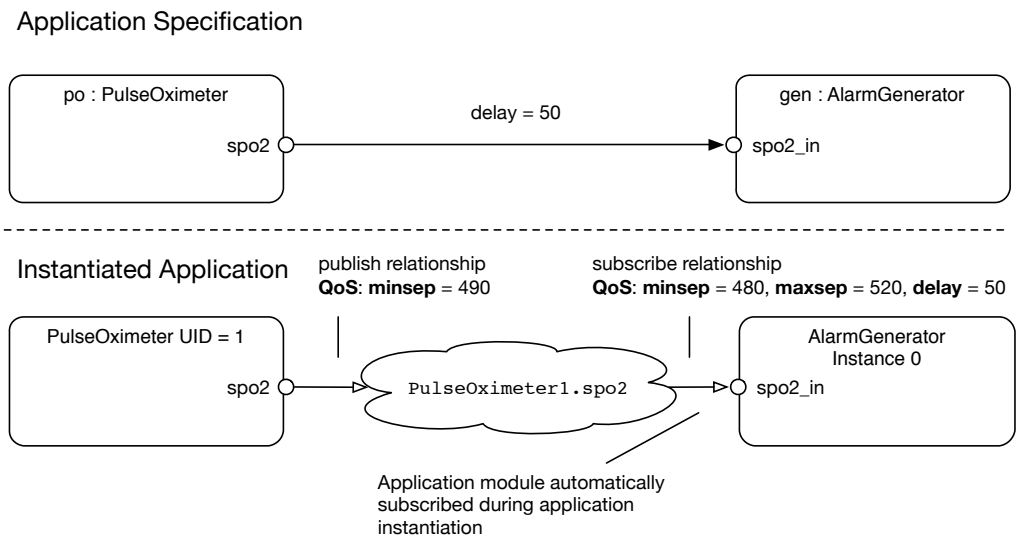


Figure 5.5: Example mapping of MDCF application dataflow declarations to publish/subscribe relationships

5.2 The Real-Time Message Bus

The MIDAS Real-Time Message Bus (RTMB) is a publish/subscribe middleware designed to enforce end-to-end real-time guarantees. The RTMB is able to do this even if users change the configuration of the network during runtime (*i.e.,* add or remove nodes) or the network contains malicious nodes (*i.e.,* nodes that attempt disruption of legitimate applications with denial of service attacks). The RTMB is able to do this because, unlike traditional real-time middleware, it uses recent software defined networking technology to enforce resource allocation directly in the network switches. Because the RTMB enforces timing behavior of network communication directly in the network infrastructure (as opposed to asking the end systems to politely share resources) its real-time guarantees are *robust*. If a device on the network becomes faulty and starts spamming the network with errant

packets, other devices or applications in the network will not notice. These features make the RTMB appealing as the communications substrate for on-demand systems and is why the MDCF uses the RTMB to implement the real-time communications channels from ODSDL applications.

5.2.1 Publish Subscribe and Quality of Service

The publish/subscribe abstraction consists of *publishers* (producers of data), *subscribers* (consumers of data) and *topics* (names used to organize the communications space). Publishers send updates to topics. As their name suggests, subscribers choose to subscriber to certain topics. When a topic receives an update from a publisher, that update is disseminated to all of that topic's subscribers. Figure 5.6 shows a system with one publisher, one topic and three subscribers.

Recall from Chapter 4 that the real-time channels in ODSDL have an associated latency parameter and that the ports on module and device components have minimum and max separation. Not coincidentally the RTMB publishers and subscribers specify minimum and maximum update separation. Subscribers can specify maximum end-to-end *network* latency. This latency describes the amount of time it takes the network to transmit the whole update from the publisher to the subscriber (see Definitions 5.2.1 and 5.2.2).

Definition 5.2.1 (Network Latency)**.** Let $\mathcal{P}_\mathcal{T}$ be a publisher publishing to topic $\mathcal{T}$ and $S_\mathcal{T}$ be a subscriber to $\mathcal{T}$. Let t_1 be the moment $\mathcal{P}_\mathcal{T}$ starts transmitting message m on the network and let t_2 be the smallest time t after m has been fully received by $\mathcal{S}_\mathcal{T}$. The end-to-end network latency of m is $\mathcal{L}\left(\mathcal{S}_\mathcal{T}, m\right) = t_2 - t_1$.

Definition 5.2.2 (Guaranteed Maximum End-to-End Latency)**.** Let $\mathcal{P}_\mathcal{T}$ be a pub-

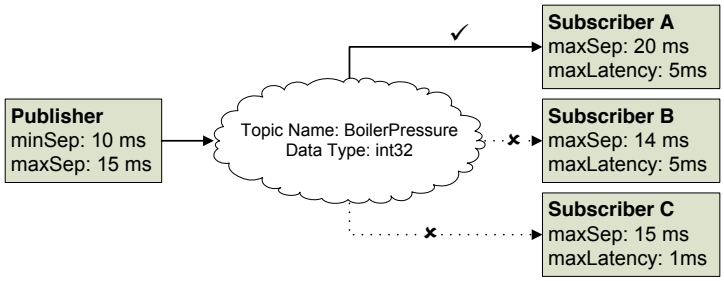


Figure 5.6: Real-Time Publisher and Subscribers with QoS.

lisher publishing to topic T and $\mathcal{S}_{\mathcal{T}}$ be a subscriber to $\mathcal{T}$. If $\mathcal{S}_{\mathcal{T}}$ requests a guaranteed maximum latency, denoted $\mathcal{L}_{max}\left(\mathcal{S}_{\mathcal{T}}\right)$, then $\forall_m \mathcal{L}\left(\mathcal{S}_{\mathcal{T}}, m\right) \leq \mathcal{L}_{max}\left(\mathcal{S}_{\mathcal{T}}\right)$.

Note that the latency semantics of the RTMB are not quite the same as in the ODSDL: latencies in the RTMB only concern the time it takes to move the message across the network while in the ODSDL channel latency is the delay between the moment one *software* agent sends the message and the receiver receives it. RTMB latency (*i.e.,* network latency) is a component of ODSDL channel latency which also includes the time it takes the message to move through the host's network stack and get deserialized.

Figure 5.6 illustrates how MIDAS will match QoS between publishers, subscribers, and the underlying system. Subscriber A is admitted to the system because its required $maxSep$ (20ms) is greater than or equal to the publisher's (15ms).

In this example the RTMB has determined it can guarantee the requested maximum latency. Subscriber B is not admitted because it requires a $maxSep$ of 14ms which is smaller than the publisher's. Finally, Subscriber C is not admitted to the system only because the underlying middleware has determined that it cannot

guarantee C's requested maximum end-to-end latency.

5.2.2 RTMB Overview

The RTMB achieves real-time guarantees on open networks built from COTS equipment by handing complete control over the network to the middleware via the OpenFlow [137] software defined networking protocol. Tight integration of the middleware with OpenFlow provides several benefits. First, it gives the middleware complete control over how data packets on the network are forwarded, prioritized, and rate-limited. Second, many COTS switches can be made OpenFlow capable with a firmware update. This means that existing network deployments can be made OpenFlow capable (and therefore could be made suitable to carry traffic for on-demand systems). Third, in many OpenFlow switches all OpenFlow rule processing occurs at line rate. This means that the middleware can affect configuration changes in the network without any appreciable loss of network peformance.

We now describe the operation of an OpenFlow network. An OpenFlow network consists of two types of entities: OpenFlow switches and OpenFlow controllers. An OpenFlow hardware switch is a Layer 2/3 Ethernet switch that maintains a table of *flow* entries and actions.

The flow table associates each flow with an *action* set which tells the switch how to handle a packet matching the flow. Table 5.1 shows an example flow table. The table has two flow entries which match against input port, Ethernet address, IP address and UDP port number. There are two actions associated with each flow. While the OpenFlow specification describes a number of different actions our prototype utilizes the enqueue and meter actions. The meter action requires

Input Port	Datalink				IP		UDP		Action
	VLAN ID	Src	Dst	Type	Src	Dst	Src Port	Dst Port	
3	0	89ab	89ac	IP	192.168.1.1	192.168.1.2	100	101	meter=1,enqueue=4:7
4	0	89ac	89ab	IP	192.168.1.2	192.168.1.1	101	100	meter=2,enqueue=3:2

Table 5.1: Example OpenFlow flow table

the switch to apply traffic policing to the flow. The enqueue action requires the switch to place the packet on an egress queue associated with a specific port during forwarding.

When an OpenFlow switch receives a packet on one of its interfaces, it compares that packet to its flow table. If the packet matches a flow table entry, it applies the action set associated with that flow entry. If the packet does not match an existing entry the switch performs what the OpenFlow protocol calls a *packet-in*. When the switch performs a packet-in, it forwards the packet to the OpenFlow controller (a piece software running on a server in the network). The controller analyzes the packet and can execute any arbitrary algorithm to generate a new flow rule. The controller can then update the switch's flow table with the new rule. Packet-in allows the OpenFlow controller to learn the topology of the network (*i.e.,* learn what ports on what switches different hosts are connected to) and then effect complex routing, forwarding and QoS strategies with algorithms implemented in a normal high level programming language like Java or C++.

We now provide an overview of how the RTMB provides real-time guarantees on OpenFlow enabled COTS networks. The RTMB implements a Global Resource Manager (GRM) which contains a specialized OpenFlow controller. When a publish-subscribe client comes online it first connects to the GRM. This allows the GRM to learn where on the network the client is located (*i.e.,* the switch

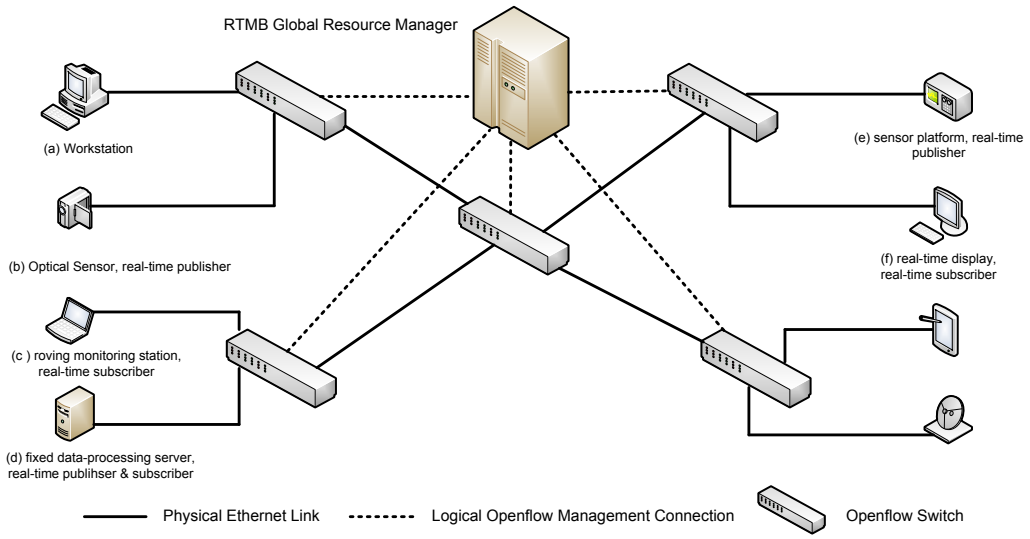


Figure 5.7: Deployment of the RTMB on an OpenFlow enabled network.

and port it is connected to). Then, when a client requests a subscription with a specified QoS, the GRM will peform *admission control*. First, the scheduling algorithm in the GRM will generate a new network configuration based on the new QoS request. The new configuration is then analyzed by a *schedulability test* which determines if any QoS constraints could be violated with that configuration (see Section 5.4.2 for an example scheduling and schedulability algorithm). If a violation is possible, the client is notified and their request is not granted. If QoS is guaranteed in the new configuration, the GRM commits the network configuration to the network using OpenFlow and then admits the client. Note that this system architecture allows us to handle non publish-subscribe best effort traffic (*e.g.,* web-browsing) on the same network transparently; the GRM will automatically map best effort traffic to the lowest priority queues on each switch. See Figure 5.7 for an example RTMB deployment.

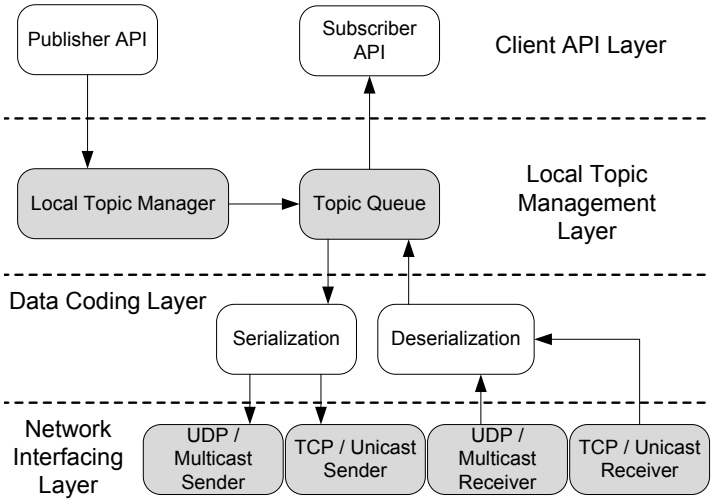


Figure 5.8: Client Library

5.2.3 Middleware Design

Now we describe the various software components in the RTMB. The RTMB adopts a brokerless architecture and the functionality of middleware is separated into two software stacks implemented in Java. The client library provides the publish-subscribe abstraction to clients that wish to be publishers or subscribers. The Global Resource Manager (GRM) runs on a server connected to the network and is responsible for managing active topics, publishers, subscribers and the underlying network configuration. Both the client library and GRM have features specifically designed to enable automatic QoS guarantees.

Client Library

The architecture of the client library is illustrated in Figure 5.8. If the application is a publisher, messages flow from the application to a local topic queue by way

of the local topic manager. This allows the client library to perform a *zero-copy* transfer of data between publishers and subscriber that are running on the same host. Each local topic queue always has a special subscriber: the data-coding layer. The data-coding is responsible for serializing messages prior to transmission on the network. After a message has been serialized, a *sender* object transmits it onto the network. The type of sender used depends on what transport protocol was negotiated with the GRM. Symetrically, the *receivers* receive messages from the network, pass those messages to the data coding layer where they are deserialized and then placed on the appropriate topic queue. Subscribers are invoked when the topic queue associated with their topic becomes non-empty.

The Client Library has one important feature used to support automatic QoS guarantees: it statically infers the maximum serialized message sizes from message types. When a publisher comes online it specifies the type of message it will publish. The API passes this information to the topic management layer, which in turn asks the data coding layer for message size bounds on that type. In our prototype, the data coding layer uses Java reflection to determine the structure of the type and infer the maximum number of bytes used to represent a message of that type on the network.[2] Maximum message size information is used by the GRM when it performs the schedulability analysis of a network configuration.

[2]Our prototype currently only supports non-inductive types (*e.g.,* record types) that can be easily analyzed for size-bounds.

Global Resource Manager

The GRM (Figure 5.9) is responsible for orchestrating all activity on the network to ensure that data is correctly propagated between publishers and subscibers. To accomplish this, the GRM must maintain configuration information about the network and implement the appropriate scheduling and network reconfiguration algorithms. Because we are concerned with providing guaranteed timing, the GRM must keep record of how switches in the network are interconnected, where clients are plugged into the network, the performance characteristics of each switch, and which multicast addresses are associated with what topics.

These various responsibilites are decomposed along module boundaries. Several of these modules' functions do not need to be extensively elaborated: the client manager is a server process that handles client's requests (*e.g.,* to start publishing on a topic); and the topic manager maintains a record of active topics and the network addresses associated with each topic, the OpenFlow controller implements the OpenFlow protocol and exposes a simple API to the flow scheduler to reconfigure the network.

The flow scheduler implements the admission control, scheduling and network reconfiguration algorithms used to ensure QoS constraints are not violated (see Section 5.4.2).

We now elaborate the network graph and the switch model data in more detail. The switch model database is a repository of performance and timing characteristics for different models of OpenFlow switch. This information is vitally important to the GRM; it needs to know how each switch in its network behaves. The information in the switch model repository is created before the middleware

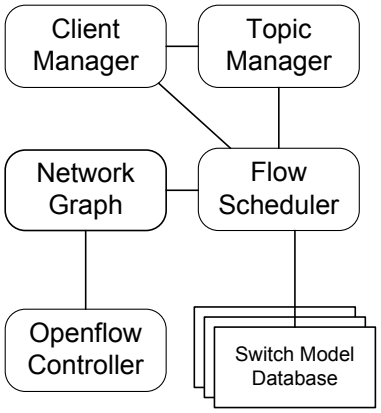


Figure 5.9: Global Resource Manager Architecture

is deployed on a network. In our protoype each switch model is represented by an XML file that is read by the GRM when the GRM starts up. Each switch model contains the model name of the switch, the number of ports on the switch, the number of egress queues associated with each port, the bandwidth capacity, and the number and precision of the hardware rate-limiters and the internal multiplexing latency of the switch's switching fabric.

The network graph maintains both static and dynamic network configuration information. The static information is specified at deployment time; it defines what switches are on the network (the model, etc.) and how they interconnect. The dynamic information is either learned via OpenFlow (*e.g.,* what ports on which switch are specific client connected) or set by the flow scheduler.

Figure 5.10 illustrates a simple network graph. The network consists of two switches. These switches are connected via an uplink cable on each of their port 1. Each switch is connected to two clients (denoted by dotted circles).

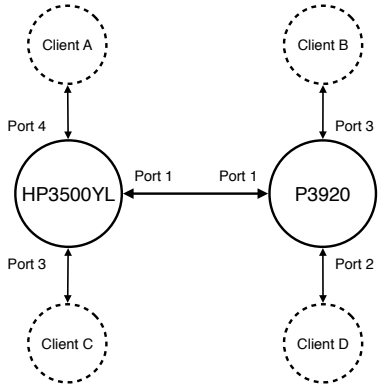


Figure 5.10: Example Network Graph

5.3 Determinizing Scheduler

ODSDL semantics dictates that tasks and message transfers always take a predictable Logical Execution Time. If a module or device sends a message at time t along a dataflow with delay D then the message should arrive at its destination at precisely t + D. If a modules task, with a specified relative deadline D, starts at time t, then its computations proceed as if the state of the input ports are frozen. Then, the results of the computation manifiest at precisely t + D. In reality, the duration it takes to transmit messages between devices and modules can vary from moment to moment due to network loads and how the networking infrastructure schedules network packets. Likewise, the actual execution time of a task can vary significantly due to different program execution paths, the state of the underlying processor (i.e., branch predictors, pipeline scheduling, cache and so on.) The Job of the Determinizing Scheduler is to orchestrate the computation and communications of application processes so they behave according to the ODSDL LET semantics.

At a high level, the Determinizing Scheduler works as follows. All ODSDL application tasks and communications in the MDCF are separated into two types of phases: A work phase where the bulk of the communications and computations are performed and an I/O phase where the I/O of the application processes are made visible to other processes or the environment. A Determinizing Scheduler service is present on all the system components (*i.e.,* the platform and devices) and maintains a two-level scheduler. The top-level scheduler runs at the highest system priority and will execute the lightweight microtasks corresponding to the I/O phases of application processing. The bottom-level scheduler schedules tasks associated with the work phase of application processes. The bottom-level scheduler can implement any scheduling discpline, as long as all tasks run at a priority lower than the Determinizing Scheduler and all work-phase tasks finish prior to their deadlines. As the work-phase tasks execute the MDCF runtime works with the Determinizing Scheduler to trap all I/O operations (*e.g.,* calls to the ODSDL `send` function) and schedule a corresponding microtask to manifest the I/O event at the appropriate time in the future.

How can the Determinizing Scheduler make the I/O events happen at the correct time? There are two obvious challenges. The first challenge is the time quanta from ODSDL semantics: It is only 1 ms in duration. If microtask execution takes to long or has to much jitter then event wont happen at the prescribed time. Fortunately, as we will see, the microtasks can be engineered so they are both very fast and predictable. They are fast because they typically only involve swapping references. They are predictable because they dont involve much branching or data. The second challenge has to do with time in a distributed environment. The "correct time for an event as defined by the ODSDL semantics is a moment on a

global shared timeline. In reality, each component will have access to a different clock which will not be perfectly synchronized. However, as long as all clocks are synchronized within some ϵ s.t., $\epsilon \leq 1/2\ ms$ the Determinizing Scheduler can manifest the I/O events when all clocks agree its the correct time.

There are two parts of this section. Section 5.3.1 gives an overview of the operation of the Determinizing Scheduler as well as psuedocode describing its functions. Section 5.3.2 makes a precise correctness claim and gives an informal proof for the correctness of the Determinizing Scheduler.

5.3.1 Operation

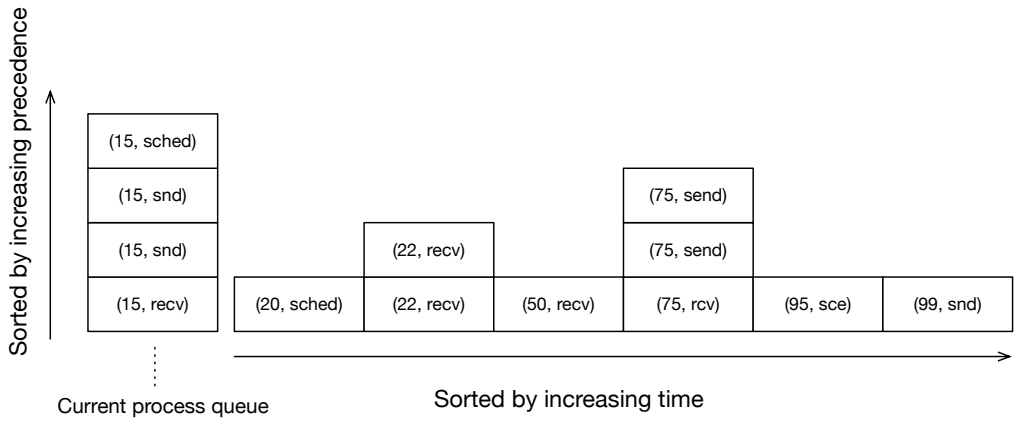


Figure 5.11: Example determinizer event queue.

The first-level of the Determinizing Scheduler runs at the highest system priority. It maintains a queue of *microtask* sets (Figure 5.11). Each microtask represents one ODSDL event (*i.e.,* $\mathsf{send}, \mathsf{sched}, \mathsf{recv}$). Each microtask set is timestamped indicating *when* the events in that queue should happen. Assuming all clocks in the system are synchronized within ϵ, the first-level scheduler will wake

Algorithm 5 Update and (Re)Schedule Future Events

Precondition: (e, t) is an ODSDL event/timstamp pair where $t > t_{now}$.

Precondition: T is the event queue.

```
1: function SCHEDULEFUTURE((e, t))
2:     t_next ←GETNEXTACTIVATION()
3:     E ← M(t)                    ▷ creates an empty queue at t if there is no entry for t
4:     INSERTORDERED(E, e)         ▷ recv ordered before sched, snd
5:     if t < t_next then
6:         SETNEXTACTIVATION(t)
7:     end if
8: end function
```

up after the local clock c equals $t + \epsilon$ to start processing the microtasks scheduled for time t. It will finish task processing before c equals $t + (1 - \epsilon)$. After it has finished processing the events for time t it will sleep until it is time to process the next batch of events. Algorithm 5 gives the function used by the MDCF runtime to schedule microtasks. Algorithms 6 and 7 give procedures used by the Determinizing Scheduler to process the pending microtask queue and execute the microtasks. We will walk through each.

Algorithm 5 is used by the MDCF runtime to schedule microtasks. For example, when a application task that was dispatched at t and with deadline D calls the ODSDL `send` function, the MDCF runtime will schedule a send microtask for time $t + D$. When the MDCF requests some microtask to be scheduled at time t Algorithm 5 will find the microtask set with the matching timestamp (creating one if non exists). If the new event must happen earlier than any event currently in the queue the Determinizing Scheduler's next wakeup time must be reset. Observe that each microtask set is actually partially ordered queues, with recv events

Algorithm 6 Process current events

Precondition: T is the microtask queue.

Precondition: The local clock $c > t_{now} + \epsilon$

Postcondition: The local clock $c < t_{now} + (1 - \epsilon)$

```
1:  function PROCESSCURRENT
2:      E ←HEAD(T)
3:      t_now ←GETTIMESTAMP(E)
4:      INSERTORDERED(E, e)                    ▷ recv ordered before sched, snd
5:      for e ∈ E do
6:          COMMIT(e, t_now)
7:      end for
8:      REMOVE(E, T)
9:      t_next ← GETTIMESTAMP(HEAD(T))
10:     SETNEXTACTIVATION(t_next)
11:     SLEEPUNTILNEXTACTIVATION()
12: END FUNCTION
```

ordered before sched, snd events. This ordering ensures that events interleave in the order defined by the ODSDL semantics.

Algorithm 6 is the main processing loop for the Determinizing Scheduler. When it wakes up, it will locate the microtask set corresponding to the current time. If the current timestamp is t_{now}, it will wakeup after the local clock reads $t_{now} + \epsilon$. The MDCF resource manager ensures that the Determinizing Scheduler will have enough resources to finish by $t_{now} + (1 - \epsilon)$. After microtask processing is complete, the Determinizing Scheduler will set a timer to wake itself up for the next timestamp then put itself to sleep.

Algorithm 7 gives the processing logic for each type of microtask. sched(m) tasks cause the Determinizing Scheduler to dispatch the work-phase task of mod-

Algorithm 7 Commit Event Effect

Precondition: e is an event. t is the timestamp of the event. S is a set of tasks to dispatch now.

```
1:  function COMMIT(e, t)
2:      if e = sched(m) then
3:          (P, D) ←GETTASKSPEC(m)
4:          if P ≥ 1 then                                   ▷ Is the task periodic?
5:              SCHEDULEFUTURE((sched(m), t + P))
6:          end if
7:          DISPATCH(m, D, COPY(m))
8:      end if
9:      if e = send(m_s, p_s, v) then
10:         F ←GETFLOWS(m, p)                               ▷ The flows leaving m_s.p_s.
11:         for (m_d, p_d, D) ∈ F do
12:             if ISREMOTE(m_d) then
13:                 RTMBSEND(m_d, p_d, D, t + D)            ▷ Timestamp message and send via the
    RTMB.
14:             else
15:                 SCHEDULEFUTURE((recv(m_d, p_d, v), t + D))
16:             end if
17:         end for
18:     end if
19:     if e = recv(m, p, v) then
20:         m.p ← v
21:         if TRIGGERSTASK(m, p) then
22:             S ← S ∪ {sched(m)}   ▷ Need to dispatch task after all other effects have been
    committed.
23:         end if
24:     end if
25:     for sched(m) ∈ S do
26:         (P, D) ←GETTASKSPEC(m)
27:         DISPATCH(m, D)
28:     end for
29: end function
```

ule m to the bottom-level scheduler. If the task of m is periodic with period P the Determinizing Scheduler will schedule another $\mathsf{sched}(m)$ microtask for P milliseconds in the future. When the Determinizing Scheduler dispatches m's task it will give the task a *copy* of m's input port state to work in. Running the task with a copy ensures it executes with a snapshot of m's state at time t. The $\mathsf{send}(m_s, p_s, v)$ microtask involves the most processing: The Determinizing Scheduler must retrieve all the data flows associated with port $m_s.p_s$ and then schedule a corresponding receive task D milliseconds in the future where D is the specified delay of the flow. The relative complexity results from having to iterate over all the flows associated with the port. Additionally, the logical destination port of the flow may not be on the local processor. If the destination is not local the Determinizing Scheduler will have to send a message to the remove destination. The message will contain the sent value (v) as well as a timestamp indicating then the value should be logically received. The recv tasks are the simplest to process: The Determinizing Scheduler sets the field variable of m representing port p with the value v (if v is a complex or record type it can just update a reference).

5.3.2 Correctness

As mentioned at the start of Section 5.3 the ODSDL semantics define the timing of events relative to a discrete global timeline but in reality the clocks distributed througout the system will not always agree. So when we say that the Determinizing Scheduler implements the ODSDL's LET semantics we actually mean that, for a given execution of an ODSDL application, all the semantic events happen and they happen when all the clocks think they should according to the local value of their clocks. We call this notion *logical execution time (LET) modulo clock syn-*

chronization . First we make this notion precise (Definition 5.3.2), then we give an informal proof that the Determinizing Scheduler implements LET modulo clock synchronization.

Definition 5.3.1 (Unit Floor)**.** The unit floor function $\lfloor x \rfloor_u$ gives the integer portion of the number x assuming x is converted to a quantity in units u. For example if $x = 5.67$ milliseconds (ms) then $\lfloor x \rfloor_{ms} = 5$.

The unit floor function (Definition 5.3.1) is for convenience. It lets us compare the discrete value of two clocks regardless of their units.

Definition 5.3.2 (LET Modulo Clock Synchronization)**.** Let $\mathcal{C}$ be the set of system clocks. Let R be the relation defined by the semantic event and state of an ODSDL program. That program is executing with *Deterministic Logical Time (DLT) modulo Clock-Sync* if for each element (E, t) of the event streams of R each event $e \in E$ happens (i) and happens when $\forall_{c \in \mathcal{C}} \lfloor c \rfloor_{ms} = t$ (ii).

Essentially, Defition 5.3.2 means that if there is some event e supposed to happen at time t according to the ODSDL semantics, it will happen when all clocks in the system, rounded down to the nearest millisecond, read t.

Definition 5.3.3 (ϵ-Precise Real-Time Clock)**.** An ϵ-precise real-time clock c is counter that increases its value monotonically with a rate approximating 1. The counter must be able to represent increments at least as small as ϵ. Futhermore $\epsilon < 1/2$ milliseconds.

Definition 5.3.4 (ϵ-synchronized clock)**.** A set of ϵ-precise clocks $\mathcal{C}$ are ϵ-synchronized if it is always the case that $\forall_{c_i, c_j \in \mathcal{C}} \mid c_i - c_j \mid \leq \epsilon$.

Definiton 5.3.3 defines the type of real-time clock needed by the Determinizing Scheduler. We say that the the clock increases at a rate approximating 1 to account for small variations seen in real physical clocks. We will require that clocks have enough precision to represent the maximum discrepency (ϵ) between clocks in the system. Definition 5.3.4 is a standard definition of clock synchronization. We just require that the clocks be ϵ-precise and be synchronized within ϵ.

Before we proceed with proving that the Determinizing Scheduler implements logical execution time (LET) modulo clock synchronization, we need to establish what moments of time synchronized clocks are guaranteed to read the same value (Lemma 5.3.1).

Lemma 5.3.1 (ϵ-Offset Equality)**.** Assume two clocks c_i, c_j are ϵ-synchronized and $\epsilon < 1/2$ millisecond. If an event happens at some time when $c_i = t$ s.t. $\lfloor c_i \rfloor_{ms} + \epsilon < t < \lfloor c_i \rfloor_{ms} + (1 - \epsilon)$ then $\lfloor c_i \rfloor_{ms} = \lfloor c_j \rfloor_{ms}$

Proof. Assume towards a contradiction that $\mid c_i - c_j \mid \leq \epsilon$ and for some valuation of c_i where it is the case that $\lfloor c_i \rfloor_{ms} + \epsilon < c_i < \lfloor c_i \rfloor_{ms} + (1 - \epsilon)$ but $\lfloor c_i \rfloor_{ms} \neq \lfloor c_j \rfloor_{ms}$. W.o.l.o.g. assume $c_i \leq c_j$. Then $c_j \leq c_i + \epsilon$. Then it must be that $\lfloor c_i \rfloor_{ms} \geq \lfloor c_i + \epsilon \rfloor_{ms}$ which is a contradiction because we have that $\lfloor c_i \rfloor_{ms} + \epsilon < c_i < \lfloor c_i \rfloor_{ms} + (1 - \epsilon)$ and $\epsilon < 1$. □

Now we can prove that the Determinizing Scheduler commits events at the right time and when all clocks agree (Lemma 5.3.2).

Lemma 5.3.2 (The Determinizing Scheduler Commits Events at the Correct Time)**.** Let $\mathcal{C}$ be the set of system clocks. Assume all clocks $c_i, c_j \in \mathcal{C}$ are ϵ-precise and $\mid c_i - c_j \mid \leq \epsilon$. Let (E, t) be any timestamped queue of microtasks stored in the

Determinizing Scheduler's pending microtask queue. Then each $e \in E$ will be started *and* finished processing when $\underset{c \in \mathcal{C}}{\forall} \lfloor c \rfloor_{ms} = t$.

Proof. Observe that the Determinizing scheduler will start processing all microtasks $e \in E$ when the local clock $c > t + \epsilon$ and it will finish processing all the tasks when $c < t + (1 - \epsilon)$. The conclusion follows by application of Lemma 5.3.1. □

Finally, using Lemma 5.3.2 we prove the overall correctness of the Determinizing Scheduler. The proof of Theorem 5.3.3 is straight forward. Since we know that the Determinzing Scheduler will always execute the microtasks at the right time (Lemma 5.3.2) we only need to check that it generates the correct *effects* for each type of event. The proof simply compares the effect required by the semantics to the effect created by the scheduler.

Theorem 5.3.3 (The MDCF implements DLT modulo Clock Sync.)**.** Let $\mathcal{C}$ be the set of system clocks. Let R be the set of the $\mathsf{merge}s$ of the input / output event streams of an ODSDL program. Assume all clocks $c_i, c_j \in \mathcal{C}$ are ϵ-precise and $\mid c_i - c_j \mid \leq \epsilon$. Then for each element (E, t) of any event stream $s_e \in R$ each event $e \in E$ happens (i) and happens when $\underset{c \in \mathcal{C}}{\forall} \lfloor c \rfloor_{ms} = t$ (ii).

Proof. Consider any element (E, t) of s_e from R.

- *Case* $\mathsf{send}\,(m_1, p_1, v) \in E$. Recall that for a dataflow $f = m_1.p_1$ **flowsto** $m_2.p_2$ **delay** d it must be the case there is some element the event stream $(E', t + d)$ and $\mathsf{recv}\,(m_2, p_2, v) \in E'$. We know by the definition of Algorithm 7 that the Determinizing Scheduler will schedule a microtask for $\mathsf{recv}\,(m_2, p_2, v)$ to happen at t_i. We know by Lemma

5.3.2 that $\mathsf{send}\,(m_1, p_1, v)$ will be committed when $\bigvee_{c \in \mathcal{C}} \lfloor c \rfloor_{ms} = t$ and that $\mathsf{recv}\,(m_2, p_2, v)$ will be committed when $\bigvee_{c \in \mathcal{C}} \lfloor c \rfloor_{ms} = t_i$

- *Case* $\mathsf{recv}\,(m_1, p_1, v) \in E$. Recall that the semantics require that any task scheduled at t see $m_1.p_1 = v$. By Algorithm 7 the Determinizing Scheduler will always update $m_1.p_1 = v$ before any tasks are dispatched because it defers all task dispatches to the end of the processing cycle. Also recall that updates to $m_1.p_1$ may trigger m_1's tasks at time t. By Algorithm 7 we know that if an update to $m_1.p_1$ triggers the execution of the module's task the task will be dispatched at the end of the Determinizing Scheduler's processing cycle. By Lemma 5.3.2 we know that the task $\mathsf{recv}\,(m_1, p_1, v)$ will commit when $\bigvee_{c \in \mathcal{C}} \lfloor c \rfloor_{ms} = t$.

- *Case* $\mathsf{sched}\,(m) \in E$. Recall that the semantics require all resulting send event from the task execution happen at $t + D$ (D is the task's deadline). By the definition the Determinizing Scheduler will dispatch the task for module m at time t with a copy of the port variables from after all receive events for t have been processed which ensures that the task will execute in a state where the values of the ports are "frozen" to a value defined by the semantics. All calls to `send` by ODSDL applications are trapped and scheduled to happen at $t + D$ therefore by Lemma 5.3.2 $\mathsf{sched}\,(m)$ will commit when $\bigvee_{c \in \mathcal{C}} \lfloor c \rfloor_{ms} = t$ and all the resulting send events will commit when $\bigvee_{c \in \mathcal{C}} \lfloor c \rfloor_{ms} = t + D$.

□

5.4 Practical Real-Time Scheduling

The MDCF must schedule tasks and messages to ensure that all processing and communication always completes before application specified deadlines. Unlike many real-time systems whose configuration is fixed from the factory, the MDCF is highly dynamic: as users start and stop applications the MDCF needs to automatically (re)-allocate resources in a way that won't disrupt the operation of the other applications. Typically, this has been the domain of *dynamic scheduling algorithms* such as Earliest Deadline First (EDF) [130] or Least Slack First [143].

Dynamic scheduling algorithms make scheduling decisions by reacting to the state of the system. For example, in EDF, whenever a task activates or finishes, the scheduler looks at all active tasks and sets the one with the closest impending deadline to the highest priority. This makes it easy to add or remove new tasks at runtime.

Unfortunately, many real-time operating systems and networking technologies do not implement a dynamic scheduling discipline. Instead, fixed-priority scheduling is used. In fixed-priority scheduling, the system designer assigns each task (or data-stream) a priority. Then, during runtime, the scheduler will use the assigned priority to determine execution order. There have been several algorithms and techniques developed to assign priorities in fixed-priority scheduling such as Rate Monotonic (RM) [130] or Deadline Monotonic (DM) where priorities are assigned with increasing task rate (*i.e.,* period) or deadline.

If the number of available priorities were unbounded then that would not be such a problem. Unfortunately most real operating systems and networking technologies only provide a limited number of distinct priority levels. For example, the ethernet switches controlled by MIDAS typically only provide 8 different pri-

ority levels [187]. This presents a problem for the us: often the number of tasks or data-flows required to support an application running on the MDCF are much larger than the number of available priorities. This means tasks will have to share priorities. This also means that as users start and stop applications priorities will potentially need to be re-shuffled in a way that does not cause tasks or flows to miss deadlines.

In this section we will show how to extend existing fixed priority scheduling techniques to make them applicable to the dynamic environment the MDCF/MIDAS operates in. Specifically, we develop a priority assignment algorithm for the limited priority setting that is asymptotically faster than the state-of-the-art. Then we devise a simple priority reassignment protocol that ensures tasks or flows will not miss deadlines during priority reassignment. Finally, we adapt existing scheduling and schedulability techniques to work with the primitives and packet-processing semantics of OpenFlow switches.

5.4.1 Fixed Priority Scheduling Techniques

In this section we review basic fixed-priority scheduling concepts from the real-time scheduling literature and then develop a new priority assignment algorithm for the limited priority environment.

Preemptive Fixed Priority, Real-Time Scheduling Problems and the Periodic Task Model

Preemptive fixed priority scheduling is a scheduling discipline where each task is assigned a priority. If the processor is executing some task and another task arrives with higher priority then the original task is suspended (*i.e.,* preempted)

and the processor starts executing the higher priority task.

Let $\mathrm{T} = \{T_0, T_1, \ldots, T_n\}$ be a set of tasks with each $T_i = (P_i, C_i, D_i)$. Each task defines an infinite sequence of job arrivals where P_i is the minimum separation between consecutive arrivals, C_i is the number of timeunits it would take to finish the job assuming there is no resource contention, and D_i is the relative deadline of each job. When T_i is specified with only P_i, C_i and D_i then it is possible the the first job of all tasks arrive at the same moment. These task sets are called *synchronous*. Sometimes a term J_i is associated with T_i to specify the *release jitter* of T_i. Sometimes a term A_i is used to specify the arrival offset of the first job of T_i.

Response Time Analysis

Response Time Analysis (RTA) [25] is used to determine the worst case response time of a task in a task set given how each task in the task set is prioritized. Formally, given a task set T where each task $T_i \in \mathrm{T}$ is the tuple (P_i, C_i, D_i) such that T_i is the minimum separation between T_i's release, C_i is the computation cost of T_i and $D_i \leq P_i$ is the deadline, the worst case response time of a task T_i is determined by calculating the smallest R_i such that the following response time equation is satisfied ($hp\,(i)$ is the function that returns the set of tasks with priority higher than T_i's):

$$R_i = C_i + \sum_{T_j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \tag{5.1}$$

If $R_i > D_i$ then we say that T_i is not feasible. If one wants to determine what T_i's worst case response time would be if it were sharing priority level p with other tasks, the response time equation must be modified with an extra term accounting

Algorithm 8 RTA-based interference check

1: Let $hp \leftarrow \{T_j \in \tau | j < y\}$ ▷ j has a higher priority than i
2: $C_p \leftarrow \sum_{T_p \in p_i} C_p$ ▷ tasks at the same priority interfere with each other
3: $R_y^0 \leftarrow C_y + C_p$
4: **while** $R_y^{k+1} > R_y^k \wedge R_y^{k+1} \leq D_y$ **do**
5: $\quad R_y^{k+1} \leftarrow C_y + C_p + \sum_{T_j \in hp} \left\lceil \frac{R_y^k}{P_j} \right\rceil C_j$
6: **end while**
7: **if** $R_y^{k+1} > D_y$ **then**
8: $\quad$ return $true$
9: **else**
10: $\quad$ return $false$
11: **end if**

for the extra delays caused by tasks executing at level p. If one assumes that jobs sharing a priority level are executed in the order that they were dispatched then the following equation can be used:

$$R_i = \left(\sum_{T_y \in p} C_y \right) + C_i + \sum_{T_j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \tag{5.2}$$

Algorithm 8 implements the fixed point computation for equation 5.2 and then checks whether $R_i > D_i$.

Response Time Analysis can be used as the basis for a task-set schedulability test: if all tasks in the set have a worst-case response time less then their deadline under a priority assignment then the task-set is schedulable (also called feasible) under that assignment. The RTA equations and algortihms described here are *exact* which means that if $RTA(T_i) = R_i$ then it is possible that there is a schedule execution where T_i takes R_i to complete and there is no execution where T_i takes more than R_i. Exact schedulability tests are useful because they provide a true

representation of worst-case resource usage which enables more efficient use of resources. The downside of exact tests is their computational complexity: Exact schedulability for fixed priority task sets is an NP-Hard problem. While the fixed-point algorithm 8 usually converges quickly in practice the number of times it needs to be called should be minimized, especially in interactive systems such as the MDCF.

Deadline Monotonic Scheduling

Given a task set T as in Section 5.4.1, Deadline Monotonic (DM) scheduling assigns higher priorities to tasks with smaller relative deadlines, *i.e.,* if $D_i < D_j$ then $\mathsf{pri}\,(T_i) > \mathsf{pri}\,(T_j)$. If the tasks arrive synchronously without and release jitter then DM scheduling is optimal in the sense that if there exists some priority assignment under which a task set is feasible, then it will be feasible under the DM assignment also. The proof of optimality is straight forward. It works by first assuming there is some non-DM priority assignment under which T is feasible. Then it is easy to show that the non-DM schedule can be transformed into the DM schedule by swapping each out of order pair one by one. After each swap, T is still feasible: Assume $\mathsf{pri}\,(T_i) > \mathsf{pri}\,(T_j)$ and $D_i > D_j$. Since T is feasible under this assigment it must be the case that there is enough slack at $\mathsf{pri}\,(T_j)$ for T_j to finish by D_j. If we swap the priorities of T_i and T_j we know that T_j will still meet its deadline because there are fewer higher priority tasks to interfere with it. T_i will still meet its deadline because there is was enough slack at its new priority level for $C_i + C_j$ to complete by D_j. Since $D_j < D_i$ we know there is still enough free processor to complete C_i units of work by D_i. Deadline Monotonic scheduling is useful because it is fast: The priority assignment can

be determined simply by sorting the task set by the deadlines. Unfortunately Deadline-Monotonic scheduling assumes that there exists at least one priority for each task in the task set.

Audsely's Algorithm

The one well-known method of finding a minimum priority assignment for fixed priority scheduling is Audsley's algorithm [27]. Audsleys algorithm is conceptually simple. Let $n = \mid \mathrm{T} \mid$. Audsley's algorithm proceeds by trying to find some task $T_i \in \mathrm{T}$ that is feasible if all the remaining tasks in T are at a higher priority. This can be checked by performing response time analysis for T_i with $hp\left(T_i\right) = \mathrm{T} - T_i$. If such a task is found, it is assigned priority n. This process continues until no task can be found that is feasible at priority n. The algorithm then proceeds to fill priority level $n - 1$ and so on (see Algorithm 9).

If an exact feasibility test is used then Audsley's algorithm is optimal in two ways. First, if there exists a feasible fixed priority assignment the algorithm will find it. Second, it will always minimize the number of priority levels needed: If there exists an assignment using $1 \leq p \leq n$ levels then the algorithm will find an assignment with $1 \leq p' \leq p$ levels. Unfortunately, as stated earlier, exact feasibility tests have the potential to be expensive and Audsley's algorithm will perform $O\left(n^2\right)$ feasibility tests in the worst case. As it turns out Audsely's algorithm is more powerful than what is needed for the MDCF: the algorithm will find optimal assignments for task sets specified with offsets for the arrival of each task's first job. As described in the next subsection if this assumption can be relaxed (*i.e.,* if it is possible for the first job of each task to arrive at the same time) then we can get an order of magnitude theoretical speed up and get by with only

Algorithm 9 Audsley's Algorithm

```
1:  pri ← ∅                                          ▷ initial priority assignment
2:  T′ ← T
3:  for j ∈ [n . . . 1] do
4:      unassigned ← true
5:      for T_i ∈ T′ do
6:          if T_i is feasible with hp(T_i) = T′ − T_i then
7:              prin ← prin ∪ (T_i ↦ j)
8:              T′ ← T′ − T_i
9:              unassigned ← false
10:         end if
11:     end for
12:     if unassigned then
13:         return ∅                                 ▷ No feasible assignment exists
14:     end if
15: end for
16: return pri
```

$O(n)$ feasibility checks.

A Faster Priority Assignment Technique

If it is possible for the first job of all tasks to arrive at the same time then the minimal priority assignment for implicit deadline task-sets can be determined with only $O(n)$ feasibility tests. The algorithm works as follows, first assign each task $T_i \in \mathrm{T}$ a priority according to its Deadline Monotonic ordering. Then if T is feasible under the Deadline Monotonic partition the ordering by greedily combining priority levels from the lowest to the highest as long as the combination does not cause any task to miss a deadline: Start from the lowest priority task T_n

Algorithm 10 Priority Partition Algorithm

```
1:  pri = DM (T)                                  ▷ Start with the Deadline Monotonic assignment
2:  n =| T |
3:  p ← n                                         ▷ p tracks current priority level
4:  for i = (n − 1) → 1 do
5:      if Any task at p interferes with task T_i then
6:          p ← i
7:      else
8:          pri ← pri [T_i ↦ p]                   ▷ Reduces range (pri) by 1
9:      end if
10: end for return pri
```

and check to see if T is still feasible when $\mathsf{pri}\,(T_n) = \mathsf{pri}\,(T_{n-1})$. If T_n and T_{n-1} can share the same priority level reassign T_{n-1} to T_n's priority and then try to merge T_{n-2}. If T_n may cause T_{n-1} to violate its deadline then they cannot share a priority level so the algorithm proceeds by trying to merge T_{n-1} with T_{n-2} and so on (See the PriorityParition procedure in Algorithm 10).

It is easy to see that PriorityParition produces a feasible priority assignment: Assume w.o.l.og that tasks T_i and T_j are merged onto the same priority level. Assume that T_i had a higher deadline monotonic priority. T_i won't cause T_i to miss a deadline if they share a priority level because T_i already maximally interfered with T_j due to its higher priority. T_i won't miss a deadline due to T_j (or any other task at the same priority) because PriorityParition would not have merged T_j with T_i if that was the case.

The argument that PriorityParition applied to a deadline monotonic assignment (DM-PriorityParition) minimizes the number of required priorities is less obvious. It is sufficient to prove two facts: First, PriorityParition produces the

smallest *order-preserving* partitioning. An order-preserving partitioning preserves the relative priority assignment of the original assignment up the partitioning:

Definition 5.4.1 (Order-Preserving Partitioning)**.** $\mathsf{pri}' = \mathsf{Partition}\,(\mathsf{pri}, \mathrm{T})$ is order-preserving if for any $T_i, T_j \in \mathrm{T}$, $\mathsf{pri}\,(T_i) < \mathsf{pri}\,(T_j)$ implies that $\mathsf{pri}'\,(T_i) \leq \mathsf{pri}'\,(T_j)$

Second, any partitioning can be converted to a deadline-monotonic order-preserving partitioning that uses the same or fewer priority levels. What

Theorem 5.4.1. Let $\mathsf{pri}' = \mathsf{Partition}\,(\mathsf{pri}, \mathrm{T})$. Let pri'' be some other order-preserving partitioning of pri, then pri' uses the same or fewer priorities than pri'' (*i.e.,* $\mathsf{range}\,(\mathsf{pri}') \leq \mathsf{range}\,(\mathsf{pri}'')$).

Proof. Assume $\mathsf{range}\,(\mathsf{pri}'') \geq \mathsf{range}\,(\mathsf{pri}')$. The proof proceeds by transforming pri'' into pri'. Because both pri'' and pri' are order-preserving the only way they can differ is that there must be some task T_i in pri'' at a higher relative priority partition than in pri'. If T_i can be demoted in pri'' then $\mathsf{range}\,(\mathsf{pri}'')$ will either stay the same or get reduced by one. Note, that if T_i can be demoted in pri'' then PriorityParition would have made the same choice. The process of demotion can be applied iteratively until $\mathsf{pri}'' = \mathsf{pri}$. □

Theorem 5.4.2. If pri' is a feasible mapping of τ, then there exists a feasible deadline monotonic mapping $dm - pri$ such that $|dm - pri| = |pri'|$.

Proof. The proof works by transforming any feasible non-deadline monotonic mapping of T, pri' into a mapping pri^{dm} which preserves the deadline monotonic ordering a while maintaining $\mathsf{range}\,(\mathsf{pri}') = \mathsf{range}\left(\mathsf{pri}^{dm}\right)$. We use the following dual index scheme to track the priority assignment of each task in the mapping: T_{ij} denotes the task that is assigned to the ith priority level (*i.e.,* $T_{ij} \in p_i$) and

that T_{ij} is the jth task in p_i. In the following example, $\mathsf{range}\,(\mathsf{pri}') = 2$ and each priority level has two tasks:

$$\mathsf{pri}' = \{T_{11} \mapsto 1, T_{12} \mapsto 1, T_{21} \mapsto 2, T_{22} \mapsto 2\}$$

For the purposes of this proof assume that each list p_i corresponding to the tasks assigned to priority i in any partitioning is sorted according to the deadline monotonic order (*i.e.,* for the example pri' above, $D_{11} \leq D_{12}$ and $D_{21} \leq D_{22}$). Next imagine the list of tasks that would be created if each p_x s.t. $x \in \mathsf{range}\,(\mathsf{pri}')$ were laid down in order, end to end. For example, if pri' is the example mapping above, then the imaginary list is:

$$\{T_{11}, T_{12}, T_{21}, T_{22}\}$$

In order to reason about that task list mathematically we define an index mapping function, $IM\,(i, j, \mathsf{pri}',)$ which will allow us to recover the imaginary list position of any mapped T_{xy}.

$$IM\,(i, j, \mathsf{pri}',) = \left(\sum_{x=1}^{i-1} |p_x|\right) + j$$

For convenience we will also define the function $\mathsf{Index}\,(T_{ij}, \mathsf{pri})$ which returns the index of T_{ij} in the original non-partitioned priority assignment pri:

$$\mathsf{Index}\,(T_{ij}, \mathsf{pri}) = \mathsf{pri}\,(T_{ij})$$

We can now define a mapping with deadline-monotonic ordering as follows:

The proof proceeds as follows: First assume there is some feasible mapping pri' such that:

$$\exists_{T_{ij} \in \mathrm{T}} \quad \mathsf{pri}^{dm}\left(T_{ij}\right) \neq IM\left(i, j, \mathsf{pri}'\right)$$

That is, pri' does not preserve the deadline monotonic ordering. pri' can be transformed into the deadline monotonic mapping $\mathsf{pri}^{dm'}$ without any additional priority levels by iteratively applying the following process:

1. Locate the lowest priority out of order task: Find the largest j and i s.t. $IM\left(i, j, \mathsf{pri}'\right) \neq \mathsf{pri}^{dm}\left(T_{ij}\right)$.

2. The task found in step 1 is out of order because at least one task which would have a lower deadline monotonic priority has been assigned to some higher priority level in pri'. Find this task and reassign its priority to i. Formally, move the task T_{xy} s.t. $x < i$ and $\mathsf{Index}\left(T_{xy}, \mathsf{pri}^{dm'}\right) > \mathsf{Index}\left(T_{ij}, \mathsf{pri}^{dm'}\right)$. Figure 5.12 illustrates this step.

After repeated application of steps 1 & 2, pri' will be transformed into the deadline monotonic preserving mapping $\mathsf{pri}^{dm'}$, and no new priority levels will be required. Is the $\mathsf{pri}^{dm'}$ created by this transformation Feasible? Yes it is.

Each time task T_{xy} is demoted to priority i the feasibility of any task $T_{ij'} \in p_i$ are not affected because they are already assumed feasible when subject to interference from task T_{xy} because the task set is feasible when T_{xy} has higher priority than any $T_{ij'}$. Can T_{xy} miss any deadline after it has been demoted to priority level i? No. Because pri' is feasible then:

$$\left(\sum_{T_{ij'} \in p_i} C_{ij'}\right) \leq D_{ij} - I_{hp(i)}^{D_{ij}}$$

That is, the total cost of every task executing at priority level i plus the interference over $[0, D_{ij}]$ by higher priority tasks must be less than or equal to T_{ij}'s deadline. When T_{xy} is removed from priority level x, and then add that task to priority level i the equation now becomes:

$$\left(\sum_{T_{ij'} \in p_i} C_{ij'}\right) + C_{xy} \leq D_{ij} - I_{hp(i)}^{D_{ij}} + C_{xy}$$

because T_{xy} no longer executes with a higher priority than i, but still contributes at most one job's worth of delay because each job at a given priority level is executed in FIFO order.

Finally, because T_{xy} was out of deadline monotonic order with respect to T_{ij} it must be the case that $D_{ij} \leq D_{xy}$. Therefore:

$$\left(\sum_{T_{ij'} \in p_i} C_{ij'}\right) + C_{xy} \leq D_{xy} - I_{hp(i)}^{D_{ij}} + C_{xy}$$

Thus T_{xy} will never miss a deadline if its priority is demoted from level x to level i. □

Finally, we can now prove that $PriorityPartition\left(\tau_{list}^{dm}\right)$ is the feasible priority assignment of τ that uses the fewest number of priorities:

Theorem 5.4.3. If pri' is any feasible priority assignment of T, then $\mathsf{range}\left(\mathsf{pri}^{dm'}\right) \leq \mathsf{range}\left(\mathsf{pri}'\right)$.

Proof. The theorem follows directly from the optimality of the deadline monotonic ordering and application of Theorem 5.4.2. □

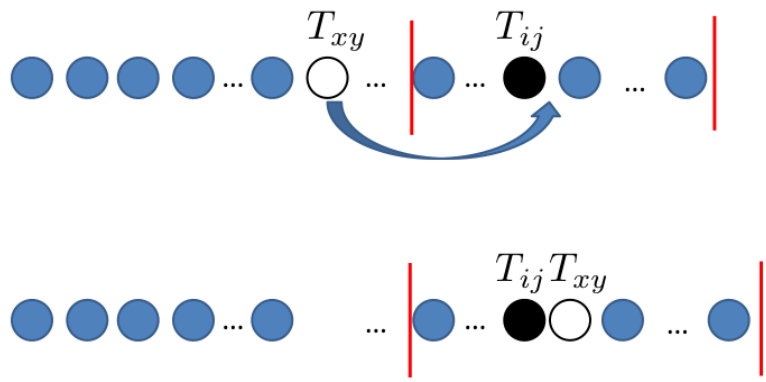


Figure 5.12: Reordering a task map.

A Simple Mode Change Protocol

Fixed priority scheduling, limited priority levels and minimial priority assignments introduce a small wrinkle for systems that change their task-sets over the course of their runtime (*i.e.,* perform a scheduling mode change). The wrinkle is this: due to the finite priorities, the priority assignment may assign a different priority to the same task even if it executes in both the source mode and the target mode. How do we ensure that task will still meet its deadline as new tasks are added (or removed) and its priority modified? In general, solving this problem is non-trivial and there are a number of complex mode change protocols that can deal with this problem under a variety of different assumptions [165].

Fortunately, the MDCF only exhibits two types of scheduling mode changes: tasks are added (*i.e.,* when a new application is launched) or tasks are removed (*i.e.,* when an application exits). Assume that the source mode taskset T is feasible under priority assignment pri and destination mode taskset T' is feasible under pri'. If $\mathrm{T} \subseteq \mathrm{T}'$ then the mode change can be performed simply by assigning each task

$T_i \in \mathrm{T}$ to priority $\mathsf{pri}'(T_i)$ in reverse priority order (*i.e.,* starting with the task that has the lowest priority in the new mode). If $\mathrm{T}' \subseteq \mathrm{T}$ then the tasks should be assigned in priority order.

5.4.2 Flow Scheduling

A valid flow scheduler must perform three tasks: First, when a subscriber comes online and requests a subscription to an existing topic the flow scheduler must generate a candidate network configuration using OpenFlow configuration primitives. Second, the scheduler must analyze the new configuration and determine if it guarantees the timing constraints of all admitted flows plus the new one. Third, if the new configuration is acceptable the scheduler must reconfigure the network carefully so no constraints of existing flows are violated during the reconfiguration. All three of these activities are non-trivial: Distributed scheduling is known to be NP-Hard if an optimal schedule is desired and there are no known exact schedulability tests for the general network setting [41].

In light of these difficulties, we do not describe an optimal approach. Instead, we focus on a strategy that is both fast (*i.e.,* polynomial-time in the size of the network), exploits the types of configuration primitives by OpenFlow, and allows for the safe transition from one valid configuration to another. Analysis and improvement of the approach in terms of network utilization and schedulability is left for future work. We start be describing how the strategy would apply to a network consisting of a single switch, then extend it to the multi-switch case.

Single Switch Scheduling

The flow scheduler generates a candidate network configuration in several phases. First, for each publish-subscribe relationship the flow scheduler queries the OpenFlow controller to determine what switch port each publisher and is connected to and the network address associated with a given topic. Then, for each publisher $\mathcal{P}$ publishing to $\mathcal{T}$, the flow scheduler configures a rate-limiter. The rate-limiter is configured with a maximum burst size B and and maximum rate R, and is set to apply to all packets that enter the switch on the port connected to $\mathcal{P}$ destined to the network address associated with $\mathcal{T}$. If a publisher $\mathcal{P}$ specifies a minimum separation between each message of $minSep$, and maximum message size of M, then the burst size and rate are set as follows:

$$B = M, \;\; R = \frac{M}{minSep}$$

This allows $\mathcal{P}$ to burst its entire message onto the network while ensuring that $\mathcal{P}$ cannot overload the network if $\mathcal{P}$ becomes a babbling idiot.

Before we can describe how the flow scheduler prioritizes flows we need to explain how to calculate upper bounds on the worst case latency of message. Latency in a switched network has a number of sources. The first is due to the bandwidth of the network link. The second is due to the physical wire that connects a network node to a switch: an electrical signal takes time to propagate along a wire (in most networks the latency effects of the wires are small because they are relatively short). The third is the multiplexing latency of the switch. Switch multiplexing latency is the time it takes a switch to move a bit entering the switch on one port to the output queue of another. On modern non-blocking switches this is usually on the order of several microseconds. Finally, there is queuing la-

tency, which is the amount of time a message spends waiting in an egress queue. In a modern switched Ethernet all these latencies are fixed (*i.e.,* do not change due to network load) except for queuing latency. Messages placed from different flows placed on queues associated with the same switch port are in contention for shared "forwarding resources." We now formally define the fixed latency, queuing latency, and end-to-end latency for a single switch.

Definition 5.4.2 (Wire Latency)**.** The function $w\left(N_1, N_2\right)$ denotes the signal propagation latency between network stations N_1 and N_2. A network station can be either a switch, or a publisher/subscriber.

Definition 5.4.3 (Fixed Latency)**.** Let $f = \left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)$. Let the maximum message size of a message publish to topic $\mathcal{T}$ be M. Then the fixed portion of the end-to-end latency, denoted $\mathcal{L}_F\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)$, between $\mathcal{P}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$ is:

$$\mathcal{L}_F\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \frac{M}{C} + w\left(\mathcal{P}_{\mathcal{T}}, s\right) + w\left(\mathcal{S}_{\mathcal{T}}, s\right) + s^{mux} \tag{5.3}$$

where C is the network bandwidth and s^{mux} is the multiplexing latency of switch s

Definition 5.4.4 (Queing Latency)**.** Let $\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)$ be the flow from $\mathcal{P}_{\mathcal{T}}$ to $\mathcal{S}_{\mathcal{T}}$, and let $s\left(i\right)$ be the ith port on switch s which the flow is routed out of, then the queuing latency of the flow $\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)$ with priority p at switch/port $s\left(i\right)$ is $\mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s\left(i\right), p\right)$.

Definition 5.4.5 (End-to-End Latency)**.** The end-to-end latency is the sum of the fixed and queuing latency:

$$\mathcal{L}_{e2e}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \mathcal{L}_F\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) + \mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s\left(i\right), p\right) \tag{5.4}$$

How can we calculate $\mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s\left(i\right), p\right)$? We adapt an approximate technique for calculating the response time of a task under fixed priority scheduling on a uniprocessor. In [36], Bini *et al.* provide a linear equation for calculating an upperbound worst case response time of a task. Assuming P_i is the minimum separation between consecutive arrivals of task T_i, E_i is the worst case execution time and $hp\left(i\right)$ is the set of tasks assigned priority higher than T_i, then the response time R_i is bounded from above by:

$$R_i^{ub} = \frac{E_i + \sum\limits_{j \in hp(i)} E_j \left(1 - \frac{E_j}{P_j}\right)}{1 - \sum\limits_{j \in hp(i)} \frac{E_j}{P_j}} \tag{5.5}$$

This equation is useful in our application because the per-task workload approximations Bini *et al.* used to derive the response time bound also approximate the traffic pattern of a flow conforming to a rate-limiter. We then transform Equation 5.6 into a worst-case bound on latency due to queuing by substituting message sizes divided by bandwidth for execution cost, $minSep$ for the periods, and subtracting the overall message transmission cost for our flow[3] Additionally, let $hp\left(p, s\left(i\right)\right)$ be the set of flows with priority higher than p at port i on switch s:

$$\mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s\left(i\right), p\right)^{ub} = \frac{\frac{M_{\mathcal{T}}}{C} + \sum\limits_{j \in hp(p,s(i))} \frac{M_j}{C} \left(1 - \frac{\frac{B_j}{C}}{minSep_j}\right)}{1 - \sum\limits_{j \in hp(p,s(i))} \frac{\frac{B_j}{C}}{minSep_j}} - \frac{M_{\mathcal{T}}}{C} \tag{5.6}$$

We can now use the upper-bound on worst-case switch latency to determine how to prioritize each flow. Common techniques for priority assignment in real-

[3]This is because the response time for a task also takes into account the execution time for that task. We only want a upper bound on the interference.

time systems include the Rate Monotonic (RM) and Deadline Monotonic orderings (DM) [26]. Unfortunately, both RM and DM theory require that each flow is assigned a unique priority. This is not possible on real networking hardware: most Ethernet switches only provide 8 priority queues per port for egress traffic. To overcome this limitation, we use Audsley's Optimal Priority Assignment (OPA) algorithm [24]. OPA has two desireable properties: It is optimal locally (if a flow set will meet its latency requirement at a single switch under any fixed-priority configuration it will also under OPA) and it minimizes the number or priority levels required to schedule the flow set. Because each port of the switch is independent in terms of its egress queueing, we only need to differentiate the priorities of flows exiting the switch on the same port.

We now describe a version of Audsley's OPA adapted to assign priorities to flows in an OpenFlow switch. Our modified OPA takes as input a set of flows (denoted $\mathcal{F}$) forwarded out of the same port. OPA starts by attempting to assign flows to the lowest priority level. If a flow f can exceeed its latency bounds at a given priority level, OPA moves on and will attempt to assign that flow a higher priority later. Conservatively, a flow $(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}})$ can miss its latency bounds if $\mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s\left(i\right), p\right)^{ub} + \mathcal{L}_F\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) > \mathcal{L}_{max}\left(\mathcal{S}_{\mathcal{T}}\right)$. If OPA exits before assigning a priority to every flow, then the flow set is not schedulable with *any* fixed priority assignment. If the number of priority levels required to schedule $\mathcal{F}$ is greater than the number of priorities provided by the switch, then the flow scheduler deems the flow set unschedulable.

Before admitting a new subscriber the flow scheduler must reconfigure the network to accomodate the new flow without causing existing flows to violate their latency requirements. Existing flows must be migrated to their new priorities

in a specific order to avoid *priority inversions*. To safely accomplish the reconfiguration the flow scheduler maps existing priorities according to their priority assignment in the new configuration: flows with lower priority are reprioritized first.

Extension to Multi-switch

The prototype flow scheduler supports real-time guarantees on networks consisting of multiple switches by tranforming the distributed scheduling problem into a sequence of local (*i.e.,* single switch) scheduling problems. Before we proceed we modify Equations 5.7 and 5.4 to describe the sources of latency for a flow that is forwarded through a sequence of switches. As in the single switch case there are fixed and queuing sources of latency:

Definition 5.4.6 (Multiswitch Fixed Latency)**.** Let ρ be a path of length m through the network from $\mathcal{P}_{\mathcal{T}}$ to $\mathcal{S}_{\mathcal{T}}$. Let N_k be the kth network node (switch or publisher/subscriber) on ρ.Let the maximum message size of a message publish to topic $\mathcal{T}$ be M. Then the fixed portion of the end-to-end latency, denoted $\mathcal{L}_F^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right)$, between $\mathcal{P}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$ is:

$$\mathcal{L}_F^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \frac{M}{C} + \sum_{1<k\leq m} w\left(N_{k-1}, N_k\right) + \sum_{1<k<m} s_k^{mux} \tag{5.7}$$

Definition 5.4.7 (Multiswitch Queuing Latency Latency)**.** Let ρ be a path through the network with length m from $\mathcal{P}_{\mathcal{T}}$ to $\mathcal{S}_{\mathcal{T}}$. Then the queuing latency due to all the switches on ρ is the sum of all the queuing latencies of the switches along the path:

$$\mathcal{Q}^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \sum_{1<k<m} \mathcal{Q}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}, s_k\left(i\right), p_k\right) \tag{5.8}$$

Definition 5.4.8 (Total Multiswitch End-to-End Latency)**.** Let ρ be a path through the network crossing m switches. Then the total end-to-end latency due to both fixed and queuing delays along ρ is:

$$\mathcal{L}^{\rho}_{e2e}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) = \mathcal{Q}^{\rho}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) + \mathcal{L}^{\rho}_{F}\left(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}}\right) \tag{5.9}$$

Given these equations for end-to-end latency for flows crossing multiple switches we describe how MIDAS generates and applies network configurations for multi-switch networks. As mentioned earlier in this section distributed scheduling is in general quite difficult. Further complicating matters is that MIDAS must be able to reconfigure the the entire network without causing any QoS constraint violations for existing flows. This is challenging because the reconfiguration of an upstream switch will impact the worst case load on downstream switches. Imagine for example a simple network consisting of two switches s_1 and s_2. Now imagine some flow f forwarded along the path s_1, s_2. Say that the minimum separation between bursts of f at s_1 is 20ms and the worst case queuing latency at s_1 is 3ms. This means that the minimum separtion that could be observed by s_2 is 17ms (the case where the first burst of f is delayed the maximum amount and then the second burst is not delayed at all). Now assume a new flow f' is admitted to the network and it is prioritized higher than f on s_1. This will increase the worst-case queuing latency of f (*e.g.,* to 10ms) at s_1 and further contract the worst-case burst separation observed by s_2 (down to 10ms).

We avoid having to calculate network-wide side effects each time a new subscriber is admitted by transforming the distributed scheduling problem into a

sequence of local scheduling problems: When a subscriber $\mathcal{S}_{\mathcal{T}}$ requests a subscription to $\mathcal{T}$ with a latency constraint $\mathcal{L}_{max}(\mathcal{S}_{\mathcal{T}})$ we first calculate the shortest unweighted path ρ between $\mathcal{P}_{\mathcal{T}}$ and $\mathcal{S}_{\mathcal{T}}$. Next, we uniformly allot a portion $\mathcal{L}_{max}(\mathcal{S}_{\mathcal{T}})$ to each switch: for each switch s_k in ρ we calculate $\mathcal{L}_{max}(\mathcal{S}_{T})_{s_k}$ where:

$$\mathcal{L}_{max}(\mathcal{S}_{T})_{s_k} = \frac{\mathcal{L}_{max}(\mathcal{S}_{\mathcal{T}}) - \mathcal{L}_{F}^{\rho}(\mathcal{P}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}})}{|\rho|}$$

That is, we split the allowed queuing latency up evenly between all the switches along ρ. We now recursively calculate the worst case minimum separation observed at each switch on the path. Let $minSep_k$ be the minimum worst case separation of bursts at switch s_k then:

$$minSep_{k+1} = minSep_k - \mathcal{L}_{max}(\mathcal{S}_{T})_{s_k}$$

Finally, we apply the single switch schedulability, priority assignment and network reconfiguration algorithms using each $\mathcal{L}_{max}(\mathcal{S}_{T})_{s_k}$ and $minSep_k$ for the appropriate switch. Because we fixed the allotted switch queuing latency when the flow as admitted, the $minSep_k$ values will never change.

5.5 Evaluation & Performance Assesment

This section describes the experimental results of an evaluation of some of the MDCF / MIDAS' key features.

5.5.1 Scheduling and Resource Reservation in the RTMB

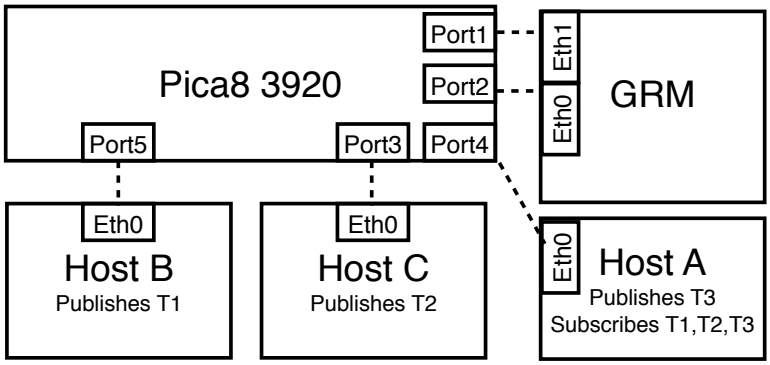


Figure 5.13: Experimental setup

We evalauted two aspects of MIDAS. First we wanted to see if the network scheduling used in the MIDAS improved the timing performance relative to that of a standard switch. Second, we wanted to see how robust the MIDAS timing guarantees are. In order to evaluate these two aspects we deployed the MIDAS on our OpenFlow test bench (Figure 5.13).

Our OpenFlow test-bench consists of 4 computers and an OpenFlow capable switch, a Pica8 P3290 [3]. Each of the 4 computers were plugged into the switches' *data-plane* ports (*i.e.,* OpenFlow managed ports). The GRM was also plugged into the *control-plane* port which carries OpenFlow management traffic. Measuring end-to-end timing in a distributed network accurately is challenging due to clock synchronization issues. We avoid these synchronization issues by exploiting OpenFlow to let us run publisher's and subscribers on the same hosts: we add an OpenFlow rule that causes the switch to intercept packets from certain flows, rewrite the packet headers, and then retransmit the packet back out the port it arrived on. This allows us to 'fool' the client; it can publish to $\mathcal{T}_x$ and subscribe to $\mathcal{T}_y$ but in reality it the messages being published to $\mathcal{T}_x$ are being sent back mod-

ified so they look as if they are from $\mathcal{T}_y$. This allows us to compare the timestamps of messages using the same system clock while still subjecting the message to the same queuing, multiplexing and wire latencies it would experience if it was being sent to another host.

All timing measurements we done on Host A. Host A was running real-time Linux with IBM's RTSJ-compliant Real-Time JVM. The RTMB client library on Host A was scheduled with the highest system priority using RTSJ Java's `NoHeapRealtimeThread`'s to ensure that they would not be interfered with by the Java garbage collector or other processes on the system. All timing measurements were made by using the system's millisecond precision real-time clock API. Prior to running our experimental scenarios we lower-bounded the amount of latency added by the Linux TCP/IP stack and the JVM by sending a message to the loopback interface. This latency was consistently 1ms, which means that observed latencies as recorded by the software are usually 1ms more than the actual network latency.

For each experiment we used the same 3 publishers each publishing to a different topic ($\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$) with a single host subscribing to each topic. Table 5.2 lists each topic, relevent QoS ($minSep$ of the publisher and max latency from the subscriber), and the bandwidth required by each. The publish-subscribe set is designed to be representative of a demanding plug and play medical system: $\mathcal{T}_1$ represents a high framerate/resolution video stream, $\mathcal{T}_2$ & $\mathcal{T}_3$ represent high resolution data coming from ECG and EEG machines. For each experiment we captured all messages received within a 10 second window and recorded their latencies. During following discussion, we use $\mathcal{S}_x$ to denote the subscriber to topic x and $\mathcal{P}_x$ as the publisher to x.

Topic	$minSep$	Max. Latency	Message Size (Bytes)	Bandwidth
$\mathcal{T}_1$	3ms	2ms	192192	512.512mbit/s
$\mathcal{T}_2$	3ms	3ms	96000	256.000mbit/s
$\mathcal{T}_3$	11ms	8ms	64000	46.545mbit/s
TOTAL:				815.057mbit/s

Table 5.2: Experimental Publish-Subcribe Set

Scenario 1: Comparison to Best-Effort

Here we compare the performance of the middleware in two network settings. In the first setting, we configure the Pica8 to behave like a normal L2/L3 switch (it uses a fair-queueing strategy to forward ethernet frames in this mode). We call this the 'best-effort' setting. In the second setting we place the network under control of the MIDAS using the strategy of Section 5.4.2. We ran three experiments where we observed the end-to-end latencies of messages published to each topic. In the best-effort setting $\mathcal{S}_{\mathcal{T}_3}$ still met its latency constraints. The same was not true for $\mathcal{S}_{\mathcal{T}_2}$ or $\mathcal{S}_{\mathcal{T}_1}$. Due to space, we report the data concerning $\mathcal{S}_{\mathcal{T}_2}$:

Figure 5.14 shows the latency of each message over the observation window for $\mathcal{S}_{\mathcal{T}_2}$. Figure 5.15 shows the same for the MIDAS managed setting. Each point on each graph represents the end-to-end latency of a single message sent to $\mathcal{T}_2$ and received by the subscriber. The x-axis is the moment (in milliseconds) that the message was transmitted. The y-axis is the latency of that message. Even accounting for jitters in the operating system and JVM the end-to-end deadline of $\mathcal{S}_{\mathcal{T}_2}$ is repeatedly violated on the best-effort system (observe the number of samples in the 5ms row of Figure 5.14). Additionally, $\mathcal{S}_{\mathcal{T}_2}$ never received 48%

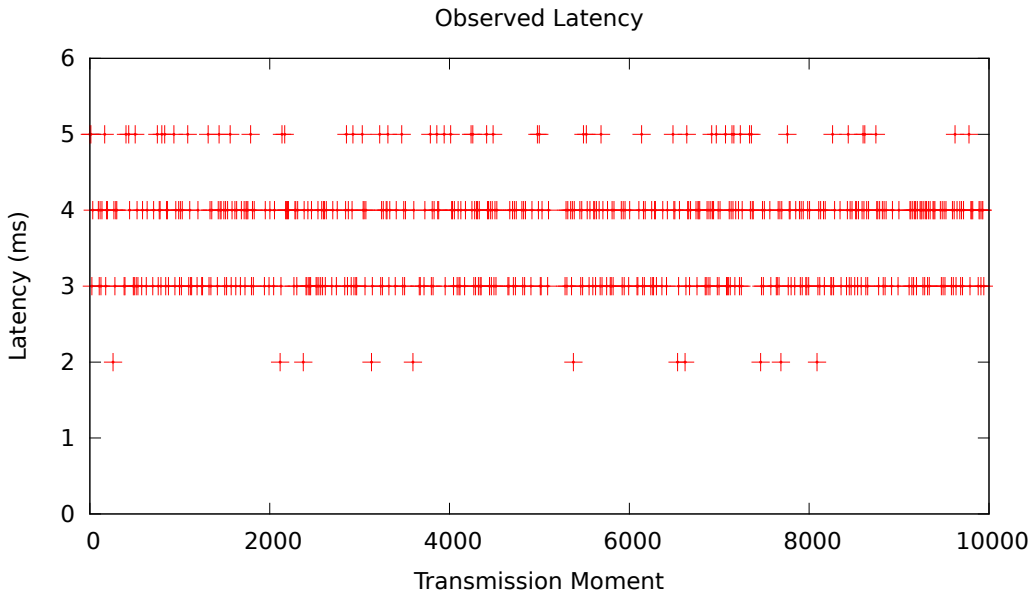


Figure 5.14: Best Effort

of the messages that were sent. This is because the messages are quite large and the egress queues can be overrun in the best effort setting. Any loss of a single ethernet frame will result in the loss of the whole message. All messages arrived in the MIDAS-managed setting. While it would be possible to reduce the message drop rate in the best effort setting by causing the senders to retransmit on failure, doing so would increase the effective latency of the message. Taking into account the 1 ms latency added by the JVM and TCP/IP stack, no messages violated the latency requirement when the MIDAS was managing the network configuration (All samples in Figure 5.15 are 4ms or less).

Scenario 2: Fault Containment

In this scenario we modify the publishers to $\mathcal{T}_1$ and $\mathcal{T}_2$ so they simulate babbling idiots (*i.e.,* set their $minSep$ to 0) and we record the latencies of messages flowing

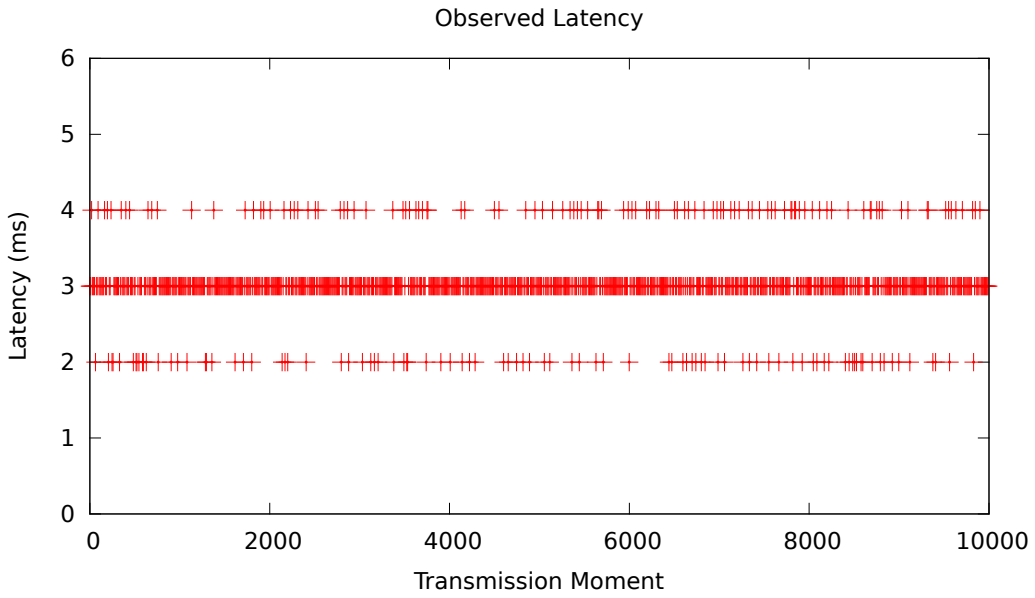


Figure 5.15: MIDAS

to $\mathcal{T}_3$. In this experiment the publishers were able to saturate a 1 gigabit per second Ethernet link each. We modified $\mathcal{P}_{\mathcal{T}_1}$ and $\mathcal{P}_{\mathcal{T}_2}$ because MIDAS will configure their respective flows with the highest priority which means they have the most opportunity to starve the other flows if they misbehave. This represents a worst case scenario for our approach. When run on the best effort network (*i.e.,* with no flow prioritization) $\mathcal{P}_{\mathcal{T}_1}$ and $\mathcal{P}_{\mathcal{T}_2}$ were able to starve enough of the network forwarding capacity from the flow associated with $\mathcal{P}_{\mathcal{T}_3}$ to cause *all* messages to be dropped. Figure 5.16 contains the observed latencies when MIDAS was managing the network. Again, each point in the graph represents a single message. The y-axis is latency of that message, and its x-value is the moment the message was transmitted. Under MIDAS, no messages were dropped and all messages arrived earlier than their required latency bounds, $8ms$.

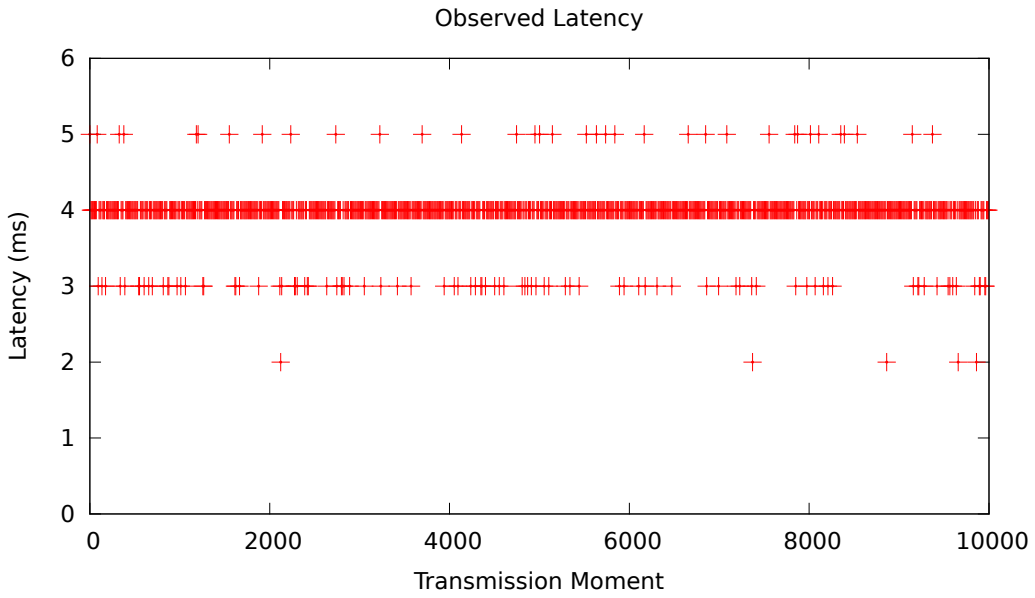


Figure 5.16: Latency bounds for $\mathcal{S}_{\mathcal{T}_3}$ when $\mathcal{P}_{\mathcal{T}_1}$ and $\mathcal{P}_{\mathcal{T}_2}$ are malfunctioning

5.5.2 Performance of the Determinizing Scheduler

We measure two aspects of our Determinizing Scheduler implementation. First, we measure the worst case execution time (WCET) of each type of Determinizing Scheduler microtask. The WCET time of the microtasks are important because it will affect the schedulability of applications: If the MDCF/MIDAS is running many applications at once, or those applications generate many I/O events, the Determinizing Scheduler will have to execute many microtasks in a millisecond. Second, we measure the level of timing determinism actually achieved by the implementation.

Both evaluations were performed on the same hardware running and operating system combination. The hardware consisted of a workstation equipped with an Intel Core i7-2600 and 8*GB* of RAM. The processor frequency was locked

at 3.5*Ghz*, hyperthreading was disabled and only one core was enabled. The operating system was Ubuntu Linux 14.04LTS running the *RT_PREEMPT* kernel patchset enabling fully preemptive priority-based real-time scheduling. The Java JVM used was IBM's WebSphere RT 2.0. The MDCF/MIDAS software architecture and implementation lets us map the Determinizing Scheduler to a `NoHeapRealtimeThread` [40] to ensure that the Java garbage collector will not interfere with the Determinizing Scheduler. All other application tasks were mapped to `RealtimeThread`s [40] with a priority dictated by the MDCF/MIDAS Resource Manager.

Microtask WCET

We randomly generated a set of ODSDL applications and extracted a set of microtasks associated with the applications. While the Determinizing Scheduler employees three types of microtasks (one each for the ODSDL sched, send and recv event) we actually tested 6 "categories" of task:

1. recv - data is received on a port.
2. sched(P) - periodic task is dispatched.
3. sched(A) - aperiodic task is dispatched.
4. send(1) - data is sent on a port with only one destination.
5. send(5) - data is sent on a port with 5 destinations.
6. send(10) - data is sent on a port with 10 destinations.

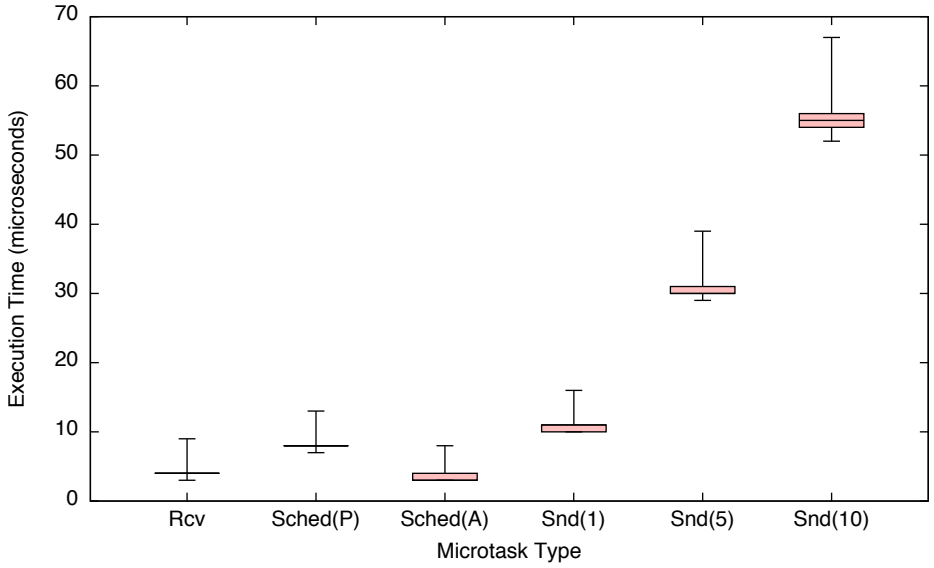


Figure 5.17: Microtask execution times in microseconds.

We ran each extracted microtask 500 times and recorded the duration of each execution. We used the `Clock.getTime()` methods exposed by the RTSJ runtime which enabled us to record each execution time with microsecond precision.

Figure 5.17 shows the measured WCET for each task category. The upper and lower bounds on execution time are indicated by the top and bottom whisker for each category. The mean is indicated by the middle whisker and the boxes (when present) show the range of the standard deviation.

On the test hardware our microtask implementation usually takes on the order of 10 microseconds to execute. The cheapest microtask is for recv events because the Determinizing Scheduler only needs to update a reference in the receiving module. Both types of sched tasks are also cheap. $\mathsf{sched}(P)$ is slightly (several microseconds) more expensive because it must calculate the next dispatch moment and schedule a new microtask while $\mathsf{sched}(A)$ only needs to dispatch a task.

The most expensive microtasks are typically for send events which makes sense because a hash map datastructure must be accessed to retrieve the set of application flows associated with the sending port. As expected, the cost of processing a send event scales linearly with the number of flows associated with the port.

Depending on the mix of microtasks needed, these results indicate that modern computing hardware can process 500 - 1000 microtasks in a given millisecond. Recall though that these experiments were run with only a single core: Access to multiple cores devoted to microtask processing potentially means even more microtasks could be processed in a given millisecond on modern hardware as long as datastructure locking is managed efficiently. Furthermore, these results reflect a Java implementation. While modern Java compilers and JVMs offer competitive performance relative to code generated from languages like C++ [129, 145], it is possible that additional software optimzations, compiler optimizations, or a different implementation language could shave microseconds from the costs of each microtask.

Measured Application Determinism

Recall that the Determinizing Scheduler is designed to ensure that application I/O events occur at the correct time, down to the millisecond, according to the ODSDL's semantics. Here, we evaluate our implementation's ability to make these I/O events happen at the right time. We randomly generated two ODSDL applications each with 10 modules/tasks. Both applications were deemed schedulable by the MDCF/MIDAS. One application had a task utilization $U = 0.75$, while the other had $U = 0.90$. Figure 5.18 gives a template for the test module. We used the UUniFast task-set generation method [37] to generate the period,

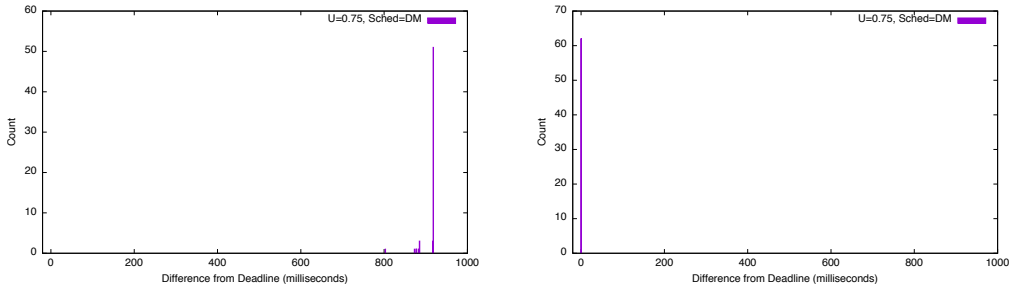
deadline, and cost parameters for each task in order to achieve the desired system utilization.

```
1  module DSTest {
2    net input "input"
3    net output "output"
4    vars {i, a, b, c}
5    task activated periodically [P]  delay [D]{
6        i = 0
7        a = input
8        while(i < [I]){
9          b = a + c
10         c = b + a
11         iter = iter + 1
12       }
13       send("output", c)
14     }
15   }
16 }
```

Figure 5.18: ODSL test module template. [I], [P] and [D] are set according to the test parameters.

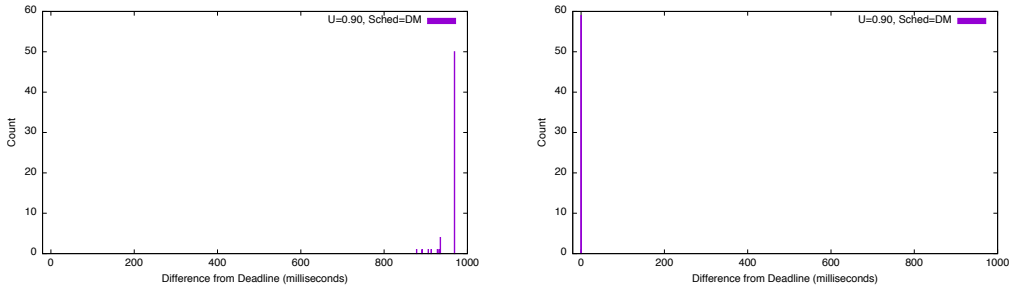
In order to have our test module tasks actually require the specied cost, we measured the cost for one of the task's loop's iterations. We validated that the task cost scaled linearly with an increase in the number of iterations (I). We then set the appropriate number of iterations for each module to achieve its specified execution cost.

Next, we randomly selected one task from each application to instrument. The instrumentation recorded the moment each time a job of the task finished relative to its deadline. We then ran each application for 1 minute using both the Determinizing Scheduler and a standard Deadline Monotonic scheduler and recorded



(a) DM **(b) DM + Determinizing Scheduler**

Figure 5.19: Task completion variability with 0.75 system utilization.



(a) DM **(b) DM + Determinizing Scheduler**

Figure 5.20: Task completion variability with 0.90 system utilization.

the relative finishing times of the test tasks.

Figure 5.19 shows the results for the $U = 0.75$ application. As we can see, when Deadline Monotonic scheduling is used the finish time of the task varied. While most of the finish times were around 900 milliseconds prior to its deadline, some jobs finished later. Under the Determinizing Scheduler, all jobs completed at exactely the deadline.

The story repeats with the higher utilization ($U = 0.90$) application (Figure 5.20). Again, with the deadline monotonic scheduler the task job complete (and

hence output moment) can vary. In fact, this task exhibited more variability than the task in the lower utilization application. Again, the Determinizing Scheduler makes all the job completions manifest at the specified deadline.

These results demonstrate that the Determinizing Scheduler apprach is able to achieve a very good level of determinism and predictability (down the the millisecond) even when using desktop class hardware, Linux, and Java as an implementation language.

5.6 Related Work

There are two important areas of related work. First, The role of a Medical Application Platform as a trusted base was inspired by the "separation kernel" concept that is becomming more commonplace in security and safety critical systems. Second, the idea that the platform could orchestrate computation and communications activities in order to achieve a sort of logical time execution was inspired by a number of other real-time systems research projects. We give a brief survey of the related work in each area and explain how that work relates to our contribution.

5.6.1 Separation Kernels

Separation kernels are like thin operating systems designed to *partition* or *separate* the different software functions they are hosting. The goal of the paritioning is to ensure that a software function in one partition cannot interfere with the software residing in another *unless* the systems integrator explicitly allows it. The concept of a separation kernel was first applied to security intensive systems. Later

the same concept (but with different features) was suggested for safety-critical systems.

The Security Context

The first mention of a separation kernel was in a 1981 paper by John Rushby [162]. The separation kernel described by Rushby was intended to reduce the *cost* of providing the assurance that a composite software system would not inadvertantly leak data from high a security channel to a low security channel.

The composite software system consists of many individual software functions. Some of these functions would be designed to only deal with low security data, while other functions would process high security data. Clearly the software functions that are destined to handle high security data should be exposed to as much verification as is possible to assure that they will not mismange (*i.e.,* leak) that data.

But what about the low security software functions? If the composite software system was constructed by linking the individual functions into a single program execeutable then composite system as a whole would also need to be closely scrutinezed to ensure that low security functions do not somehow interfere with the high security functions or otherwise gain access to classified data. Verification of the composite system can be exponentially more complex (because one has to worry about all the possible interactions between the individual components) which results in a much higher verification cost.

The first core idea of the separation kernel is to mitigate the whole system verification cost by running each software function as a process isolated inside its own partition. The job of the kernel, then, is to ensure that functions operating in one

partition cannot interfere or interact with functions inside another (unless explicitly allowed by the systems integrator). But how is this different than just running each software function inside its own process on a modern operating system with memory protection?

The second core idea of the separation kernel is *simplicity*. A modern operating system itself is very complex. If it exhibits a security vulnerability or an implementation defect one process may be able to interfere with another in unexpected ways. A high level of assurance for individual software functions is useless if the underlying operating system cannot be verified to the same level of assurance. Unfortunately a full fledged operating system can be more complex than the individual software functions that it hosts. Therefore a good separation kernel should be simple as to reduce the cost of its own verification and validation. Rushby was a proponent of a separation kernel whose design and implementation could be fully and formally verified via formal mathematical proof. In theory, this would bring a high level of assurance to the implementation and design of the kernel. System integrators could then focus their verification and validation efforts on the high security software functions under the assumption that the separation kernel would ensure that the low security software functions would not acess a high security function or channel.

Since Rushby'y 1981 paper there has been many ongoing efforts to create a formally verified separation kernel. [157] proposes a technique for the formal verification of a separation kernel. In 2006, Heitmeyer *et al.* [78] were able to produce a formally verified separation kernel and provide a high level of assurance with respect to the requirements laid out in the Common Criteria. More recently, Klein *et al.* [111] were able to produce the formally verified separation kernel

seL4.

The Safety Context

Separation kernels for security work because they can enforce a useful *negative* property: High security data cannot inadvertently flow from a high-sec partition to a low-sec partition. Separation kernels are able to enforce this property because they exist at a (low) level in the system software stack where they can mediate the interactions between individual software functions and between software functions and the underlying hardware.

While it is not exactely the same, there are significant parallels between the challenges integrators of safety-critical systems face and those of security-critical systems in terms of the cost of providing high assurance. Like security-critical systems, modern safety critical systems can be composed of many software functions. In any given system, each software function may have a different level of criticality (*i.e.,* some software functions may not have a safety-critical role while other functions would). Thus if integrators want to combine a large number of functions with a single computing platform they must either ensure that 1) the composite system is verified to the highest level of assurance as a whole or 2) verify each function indpendently and then use some lower-level mechanism to ensure that the functions cannot interfere with each other in unexpected ways. Option 2 requires some form of separation.

John Rushby recognized the utility of a separation kernel for safety critical systems in 1986 [158]. In that paper Rushby makes a first attempt at formalizing what it means to enforce a negative property but does not discuss actual enforcement mechanisms. In the 1990's there were efforts within the aviation industry

to standardize the separation services that a kernel could provide and how those interfaces would present to sofware functions as APIs (see: [54]). Those efforts culiminated in what we now know as Integrated Modular Avionics, or IMA [160]. We must note that IMA is a concept, not a specific set of technologies, even though there are a variety of platforms and technologies on the market that offer "IMA" capabilities.

Unlike security separation kernels, IMA platforms have to do more than just ensure one software function cannot access (or corrupt) another's data. Because avionics have real-time requirements, the system integrator must ensure that each real-time function has access to appropriate computing resources *when it is needed*. Computing resources can be processor time, the network, sensors, and actuators.

To ensure that functions have access to processor time system IMA integrators will typically use a RTOS that complies with the Application Executive (APEX) ARINC 653 [17] standard. In addition to running each software function in its own protected memory partition with its own memory region (space partitioning) these RTOS ensure each function access to processor time by employing preemptive cyclic scheduling. At integration time, the execution schedule for each function is chosen. Each function is allowed to process its tasks during its scheduled execution slot. The RTOS will forcibly preempt any task that runs beyonds its function's execution slot. This ensures that a malfunctioning task of one function cannot prevent a different function's access to allocated processor time.

There are a variety of network technologies designed to be used as the communications substrate of IMA systems such as Avionic Full Duplex Ethernet [18, 4] the Time Triggered Protocol [114], or the ARINC 629 data-bus. Like the APEX compatible RTOS, all these networking technologies must be configured (*i.e.,*

choose the transmission schedule and/or priorities) during system integration to ensure that each avionics function has access to networking resources at the right time.

Integrated Modular Avionics have been successfully used facilitate systems integration for a number of high profile aircraft including the Airbus A380 [94], as well as the Boeing 787, C-130 and KC-767 Tanker [180].

5.6.2 Execution Strategies for Real-Time Determinism

In [81, 79] Henzinger *et al.* proposed the *embedded-machine* or e-machine. The e-machine implements an execution strategy for real-time programs comprised of a graph of periodic real-time tasks generated by the compilation of Giotto programs. Like our determinzing scheduler, the e-machine separates task computation from inter-task computation: The e-machine scheduler dispatches a task an keeps its outputs until the task deadline at which point it moves the outputs to the next task's inputs and dispatches the next task. While [81] only supported periodic tasks, there have been a number of extensions to support aperiodic and event driven models [67]. Unlike the determinizing scheduler the various incarnations of the e-machine are not designed for use in a distributed environment.

In [189], Zou *et al.* describe execution strategies for PTIDES [58, 59] programs based on Discrete Event Simulation. While the strategies described in [189] can work in a distributed setting (unlike the e-machine) assesing their schedulability is complicated relative to the Determinizing Scheduler. While the DS transforms ODSDL into a set of aperiodic tasks where traditional scheduling techniques can be used, [189] requires static analysis to determine causal relationships between actors. Furthermore, current techniques to asses the schedulability for

PTIDES programms utilizing the complete PTIDES programming model depends on a reduction to reachability in timed automata and make not be appropriate for online schedulability analysis.

Chapter 6

Case Studies

6.1 Introduction

In this chapter we take two of the motivating examples from Chapter 2 and use them as case-studies. The goal is to tie the work in the preceeding chapters together. We will use the case studies to illustrate how the regulatory framework (Chapter 2), On-Demand Systems Description Language (ODSDL, Chapter 4) and Medical Application Platform (MAP, Chapter 5) work together to provide safety assurance for on-demand systems via a safety argument. Additionally, these case studies will give us the opportunity to evaluate the performance ramifications of a *modal refinement* (Section 3.2) based device/application compatibility check using realistic device specifications.

This chapter is organized as follows:

First we will describe some key assumptions about the regulatory framework and on-demand ecosystem (Section 6.2). These assumptions will be critical to understand why certain claims in each use-case's safety arguments are justified. Un-

fortunately, as of this writing, there is no real regulatory framework as described in Chapter 2. While our regulatory framework and ecosystem will be imaginary, we will relate the key assumptions to realistic and/or concrete examples.

Next we will describe the case-studies (Sections 6.3 & 6.4). For each case-study we will define the safety requirements and discuss scenario specific assumptions. We will then use the ODSDL to design an on-demand system to address the clinical scenario and explain how the different features of the ODSDL help us in our task. We will then construct an informal safety argument for the system. The safety argument will illustrate how the regulatory framework, properties of the ODSDL, and MAP combine to provide safety assurance *for that particular* system specification. Because each safety argument is by its nature informal, there is the possibility of *assurance deficits* (*i.e.,* gaps that may make the argument unconvincing). For each case study we will endeavor to identify important deficits and discuss how the deficits may (or may not) be remedied.

Finally, we will benchmark our modal-refinement based compatibility checking procedure using the device specifications from the case-studies (Section 6.5). The goal of the benchmark is to ascertain how long checking modal refinement takes for realistic specifications and to develop a preliminary understanding of its practicality.

6.2 Ecosystem Assumptions

Here we make a number of assumptions about the ecosystem that apply to both use-cases. These assumptions add detail to the basic processes outlined first in Chapter 2.

ODSDL is Standard

We assume that the ODSDL is the standard used by application developers to program applications and express requirements on device behavior, and it is used by device manufacturers to specify how their device's behave. Recall that the ODSDL lets developers and manufacturers model device/environment interactions with specific physical action types. These action types are just strings and their *physical meaning* is not defined in the core ODSDL. Instead, it is the job of the Ecosystem standards consortium to define a taxonomy of phsyical action types, define the physical meaning of the action, and define the compliance criteria for each action a device purports to generate or react to. We will give examples in the case study specific ecosystem assumption sections later.

Furthermore, we assume that there is a certified Development and Verification Environment (DVE) available to the ecosystem stakeholders (Figure 6.1). The DVE enables an ecosystem specific form of Model Driven Development (MDD). The DVE lets application developers program/specify their applications using ODSDL. Then the developer can use the DVE to both automatically generate an application bundle certified MAPs can execute as well as generate TPMS models of the system that are suitable for model-checking, simulation, or other forms of analysis.

Lastly, we assume that the notion of device/application compatibility used by the ecosystem is that of *weak modal-refinement* between TPMSs (Section 3.2). When a platform checks to see if the device chosen by the user is compatible with an application is desugars the ODSDL specifications into TPMS specifications and then checks whether the device TPMS weakly refines the requirements TPMS.

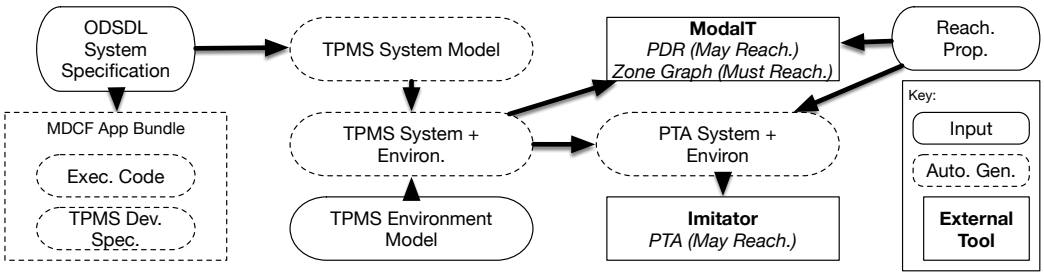


Figure 6.1: Development & verification workflow in the Ecosystem's ODSDL Development & Verification Environment

Device Compliance

We assume that device manufacturers must follow some sort of MDD process to design and implement their devices. In particular we assume that the device manufacturers must design the "digital" (*i.e.,* part the controls the sensors and actuators) portion of their device by creating a timed-automata model and then autogenerating an implementation with a tool like TIMES [9]. After the physical implementation of the digital portion has been created, its timing characteristics must be tested and verified to conform to the initial timed automata specification following a process such as described in [97]. The portion of the device excluded from the digital portion (*i.e.,* the sensor and actuators) must be verified for compliance according to specific criteria established for the sensor or actuator type. We will discuss this more in detail in the ecosystem assumptions specific to each use-case.

Furthermore we assume that each certified device properly implements the ecopshere's standard interoperability protocols and runs a local version of the determinizing scheduler (Section 5.3): When connected, the device's clock is always synchronized to the connected platform's clock (within some ϵ), and when

the device receives a timestamped message carrying and ODSDL event, it won't manifest that event to the device's control logic until the timestamp.

Platform Compliance

We assume that each certified platform properly executes ODSDL programs. This includes properly implementing the ODSDL's logical execution time semantics and correctly implementing the procedure used to check modal refinement. Additionally, we assume that each platform is able to provide guarantees of separation and isolation between concurrently running applications (*e.g.,* [162, 7, 158, 160, 159]).

Furthermore, we assume that the correctness of the platform has been rigorously verified, following the *types* of guidance for highly critical systems laid out in, for example, the DO-178/254 guidance standards for avionics [98, 83, 146] and the Common Criteria [154, 140]. [1]. We also assume that, where possible, the functional correctness of the platform software has been formally verified, *e.g.,* as in [111].

6.3 Laser-Ventilator Interlock

6.3.1 Scenario Specific Assumptions

Before we proceed with a description of our Laser/Ventilator interlock application we need to make some assumptions. The first assumptions will concern the

[1] A discussion of what particular aspects of the referenced guidance should be applied is beyond the scope of this dissertation. We merely assume that the platform is verified as extensively as any other existing safety critical software system

clinical environment and the intended use of the application. Based on the scenario assumptions we will devise the safety-property the application must satisfy. Next, we will state some assumptions about the ecosystem compliance criteria for laser-scalpels and ventilators. These assumptions about device compliance will help the reader interpret the meaning of the ODSDL device requirements specifications used in the application, and they will be exploited in the safety argument itself.

Clinical Environment & Intended Use

We assume that our application will only be designed to prevent surgical fires and won't try to protect the patient from O_2 desaturation. We assume that the anesthesiologist and the surgeon will still communicate during the procedure and that the anesthesiologist will inform the surgeon if s/he should take a break from cutting in order to let the ventilator run and increase the patient's SpO_2 level.

We also assume that the concentrated O_2 in the patient's airway dissipates quickly after the ventilator has been stopped or paused. While the dissipating is not instantaneous, we assume that is dissipates down to safe levels in under a fraction of a second (though this may differ between patients, ventilator, and gas mixture settings).

Laser/Ventilator Device Compliance

Table 6.1 contains a listing of the ODSDL physical type names we assume have been standardized in our imaginary ecosystem. Each type is designated with a name, and can represent a physical signal that is either just an event (*i.e.,* "it happened") or an event that carries a value (*i.e.,* "it changed its output to value

Physical Type	Value/Event	Description
LaserOutputWatts	Value $[0 \ldots 100]$	Laser output energy in watts.
ReqPauseEvent	Event	When the pause request button is pressed.
ToggleOnEvent	Event	Pressure is applied to the laser activation button.
ToggleOffEvent	Event	Pressure is removed from the laser activation button.
VentFlowRateLSec	Value $[0 \ldots 50]$	Gas flow rate in litres/second.
EmergencyBtn	Event	When the emergency button is pressed.
RateInputPanel	Value $[0 \ldots 50]$	Req. flow rate received from device panel.

Table 6.1: Physical types used by devices in the Laser/Ventilator interlock application.

v").

We assume that if a laser scalpel has been certified its actual laser output will match its specified output (LaserOutputWatts) within 2% if the laser is active. We assume that if the laser specifes an output of 0 at discrete millisecond t (*i.e.,* it is supposed to be turned off) it will physically disconnect the laser emitter from the power circuit before $t + 1$. This design requirement is verified by the certification authority via manual inspection of each submitted laser-scalpel design. ReqPauseEvent, ToggleOnEvent, and ToggleOffEvent are all events that represent the physical signal generated when the surgeon presses one of two buttons on the scalpel. ReqPauseEvent fires when the surgeon clicks a button for requesting ventilator pause. ToggleOnEvent and ToggleOffEvent let use model a toggle button: ToggleOnEvent fires when a button is depressed, ToggleOffEvent fires when the surgeon lets off the button.

Likewise, we assume that if the ventilator has been certified its actual output

will match its specified output (VentFlowRateLSec) within some error. However, once the ventilator indicates that it should be stopped by discrete millisecond t, the ventilator mechanism (usually a piston or some other pump) must have its movement fully arrested by time $t + 1$. The RateInputPanel even represents when the operator programs and commits a new ventilator flow rate (*e.g.,* types in a new rate and hits enter.) We must note that real ventilators expose many more parameters than just "flow rate". Typically, you can specify respiratory rate, FiO_2 (amount of O_2 delivered), tidal volume, pressure and many others [186]. In this case study we focus on flow rate as the primary ventilation parameter to simplify the presentation. EmergencyBtn is an event that represents when the operator presses a button to activate an "emergency mode". Emergency mode is intended to disable the pause feature of the ventilator if one exists.

6.3.2 Application Design

Our safety interlock (see Figure 6.2) consists of a laser-scalpel (Figure 6.3), ventilator (Figure 6.4), and two application software modules (Figure 6.5). The environment consists of an operator and patient. At a high level, the interlock works as follows: when the surgeon wishes to use the laser scalpel s/he will depress a "request" button on the scalpel. The scalpel will send an event to the application requesting for it to be enabled. When the application gets the activation request it will first ask the ventilator to pause. After the ventilator pauses, the ventilator will acknowledge the pause request to the application which will in turn send an "enable" message to the scalpel. The laser scalpel will remain enabled (*i.e.,* can be turned on) until just before the ventilator resumes gas flow.

Whether or not this system satsifies its safety requirement (*i.e.,* that the ven-

tilator is never on when the laser is) depends on the relative timing of the mode-changes in the devices and the events received and generated by the application. We will now discuss the specification of each component and how the specified *must* behavior of each component helps satisfy the system safety requirement, while any specified *may* behavior will not cause a violation.

The application requires that the laser-scalpel (Figure 6.3) start in a "disabled" state (*i.e.,* it will not react to the surgeon toggling the 'on' button). The scalpel will only become receptive to the 'on' button after it has received an signal on its `allow` network input port from the application and moved into the `READY` control location. Once the scalpel is enabled the specification allows for some timing variability in laser activation time (the parametric contraints on the invariant of `ACTIVATE` and the guard leaving `ACTIVATE`). This variability is there to account for possible power systems design differences between different scalpels: Some designs may a longer take time to energize its circuits until a laser beam is formed. Furthermore, some designs may be more or less predictable in terms of their energizing time. This variability is not extended to the deactivation time: it is required that the laser beam shutoff in the same millisecond as when the `btn_off` event occured. One critical aspect of laser specification is the amount of time it remains enabled: The laser-scalpel will disable itself (and shutoff the laser if active) once 4.979 seconds have elapsed since it was enabled. Why 4.979 seconds? As we will see our ventilator specification has the ventilator pause for 5 seconds and then resume, but due to variability in the ventilatory, network latency, and application processing delays the ventilator may have already been paused for up to 0.020 seconds before the laser scalpel gets enabled. Disabling the scalpel 1 millisecond early ensures that there is no overlap of time when both the laser and

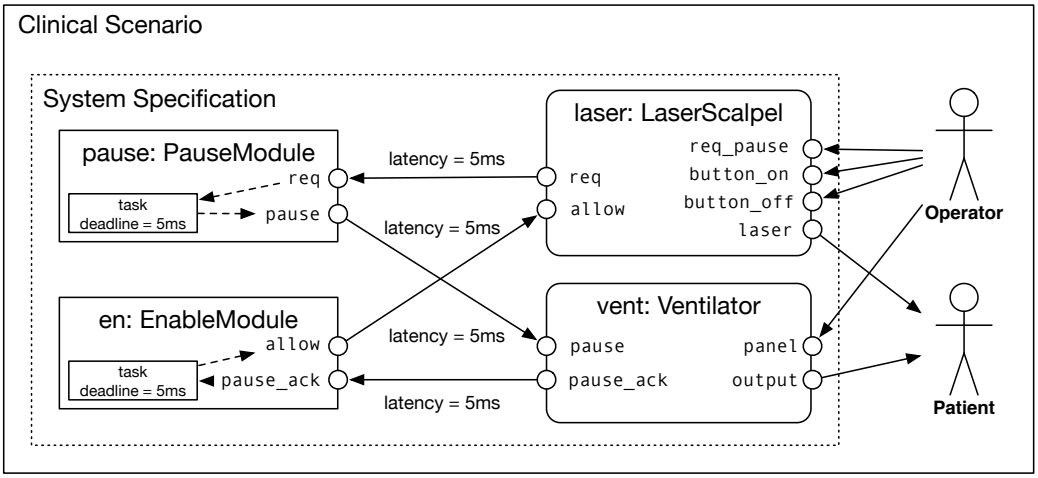


Figure 6.2: Laser/ventilator clinical scenario and safety interlock system specification.

ventilator are on.

In addition to the timing behavior just described, the ventilator specification (Figure 6.4) also admits some *functional* variability. First, it allows the operator to change its programmed flow rate via the device's panel (the *may* edges labeled `panel ? rate`). It also admits ventilators that allow the operator to put the machine into an "emergency" state where it will ignore all future pause requests. The ventilator spec also allows some timing variability in pausing (the parametric contraints in `PAUSING`) and sending the pause acknowledgement message (the parametric constraints in `PAUSE1`). The variability in `PAUSING` accounts for mechanical differences between different ventilators (*e.g.,* some pump mechanisms may have inertia to overcome before they fully stop) while the variability in `PAUSE1` can account for internal processing differences.

The application software modules themselves are quite simple (Figure 6.5). The task of each module activates when the module receives and input on the

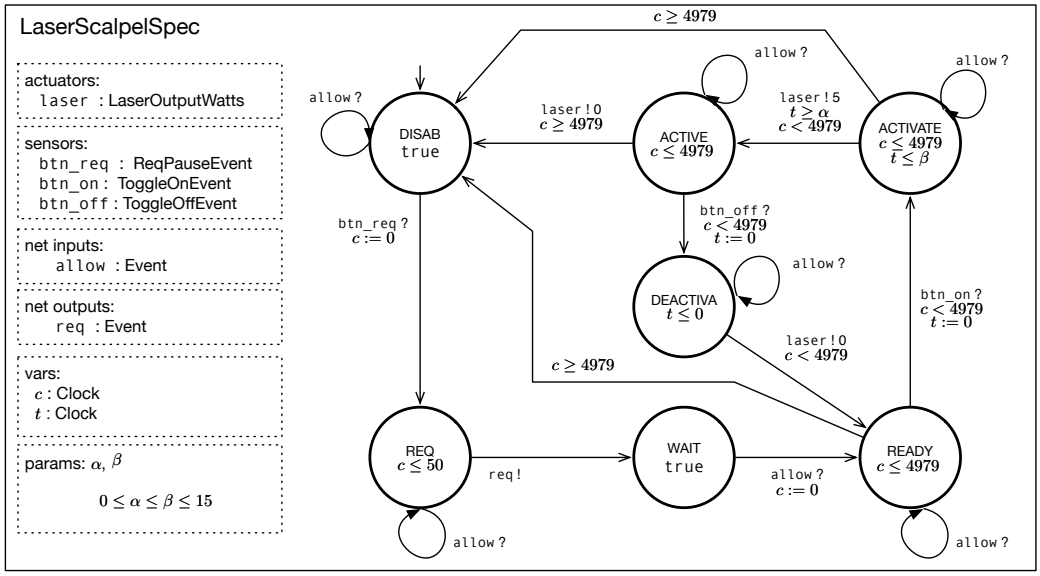


Figure 6.3: Requirements for laser behavior

input port and then sends an output 5 milliseconds later.

6.3.3 Safety Argument

Our safety argument for the Laser / Ventilator interlock application (Figure 6.6) is an instatiation of the argument pattern from Section 2.4. We will describe each node of the argument. The top-level goal of the argument (**G: NoMutualActivation**) is to argue that there is adequate assurance that *all* possible instantiations of the the aser / Ventilator interlock application permitted by the ecosystem ensure that the laser is never active when the ventilator flowing O_2 to the patient.

The goal **G: NoMutualActivation** is discharged using the *platform argument strategy* (**S: PlatArg**Section 2.4) which requires that the property in question is verified using a model-based reasoning step (**G: ModelSat**), and then that

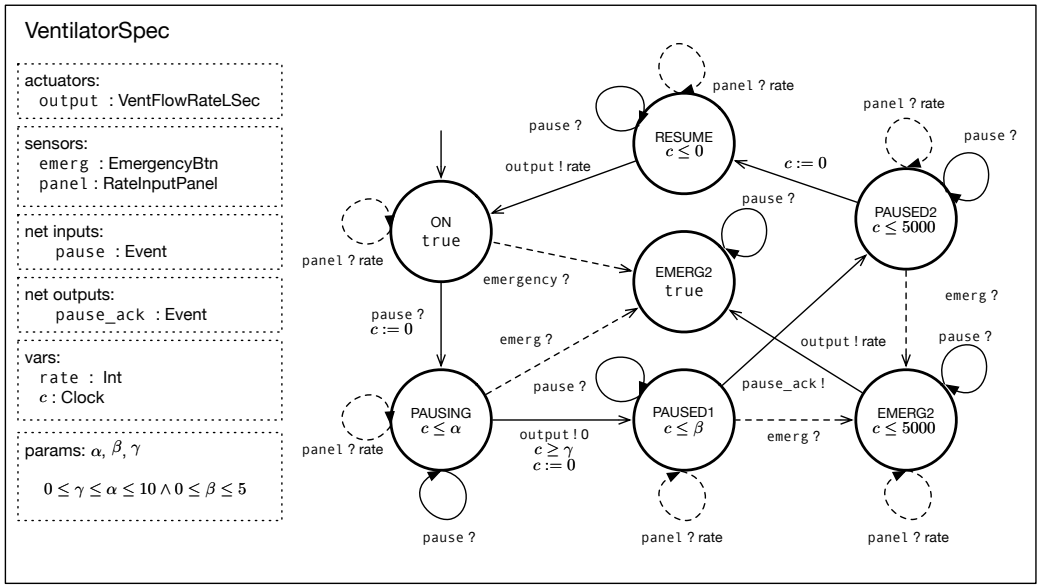


Figure 6.4: Requirements for ventilator behavior

all the models used have their adequacy justified given the safety property, system, and environment in question. Note that we take advantage of our imagined ecosystem's reliance on the ODSDL and the certified DVE toolchain: The **S: PlatArg** strategy is used in the context of models are automatically derived from the ODSDL application specification (except the environment model).

The model-based reasoning step is captured by the goal **G: ModelsSat** and its sub-argument. We took the application TPMS model autogenerated by the DVE and composed it with a handcrafted TPMS model of the environment to create a closed-model of the system. The environment model (`Env`) had two state variables that represented the current laser (`Env.laser_input`) and ventilator (`Env.vent_input`) input levels. The environment model also non-deterministically generated `btn_req`, `btn_on` and `btn_off` events to the laser-scalpel.

We used model-checking to verify two important properties of the closed sys-

```
1 module PauseModule {
2   net input "req" datatype Event
        minsep 100
3   net output "pause" datatype Event
        minsep 100
4   vars { }
5   task activated by port "req"
        delay 5{
6     send("pause")
7   }
8 }
```

(a) Pause module.

```
1 module EnableModule {
2   net input "pause_ack" datatype
        Event  minsep 100
3   net output "allow" datatype Event
        minsep 100
4   vars { }
5   task activated by port "pause_ack"
        delay 5{
6     send("allow")
7   }
8 }
```

(b) Enable module

Figure 6.5: Interlock software modules.

tem model:

- **P1**: The state `Env.laser_input` $> 0 \ \wedge \ $ `Env.vent_input` > 0 is not reachable.
- **P2**: The model does not have timelock.

P1 is simply a formalization of the mutual exclusion property from **G: NoMutualActivation**. We checked **P2** because the ODSDL device specification sublanguage does not prevent the application designer from committing any modeling errors that result in deadlocks or timelocks. If the model had a timelock then the model **P1** could be vacuously true. We verified **P1** & **P2** using the PDR model-checking engine in our tool *ModalT*. ModalT's evidence includes a log file of the model checking process as well as an *inductive invariant* over the model's transition relation that implies that the bad states cannot be reached. The inductive

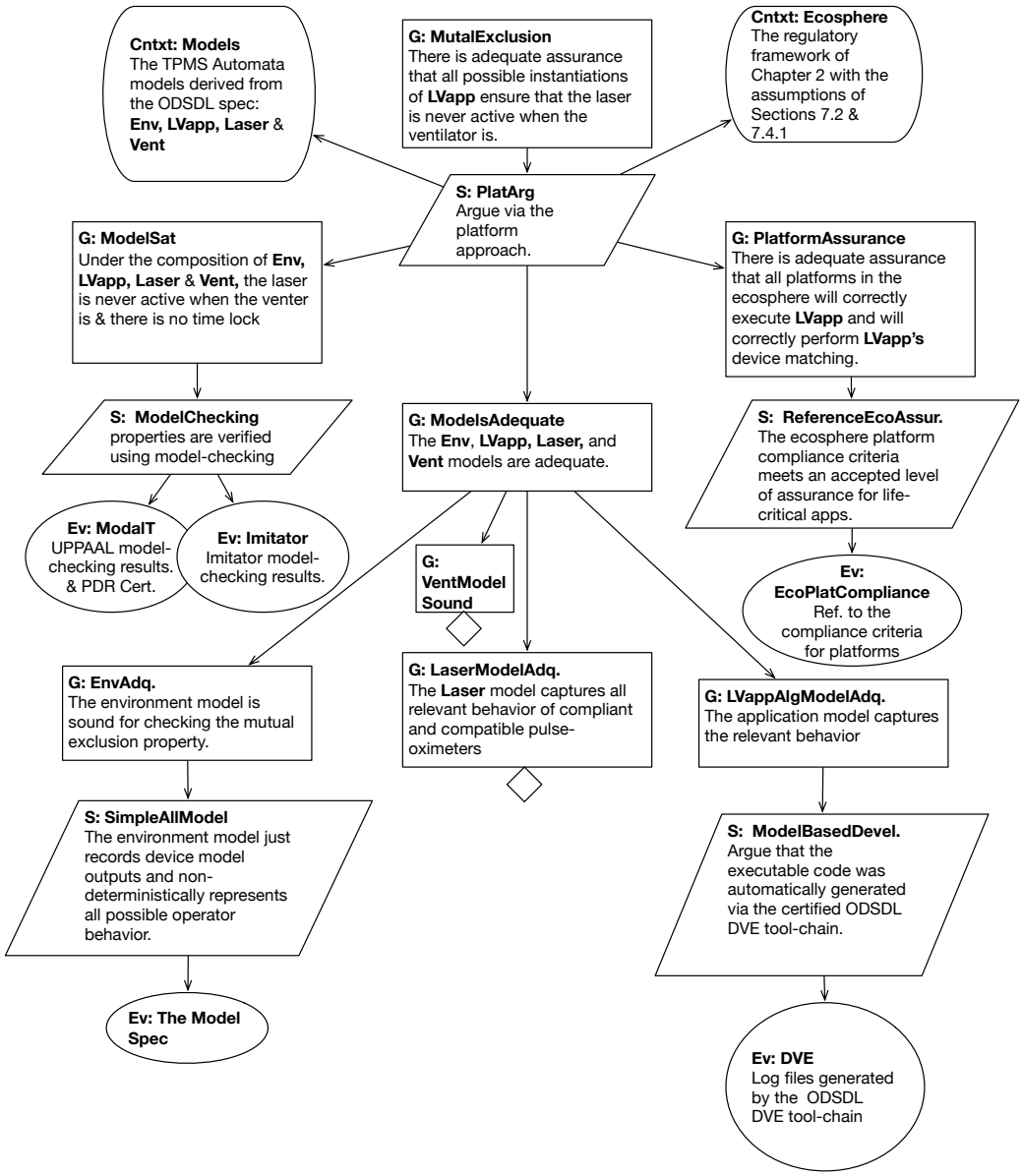


Figure 6.6: Assurance case fragment for the Laser / Ventilator Interlock application.

invariant serves as a certificate and can be verified with an SMT solver that supports a theory of linear arithmetic over the reals. We also used the DVE to export the TPMS model to the input-language supported by the *Imitator* model-checking tool [10, 11, 12] which can check reachability of Paramteric Timed Automata (PTAs) using parametric extensions of zone-graph methods [14, 13].

We argue the adequacy of the model-based reasoning in the sub-argument rooted at goal **G: ModelsAdequate**. Our argument pattern requires us to justify each model that we used. We justify our environment models by arguing that it only needs to generate all possible operator inputs and correctly record the current device mode. Since the model is simple, this can be verified by inspecting the model itself (**Ev: TheModelSpec**).

Justifying the adequacy of the device models (**G: LaserModelAdequate, G: VentModelAdequate**) is more interesting as that argument depends on the ecosystem device certification criteria. Indeed, this part of the overall argument is critical because it requires the application developer to show the connection between the modeled universe and the real physical universe. We elaborate on the argument for the laser model (**G: LaserModelAdequate**, Figure 6.7). **G: LaserModelAdequate** has two parts. The first part (**G: CompatModelAdq.**) is to show that the models used in the model-based reasoning captures all the possible behaviors *modeled* by the specification of any compatible device. Since our imaginary ecosystem uses modal refinement between TPMSs as its notion of compatibility, and our models are TPMSs, this sub-goal is discharged with the property preservation and compositional reasoning theorems of TPMS (Theorems 3.2.3, 3.2.9 and 3.2.13).

The second part (**G: ImpModelAdq.**) connects the behavioral spcification

the device presents with its actual possible physical behaviors via the Ecopshere properties and certification criteria. For this system it is critical that when the laser model says the laser is off it is not actually emitting any laser energy. Recall, though, that the compliance criteria we have assumed for lasers allows for lasers that don't output exacetly what they specify. There is some error allowed for physical differences and/or defects in the laser emitting medium and supporting power infrastructure. Also recall that the compliance criteria requires that when a laser says it is off, its laser emitting sub-system must have its power source physically disconnected and that to achieve certification, the physical disconnection must be externally verified. Our argument for **G: ImpModelAdq.** concludes by arguing that laser output is irrelevent when the laser is on, but that the ecosystem certification criteria and processes gives a sufficient amount of assurance that the laser will actually be off when it needs to be.

Potential Assurance Deficits

Here we will discuss two classes of deficits the reviewer might identify. The first class concerns the specified safety property. The reviewer might *judge* that the safety properties argued for in the safety case are incomplete or don't reflect the right tradeoff between risks and benefits.

One potential problem with the top-level safety requirement is that it ignores the potential for O_2 desaturation of the patient. If the surgeon keeps requesting a ventilator pause, and if the anesthesiologist is inattentive, they can keep the ventilator stopped indefinitely, eventually causing the SpO_2 levels of the patient to dangerous levels. This deficit can be dealt with in a number of ways. It may be that the safety case reviewer with expert medical knowledge judges the risk

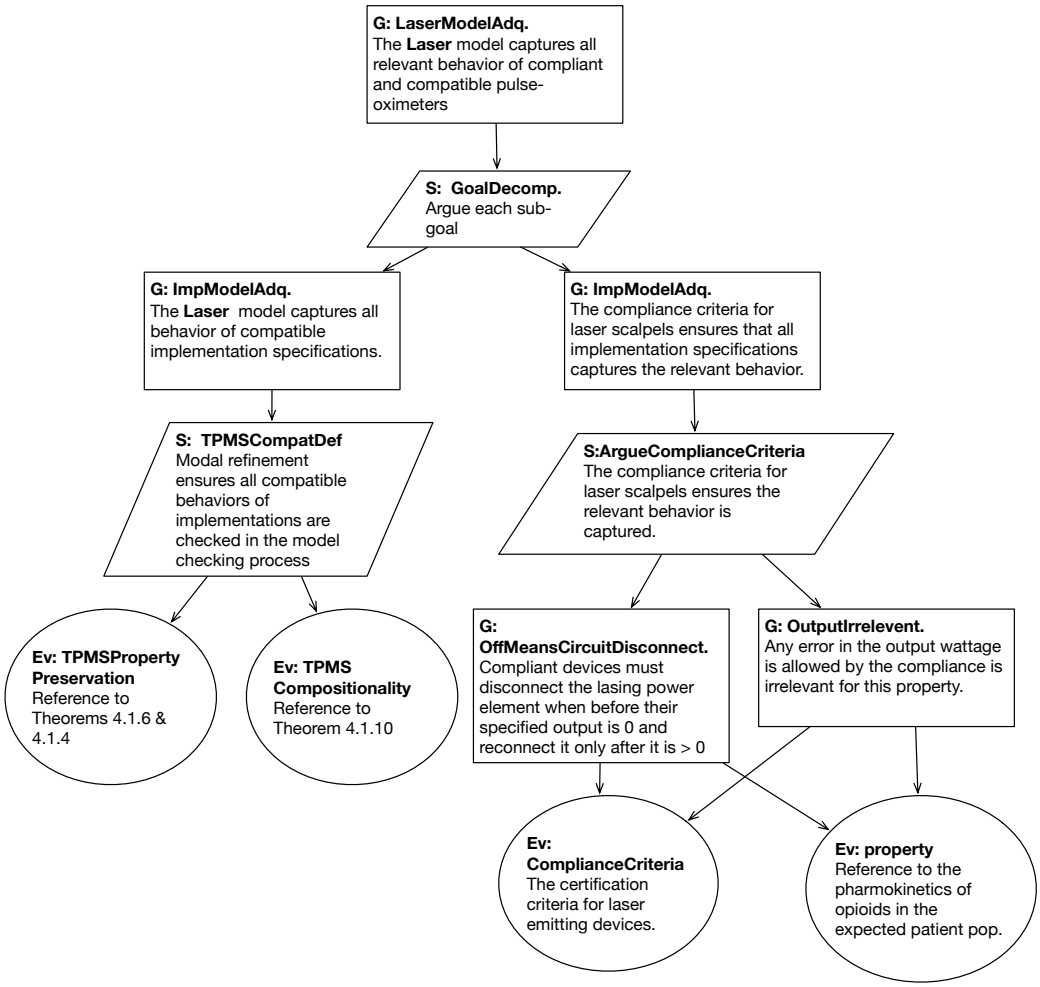


Figure 6.7: Arguement fragment for the adequacy of the laser model.

of surgical fire large enough relative to the risk of desaturation that they allow the deficit (*i.e.,* the benefit of preventing fire outweighs the risk of accidental desaturation). Or, they may decide that the risk of desaturation is significant. If they judge the desaturation risk significant enough, they can deny the certification of the application and ask the application developer to redesign and satisfy extra safety requirements (*e.g.,*, that the ventilator is never paused when the patient's SpO_2 is below some expert defined threshold).

Another problem with our safety requirement is that it only targets the device mode, rather than the physical state of the airway. Indeed, even if the ventilator has been stopped, concentrated O_2 may remain in the airway and contribute to the fire hazard. The expert review may decide that the gases in the airway will tend to dissipate fast enough and make the risk insignificant enough. On the other hand, like with the previous example, they may want the developer to redesign their application to ensure that the laser is not on when gases in the airway are present.

The second class concerns the substance of the safety case itself: they might not find the argument or its evidence compelling enough given the risks/benefits to the clinical scenario. For example, the reviewer may judge that the compliance criteria for either laser for verifying the "0" output is inadequate for this application. For example, the compliance criteria might not make any mention of the laser shutoff functions *reliability*, or if it does, it does not require a high-level of reliability.

In addition to potential deficits linking the device's modeled behavior to its real behavior, the expert review might have reason to believe that the tools (*e.g.,* ModalT, Imitator, or the DVE itself) used to design the application and perform

the model-based reasoning are not adequately trustworthy. For example, they might not believe the tools are free from defects that can significantly impact the results.

6.4 Closed-Loop Management of Patient Controlled Analgesia

6.4.1 Scenario Specific Assumptions

Like with the Laser/Ventilator interlock application we need to state some assumptions before we can proceed. As before, the first assumptions will concern the clinical environment and the intended use of the application. Based on the scenario assumptions we will devise the safety-property the application must satisfy. Next, we will state some assumptions about the ecosystem compliance criteria for pulse-oximeters and PCA pumps.

Clinical Environment & Intended Use

We assume that the PCA management application is designed to be used with patients convalescing in an ICU, or a Medical Surgical Stepdown Unit (MedSurg Unit). We assume, based on risk/benefit analysis and knowledge of physiology, that medical professionals are in agreement that an SpO_2 level of 75% is an acceptable threshold between safe and dangerous respiratory function (*i.e.,* that an $SpO_2 < 75$ is dangerous but ≥ 75 is safe).

We assume then, that our application's safety goal is to keep the patients $SpO_2 \geq 75$ at all times. We also assume that a simple linear model (the one

Physical Type	Value/Event	Description
PercentSpO2	Value $[0 \ldots 100]$	Blood oxygen saturation percentage (SpO_2)
InfusionRateMlMin	Value $[0 \ldots 1000]$	Infusion rate ml/Min.
BolusReqEvent	Event	Bolus requested by the patient.
GenAlarm	Event	A non-specific alarm event.

Table 6.2: Physical types used by devices in the PCA management application.

of Figure 6.13) of patient pharmokinetics is sufficient, *i.e.,*, it represents a worst-case scenario of a patient who is very susceptible to opiod induced respiratory distress.

PCA Pump/Pulse Oximeter Compliance

Table 6.2 contains a listing of the ODSDL physical type names we assume have been standardized in our imaginary ecosystem for pulse-oximeters and PCA pumps.

PercentSpO2 represents an event where the pulse-oximeter samples the current SpO_2 value from the environment. We assume that the ecosystem compliance criteria allows from some small error (*e.g.,* 2-3%) from an reference value generated by an approved pulse-oximeter calibration tool [90, 89]. The calibration tool simulates the red and infra-red light emission/reflection characteristics of human tissue for a given SpO_2 value.

InfusionRateMlMin represents an event when the infusion rate of the pump changes to a different value. We assume that all compliant pumps have been tested for flow accurancy at each programmable value of flow. Due to the physical

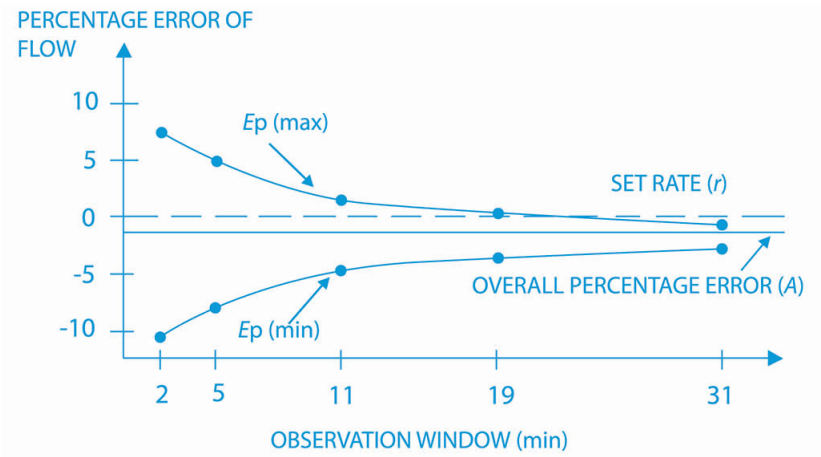


Figure 6.8: Example trumpet curve. Taken from [171].

characteristics of the pump mechanism, it is not practically possible to have a pump deliver precisely the flow rate it species. Typically, the accuracy of infusion pumps are measured using "Trumpet curves" [148]. Trumpet curves (see example in Figure 6.8) visualize the the maximum flow-rate error observed over a specific window of time. They are called trumpet curves because the relative error over a short window is much larger than over a long window, causing the graph to resemble a trumpet. We assume that the compliance criteria for pumps is fairly strict: The flow error can be no greater then 10% and must drop to less than 2% after 100 milliseconds of infusion. BolusReqEvent represents an event that is generated when the patient requests a bolus from the patient.

GenAlarm represents the activation of a non-specific technical alarm. The alarm can result, for example, from the pump detecting some hazard such as air bubbles in the infusion tube, some mechanism overheating, or the drug resevoir becoming empty. Our assumed compliance criteria only requires that the device generate an audible alarm at the same time as the event is fired.

6.4.2 Application Design

Our application for preventing PCA overdose (Figure 6.9) works as follows: The pulse-oximeter periodically samples the patient's SpO_2 and then transmits the (possibly averaged) sample to the application module (Figure 6.12). If the received SpO_2 value is < 95 then the application sends a signal to the pump's `disable` network port. Otherwise the it will send a signal to the pump's `enable` port. When the pump gets an enable signal it will become receptive to bolus requests from the patient. If a bolus is active when the pump gets a disable signal the infusion is stopped.

Unlike the laser/ventilator interlock, satisfaction of the safety requirement depends on the possible interations between the devices, application code, *and* patient: It is impossible to asses whether or not the safety requirements are satisfied without coupling the system specification to some model of patient physiology and behavior. So instead of walking through each specfication with the aim of showing the reader why the system "is safe" we will simply give a functinal overview of the specifications and discuss why certain behaviors are present. The question of safety verification will be examined in depth later in the safety argument section.

Figure 6.10 gives the pulse-oximeter specification. This particular specification does not admit any modal variability (*i.e.,* there is no timing or functional variability allowed). Every 100 milliseconds the pulse-oximeter samples the SpO_2 value from the patient and then stores the sample in a 2-history buffer. If the $SpO_2 \geq 95$ the pulseoximeter will transmit the average of the buffer to the application. If the SpO_2 is < 95 then it will transmit the most recent sample. The reasoning is that if the SpO_2 is high the averaging effect will help avoid spurious

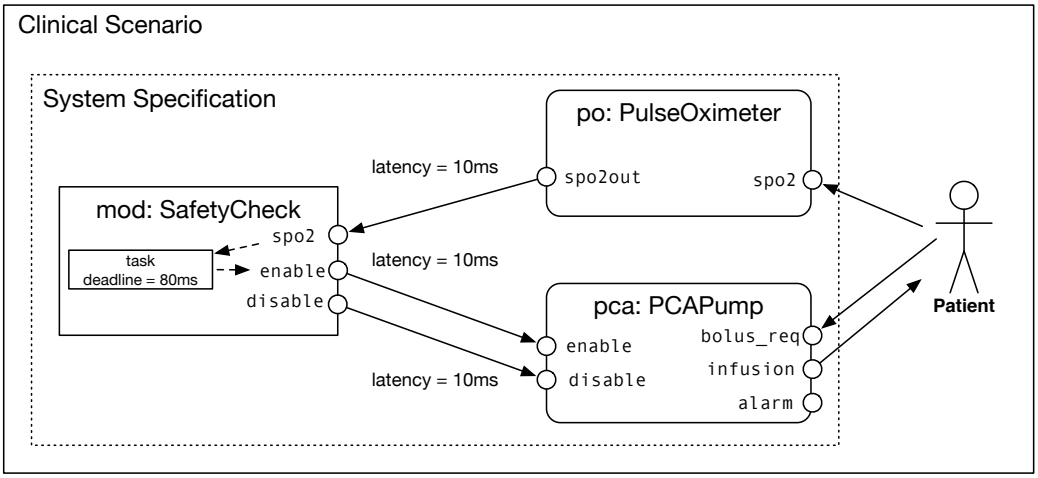


Figure 6.9: Closed-loop management of PCA clinical scenario and system specification.

alarms, but if the SpO_2 is low greater sensitivity is desired. While this specification does not model any particular pulse-oximeter, it does capture the types of behavior seen in real pulse-oximeters.

Figure 6.11 shows the specification for the PCA pump. The specification dictates that the pump should not respond to patient bolus requests while it is disabled. This specification admits both functional and timing modal variability. The timing variability is designed to admit pumps with different pump mechanisms: Different pump mechanisms can impact how long it could take to start or stop an infusion. Observe the parametric timing constraints in `STARTB`, `STOPB`, `STOPBD`, and `STOPBDE`. These locations are used to model the time it takes to start and stop a bolus respectively. The starting locations are constrained with a different set of parameters than the stopping locations to allow for pumps that take a differnt amount of time to start vs. stop. The functional variability present in the specification allows for pumps that can detect certain error conditions (*e.g.,*

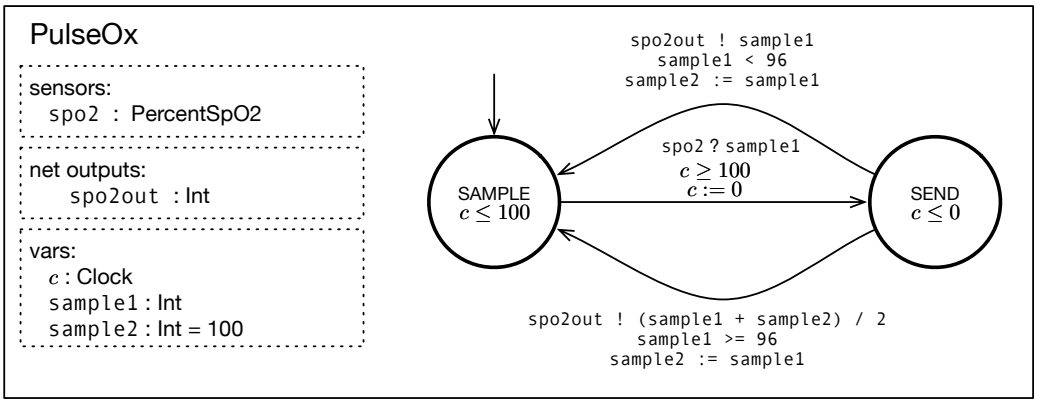


Figure 6.10: Requirements for pulse-oximeter behavior

bubbles in the infusion line) and automatically cease infusion.

Figure 6.12 shows the PCA management application's software module. Whenever it gets a new SpO_2 sample it checks if it is above or below the safe threshold and enables or disables the pump accordingly.

6.4.3 Safety Argument

Like the Laser/Ventilator interlock application, our safety argument for the PCA management application (Figure 6.14) is an instatiation of the argument pattern from Section 2.4. The top-level goal of the argument (**G: NoOverinfusion**) is to argue that there is adequate assurance that *all* possible instantiations of the the PCA management application permitted by the ecosystem prevents PCA therapy from depressing the patient's SpO_2 levels to $< 75\%$.

Like before, the top-level goal **G: NoOverinfusion** is discharged using the *platform argument strategy* (**S: PlatArg**Section 2.4) which requires model-based reasoning step (**G: ModelSat**), and then that all the models used have their adequacy justified. Again, we take advantage of our imagined ecosystem's reliance

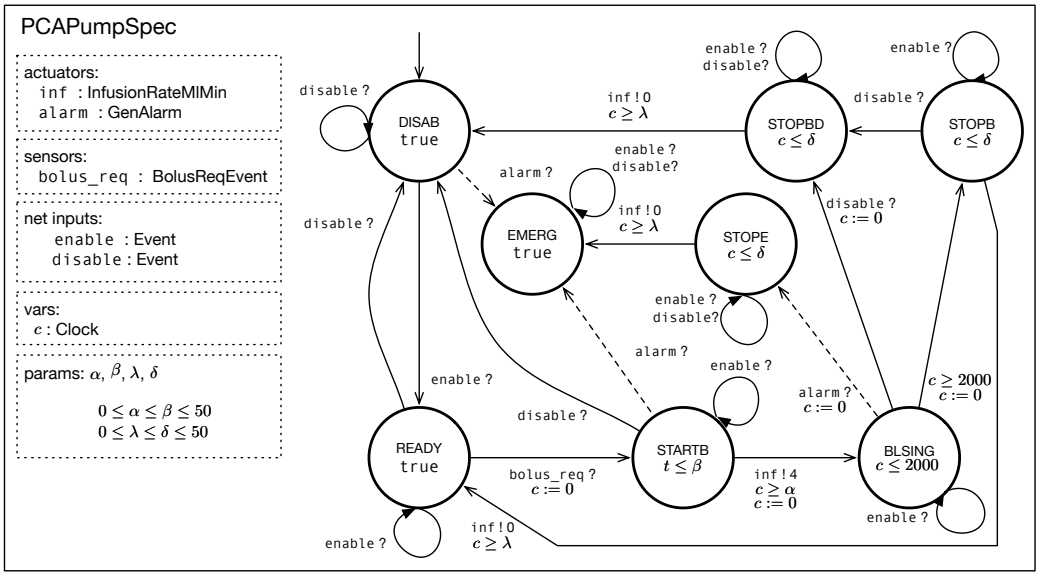


Figure 6.11: Requirements for PCA pump behavior

on the ODSDL and the certified DVE toolchain: The **S: PlatArg** strategy is used in the context of models are automatically derived from the ODSDL application specification (except the patient model).

The model-based reasoning step is given in the sub-argument of goal **G: ModelsSat**. We verified that the applications satisfies the desired property by using the DVE toolchain to automatically extract ModalT TPMS and Imitator PTA models from the ODSDL application specification. To model the patient's physiology we hand-crafted a TPMS model (Figure 6.13) of the patient behavior and pharmokinetics (*i.e.,* the relationship between infusion rate, and the patient's drug- and SpO_2 levels). This patient model is the same model used in [149, 21] and was originally derived from a simple text-book model of opiod pharmokinetics.

The patient model is a "discrete simulation" of the patient's opiod metabolism. Each 100 milliseconds it takes a "simulation step". At each step the current drug

```
1 module SafetyCheck {
2   net input "spo2" datatype Int  minsep 100
3   net output "enable" datatype Event minsep 100
4   net output "disable" datatype Event minsep 100
5   vars { }
6   task activated by port "spo2"  delay 80{
7     if("spo2" < 95){
8       send("disable")
9     }
10    else{
11      send("enable")
12    }
13  }
14 }
```

Figure 6.12: SafetyCheck module for the PCA control application

level is computed by adding the current infusion rate to the current drug level and substracted out the amount absorbed or metabolized away. The SpO_2 level is then computed by subtracting the drug level from 100. At any time, the infusion pump can signal to the patient model of a rate change. Likewise, at any time, the pulse-oximeter can sample the patient's current SpO_2 level. Patient behavior is modeled with the value of the `tolerance` variable: If the drug level is less than `tolerance`, the patient can request a PCA bolus. The patient model is always receptive to alarm outputs from the pump (we have elided tese transitions from Figure 6.13 for simplicity).

Like with the Laser/Ventilator application we use model-checking to verify two properties:

- **P1**: Cannot reach a state such that`Patient.spo2` < 75.

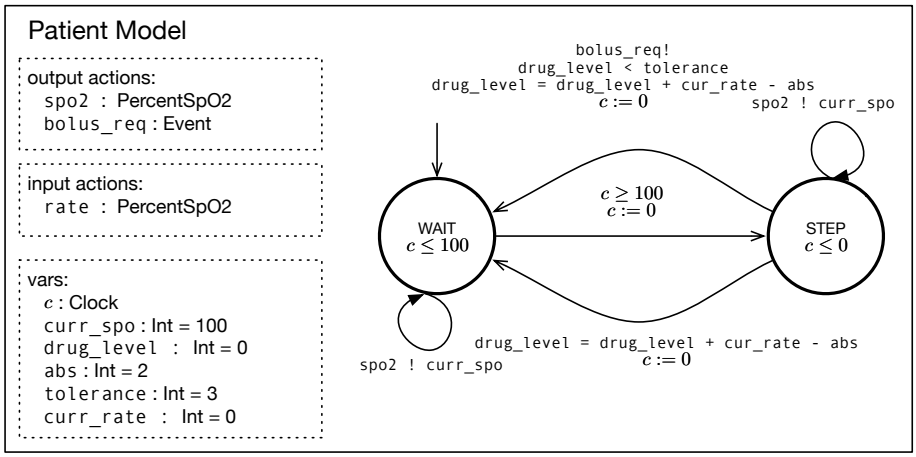


Figure 6.13: TPMS model of patient behavior and opiod pharmokinetics.

- **P2**: Cannot reach a timelock.

and like with the Laser/Ventilator application we use the two model-checkers ModalT and Imitator. Both tools verified that both **P1** and **P2** are satisfied by the system model.

We justify the adequacy of the patient model (**G: PatientAdq.**) by arguing the model is *sound*, *i.e.,* real patients react much less agressively to opiod infusion compared to the model. The idea being if the model patient does not experience overdose, neither will a real patient. The evidence we use to support this argument is a citation to a textbook on opiod pharmokinetics.

Figure 6.15 illustrates the argument supporting the adequacy of the pulse-oximeter model (**G: PulseOxModelAdq.**). Like with the laser/ventilator application there are two parts to the argument: That the model used in the model-checking captures all the relevent behavior of the compatible device models and that the behavior of the device models captures the relevent physical behavior of the devices. Like before, the first part is discharged with an appeal to the relevent

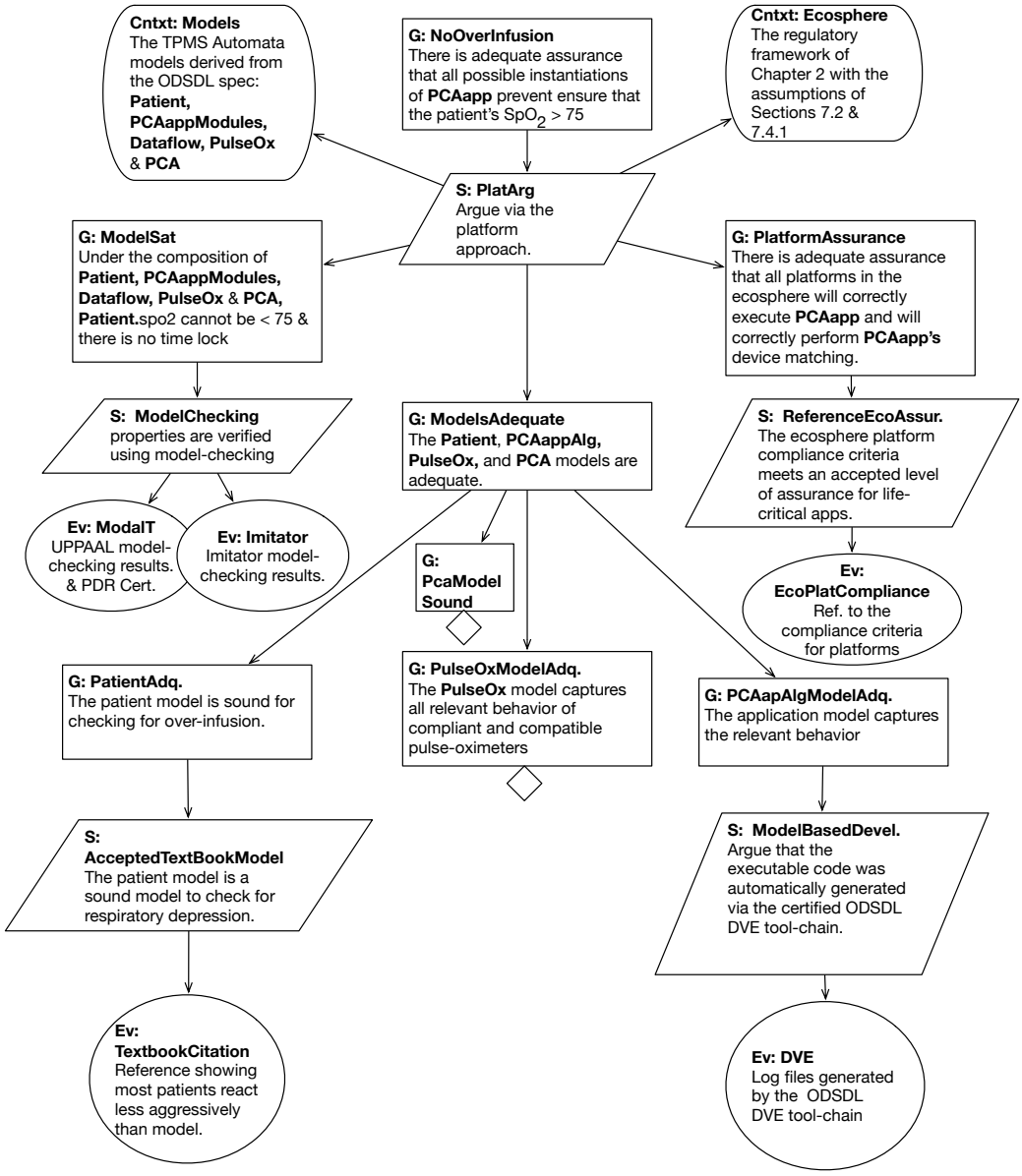


Figure 6.14: Assurance case fragment for the PCA management application.

theorems on TPMS refinement, property preservation, and compositional reasoning. For the second part we argue that any divergence from the modeled behavior (in terms of timing or sensring errors) allowed by our ecosystem's pulse-oximeter compliance criteria is small and inconsequential relative the pharmokinetics and patient physiology in question. For example, our assumed compliance criteria for pulse-oximeter sensing error is +/-1 %. While this may allow a slight divergence of the system's modeled behavior with reality, it won't make much practical difference to the health of the patient.

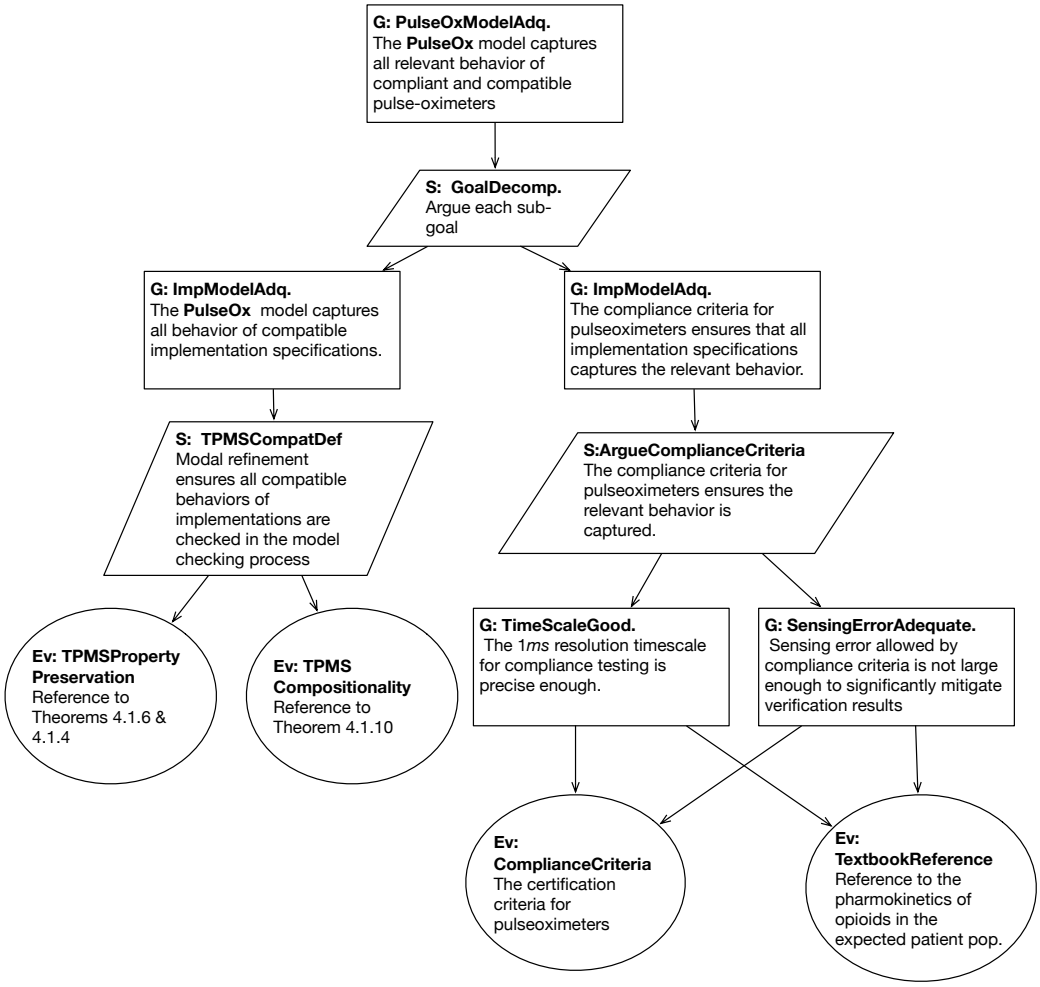


Figure 6.15: Arguement fragment for the adequacy of the pulse-oximeter model.

Potential Assurance Deficits

This safety argument for the PCA management application can harbor the same kinds of deficits as those discussed for the laser/ventilator application. The review might find fault with the top-level safety requirement (*e.g.,* that it is incomplete).

They also might find fault with the argument used to justify the adequacy of the device models or platform. We will not re-hash the same deficits for this argument. Instead we will focus on a new deficit related to the substance of the argument resulting from the top-level safety requirement.

Unlike the laser/ventilator application, the safety requirement for the PCA management application is phrased in terms of the patient's physiologic state: The PCA must cause the patient's SpO_2 to drop below 75. For the laser/ventilator application it is easy to have confidence that the environment model is justified (its simple and records the modes of the devices). The same cannot be said for the patient model used here. Modeling of human physiology is very difficult, and most so called high-fidelity models are quite complex (*e.g.,* tens of differential equations with tens of variables interacting with non-linear dynamics). The situation is further complicated when one tries to model sick patients where multiple disease states are interacting. An expert reviewer might simply not accept that our patient model adequately or soundly captures the behavior of the patients in the population our application targets.

6.5 Performance Evaluation

It is important to ascertain the performance implications of using *modal refinement* of TPMS as our notion of application device compatibility. While the refinement checking problem for TPMS is at least as hard as checking two timed automata for timed (bi)-simulation (known to be an EXPTIME problem [48]), the algorithm we present in Section 3.3.5 uses a number methods (symbolic zones and modern SMT solver capability) to potentially decide modal refinement quickly in practice.

The goal of this section is report on how well an implementation of our algorithm performs on the specifications used in our case studies.

6.5.1 Setup

For this evaluation, we implemented the scale insensitive refinement checking algorithm from Section 3.3.5 using the Scala programming language. Zones were maintained using a difference bound matrix [60, 33] library from the PRISM model checker [116]. Our implementation used the Z3 SMT solver [57] version 4.4.2 to check for zone-closure and guess valid parameter values. Each refinement check was run on a Linux workstation equipped with an Intel Core i7-4770 CPU with 32GB of RAM. All timing information capture includes the time taken to desugar an ODSDL specification into a TPMS and then run the actual refinement checking algorithm on the TPMS.

For each device requirements specification we ran seven experiments which we call Good, Tau1, Tau3, Tau5, BadConstant, BadArith and BadTrans. The Good experiment simply took the device specification and transformed it into a valid *implementation* by fixing appropriate parameter values and turning any may transitions into must transitions. The Tau1, Tau3, and Tau5 are designed to increase the state-space of the implementation with extra τ (silent) transitions that do not affect the I/O behavior of the implementation. For each Tau$N$ experiment we took the Good specification and *interleaved* in N =1,3 or 5 τ-transitions. Each extra τ-transition has the affect of doubling the effective state space of the device implementation specification.

For the Bad$X$ experiments we took the device implementation specification created for Good and mutated it to create a bad implementation. BadConstant

randomly selected a constant used in a clock constraint and changed it to some random value. BadArith randomly selected some arithmetic expression (*e.g.,* used in a state variable update or output action) and changed it by either changing the operator or by extending the expression with some new operation. For BadTrans randomly added a new transition.

6.5.2 Results

Figures 6.16 & 6.17 show the results for the Laser / Ventilator interlock system specifications and Figures 6.18 & 6.19 show the results for the PCA management system.

We immediately see that our refinement checking algorithm is able to decide refinement in under 9 seconds under all experimental variations for 3 of the 4 device specifications. Also, in general, it takes longer to check the refinement of a valid implementation. This extra time makes sense, because the algorithm must build and check the entire product zone graph of the specification and implementation when the implementation is valid.

As expected, refinement checking takes much longer when the state-space of the specification is large. This phenomenon is exemplified most with the pulse-oximeter specification. The pulse-oximeter specification contains two state-variables (in addition to the control location variable) that buffer the sampled SpO_2 readings from the patient. The buffer variables can take on 100 possible values. Combined with the two control locations the specification's discrete state-space (ignoring clock valuations) is 20,000. Interleaving in the tau transitions quickly causes the state-space to explode (remember the refinement checking algorithm has to potentially check the cross-product of the states). Nevertheless,

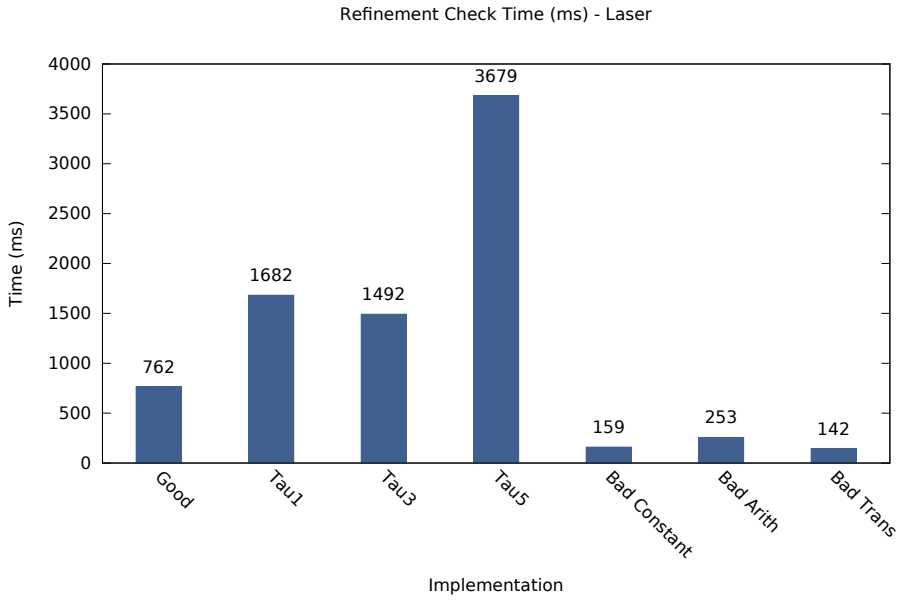


Figure 6.16: Refinement checking time - laser-scalpel specification.

refinement is still decided in just over 6 minutes for the Tau5 version of the pulse-oximeter implementation. While this might be too slow for a compatibility check at the bed side during a medical procedure, it is still probably fast enough for a compatibility "pre"-check performed by hospital technical staff prior to use. A less significant example of this phenomenon occurs with the ventilator specification. The ventilator specification allows the operator to adjust the programmed flow rate from the front panel (range of $1 - 50$). Even with this internal state, the Tau5 version of the ventilator implementation is still checked for refinement in about 8 seconds.

One interesting comparison is between the PCA pump and laser scalpel specifications. Both specifications don't contain any significant discrete internal state (the PCA pump has 8 control locations vs. the laser scalpel's 7) yet checking the

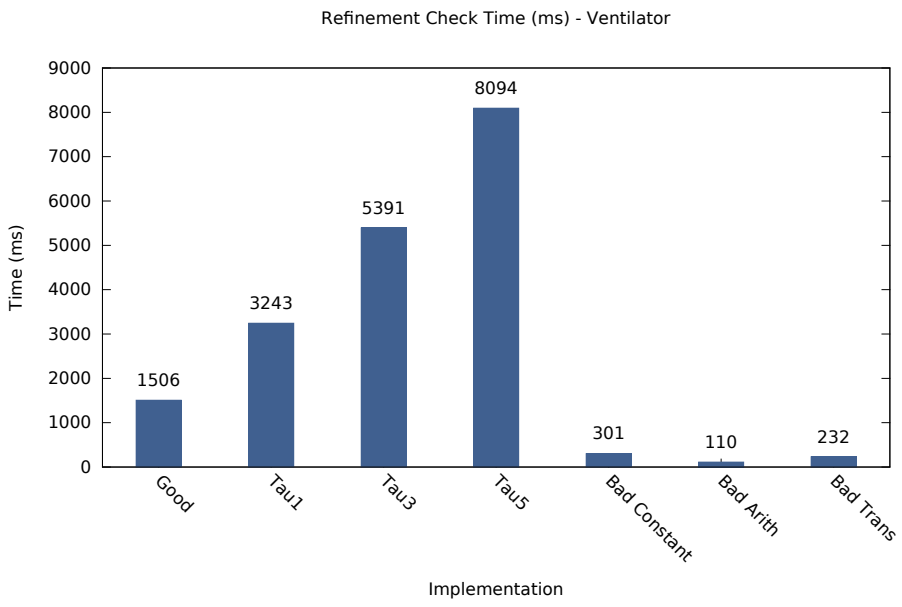


Figure 6.17: Refinement checking time - ventilator specification.

valid implementations of the laser scalpel take signficantly longer. Why is this? While we did not do any profiling of our algorithm the difference likely results from the extra clock in the laser scalpel's specification. More clocks mean the expression sent to the SMT solver must quantify over more variables and the DBMs used to store the concretized zones have more dimensions. Furthermore, more clocks are known to make zone graphs larger (to account for clock differences that arise due to clock resets).

Finally, we observe that without exception, deciding modal refinement for invalid implementations was always very fast (just a fraction of a second). Why is this? We believe that the speed results from the fact the algorithm essentially does a breadth first search of the product statespace, and the types of errors we introduced at the syntactic (ODSDL) level always manifest shallow in the semantic

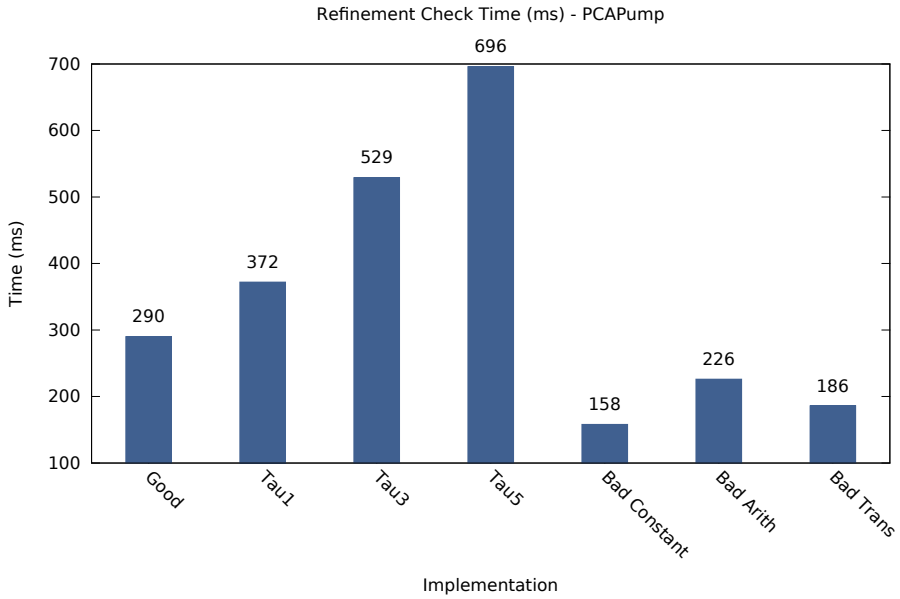


Figure 6.18: Refinement checking time - PCA pump specification.

(TPMS) level.

6.6 Related Work

To our knowledge there has not been any case-studies illustrating how to design and assess an application for an ecosystem of interoperable medical systems. However, there has been numerous case-studies designed to study specific clinical scenarios where automation could be applied. There have also been a number of case-studies where investigators have examined some device specific issues related to interoperability. We will give a brief summary of the most relevent works.

In [104] Kim *et al.* looked at the laser surgery fire prevention scenario and proposed a system that could preserve its safety properties even in the presence

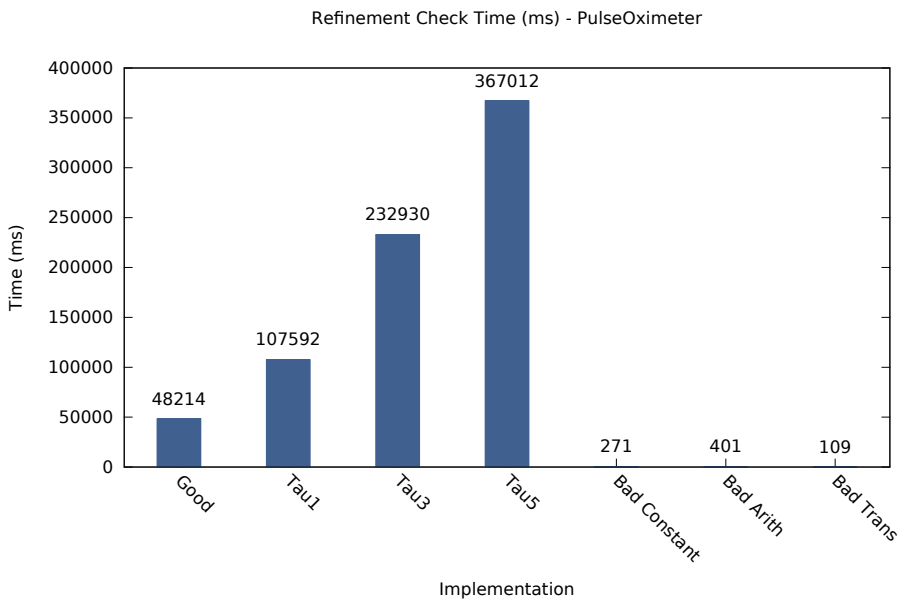


Figure 6.19: Refinement checking time - pulse-oximeter specification.

of network failures. Kim *et al.*'s system worked by synthesizing a coordination protocol on the fly based on safety property predicates declared before runtime. In [20, 19], Arney *et al.* looked at the problem of synchronizing an x-ray machine with a ventilator. There, the problem is sometimes the surgeon wants to take a chest x-ray while the patient is under artificial ventilation. Arney's *et al.*'s system sychronized the x-ray with the ventilator so the x-ray image was taken when the patient's lungs were at peak inspiration (*i.e.,* moving the least). This capability would allow surgeons to take x-rays without having to turn off the ventilator.

Bu *et al.* [45] examined a version of Kim *et al.*'s system and proposed a type of runtime verification to detect when specified safety constraints were violated. Bu *et al.*'s argument being that model-checking a system *a priori* can be impractical and perhaps it is better to simply detect problems when they happen and alert the

operator rather than attempt to avoid problems altogether.

David Arney and colleagues also studied the problem of preventing overdose due to PCA therapy [21]. Arney *et al.* designed several closed-loop control protocols and verified different safety properties using both model-checking and simulation. One of their protocols was designed to be resilient to network faults (*i.e.,* the safety properties were preserved in the presence of dropped messages). Pajic *et al.* published a much more detailed version [149] of the study that originally appeared in [21].

Larson *et al.* looked at the *requirements engineering* process for infusion pumps in [122] with a special eye towards requirements relating to interoperability. Jakub Jedryszek expanded on [122] in his master's thesis [95] where he investigated how model driven development, in the context of AADL models, could be applied to interoperable infusion pumps. Likewise, Shiwei Luan's masters thesis [132] described the development of a modular infusion pump enabled with interoperable capabilities.

Chapter 7

Conclusion

The goal of this dissertation was to illustrate how a collection of inter-related techniques and technologies could help us achieve safe on-demand medical systems. While the approach we described most likely isn't the only way to achieve on-demand system safety, it provides concrete technical details we hope can serve as a foundation for future technical investigations and discussions on on-demand system safety.

First we described the concept of "on-demand systems" and how they differ from traditionally engineered systems. We identified three ways vendor neutral on-demand capabilities could benefit healthcare: 1) Reduced medical errors and clinician workload, 2) Device procurement flexibility for the Health Delivery Organization (HDO) and 3) Enable innovation by allowing smaller device and software companies entry to the health-care technology market. We also identified some key technical challenges. The list of challenges we identified is certainly not complete (*e.g.*,, we do not address the security challenges facing connected medical systems), but they are nonetheless important. We identified 1) Scalability

limitations in current regulatory frameworks, 2) Standardization of behavior and 3) Timing predictability in open & distributed dystems as three challenges that must be overcome if we want a viable ecosystem of on-demand medical devices and software. We argued that if we fail to overcome these challenges then the viability of the on-demand concept will be diminished.

We set out to address these challenges with a number of inter-related technical solutions. First we proposed a regulatory framework and associated assurance argument pattern. The regulatory framework makes application developers ultimately responsible for the system safety argument and enables a sort of compositional reasoning for safety. Devices would get certified against an interface (behavior) specification. Applications would specify their required devices and timing constraints and would be certified assuming their requirements are always satisfied. Specialized computing platforms called Medical Application Platforms (MAPs) would form a trusted base and ensure that applications are only used with compatible devices and that the timing constraints are always satisfied.

Then, we introduced Time Parametric Modal Specifications (TPMS). TPMS were designed specifically as a device behavior specification language for use in an on-demand ecosystem regulated by our framework. TPMS allow application developers to precisely specify required behavior and device manufacturers to precisely specified what behavior their device offers. TPMS allows for "loose" specifications (*i.e.,* the application developer can specify the acceptable timing and functional variability of a device) and offers a notion of refinement that can relate device behavior to application requirements. TPMS refinement enables the type of top-down reasoning required by our regulatory framework because it preserves safety properties.

Next we proposed the On-Demand Systems Description Language (ODSDL) for application developers to write on-demand applications. The ODSDL lets application developers specify the *logical architecture* of the system their application creates (essentially they specify a virtual system). The ODSDL has two features to support system predictability. First, it uses TPMS to explicitly specify required device behavior. Second, it has predictable logical execution time semantics (LET) which guarantees input deterministic exection of ODSDL applications.

We implemented a prototype Medical Application Platform with the MD-CF/MIDAS and showed how the LET semantics of ODSDL applications can be ensured in an open and dynamic "plug & play" environement. We described some modifications of classical deadline-monotonic fixed priority scheduling theory that enables its use in a dynamic and open environment. We assesed evaluated our implementation and showed that the LET semantics were properly implemented and that the prototype platform is resilient to "babbling idiot" faults.

Finally, we showed how each of the contributions above work together to enable system safety with two case-studies. We took two real clinical scenarios (closed-loop management of PCA infusion and coordinating a laser scalpel/ventilator), designed applications to address each scenario's safety requirement using ODSDL, and then constructed a safety argument for each application assuming our proposed regulatory framework exists. We then used the device specifications created for the case-studies to evaulate the performance of our TPMS refinement checker.

7.1 Discussion

7.1.1 Can we truly ensure safety?

While we believe the work presented in this dissertation can serve as a blueprint of techniques that can help practitioners create safe on-demand systems, the work presented here does not guarantee system safety. Our goal instead was to show how an appropriately designed regulatory framework and associated technologies would allow the application developer to constrain possible system behavior with enough confidence to make a reasonable safety argument. The work presented here does not prevent a designer from producing an unsafe application, nor does it prevent the regulator from approving an unsafe application: If the developer does bad model-based reasoning and the reviewer doesnt detect that bad reasoning, then they may allow a flawed application on the market. Concrete examples of possible *assurance deficits* were given with the case studies in Chapter 6. The flawed application may then cause an accident when it is instantiated and controlling real devices.

Should we not try and build on-demand systems for safety critical situations until safety can be guaranteed? We think that perspective is unreasonable, especially considering the techniques, technology and current regulatory processes cannot 100% guarantee the safety of traditionally engineered monolithic systems. Indeed, even for current traditionally engineered monolithic systems, safety assessment at some point comes down to (hopefully expert) judgement. Why is this? Its simply not possible for a human (i.e., designer, engineer, regulator and operator) to know everything. For a given system we cannot know, with 100% certainty, how it will behave or what the complete list of hazards are or even a

complete notion of what "safe" is. Effectively, each time someone declares that a system is safe they are actually doing a form a model based reasoning. While, the "model" may not be a formal model, it still will be some approximation of real-world behavior based on previous experience. Just like for our framework, it is ultimately up to the reviewer to judge whether the models, evidence, and overall argument is sufficient given the intended use of the system and the associated risk/benefit tradeoffs.

7.1.2 Do We Really Need Timing Guarantees?

A significant portion of this dissertation is devoted to describing how to make a platform that can guarantee the timing distributed behavior of a system in an open and dynamic environment. One may question whether strict timing guarantees are really necessary to support plug & play medical applications and whether it would be worth it to engineer a real platform capable of making those guarantees. Indeed, many of the clinical algorithms proposed to address the clinical scenarios enumerated by the MD PnP program are designed to tolerate deadline misses and even network message drops (See, for example, [21, 149, 20] or [104]).

First, just because many applications don't depend on strict timing guarantees does not mean there aren't others that will. For example, many closed-loop control applications (*e.g.,* diabetic glucose control [151, 112, 51]) do not have currently known solutions that are tolerant to deadline miss or network message loss.

Second, just because an application (or scenario) does not need timing guarantees does not mean it wont benefit from it. If a doctor or nurse deploys two applications at the same time and these applications have significant mutual resource contention, so much so that deadlines are consistently missed, then those

applications will consistently be in a “failsafe” mode and not able to offer clinical utility. Furthermore, guaranteed timing and other determinism features simplifies the job of the application developer: They will have fewer behaviors they must consider and anticipate during design [124].

7.1.3 The Cost and Benefit of Modal Refinement.

Some readers may question the wisdom of using refinement checking to check the compatibility between applications and devices. While our case-studies show that in practice refinement checking can be relatively quick, TPMS refinement checking is at least an EXPTIME problem [48] and the decision procedures we have presented need to unroll a zone-graph and are susceptible to the state space explosion problem.

Our decision to use modal refinement as our compatibility relation is the conscious result of a trade off. It seems that, on the one hand we could standardize device behavior and make compatibility checking very easy (e.g., a developer could simply specify they need a device by its standard “type name). But that approach would mean all the stakeholders would need agree on the standard behavior for each device type. As discussed in the introduction, we don’t have confidence such a standard could be achieved in a timely manner.

On the other end of the spectrum is our approach, where we standardize a simple language to model behavior and rely on potentially expensive procedures to decide compatibility between an applications requirements and a devices capability. Is there some middle ground approach that doesn’t require a complete standardization of device behavior but also doesnt need an expensive compatibility checking procedure? At this time the answer is not clear.

7.2 Gap Analysis & Future Work

Here we provide a combined "gap analysis" and future work section. The goal of the gap analysis is to describe the "delta" between the current state of affairs and what would be necessary to achieve safe medical device interoperability using the framework described in this dissertation. Unlike a traditional future work section, which usually focuses on establishing a direction for technical research, this section attempts to discuss the major technical *and* social gaps (which we further refine into the categories of *process* and *organization*). Indeed, for the problem of safe medical device interoperability, it is likely impossible to completely divorce the technical concerns of a solution from the social (*e.g.,* regulatory) context, therefore it is important to describe the current gaps on both fronts, and how they inter-relate.

Table 7.1 provides a condensed summary of the gap analysis. Each row describes a requirement to make our framework work, a summary of the current (as of this writing) status, suggestions for future work to remedy deficiencies in the current status, and then an indication as to whether the gap would be addressed by primarily social or technical means. For the rest of the gap analysis, we will devote a subsection to discuss each row of Table 7.1.

7.2.1 Regulatory Support for Compositional Reasoning

Overall, the regulatory framework proposed in this dissertation is highly speculative. As of the writing of this dissertation, we are not aware of any regulatory framework that allows for compositional certification or approval: *i.e.,* where individual components certified against an interface could be combined in such a

Requirement	Current Status	Needed Future Work	Social or Technical?
Regulatory support for compositional reasoning.	Only monolithic and/or ”pairwise” medical device system approval.	Development and regulatory acceptance of interop. device ecosystem similar to that as described in Chapter 2	Organ. & Process
Suitable criteria for device interface compliance.	Various “academic” approaches exists.	Investigate what types of evidence is needed to support an assurance argument.	Organ. & Technical
Physiological models appropriate for model based reasoning.	Models exist that capture patient physiologic at various levels of accuracy, complexity and fidelity.	Develop models that are amenable to model based reasoning and adequately capture patient dynamics.	Technical
Expressive formal device interface language.	TPMS expresses the timed-reactive behavior of devices with variability.	Extend TPMS to supports hybrid behavior and other non-func. properties.	Technical
Trustworthy devices.	Cryptographic techniques to validate device certification are well understood.	Techniques to detect device-interface behavior divergence	Technical
Rigorously verified and validated platforms.	Operating systems have been formally verified for data handling properties.	Investigate how to formally specify and verify non-functional properties (*e.g.*, timing).	Technical

Table 7.1: Requirements, current status and gaps.

way that enables compositional system level safety arguments. In the abstract, we don't believe that there are any serious *technical* reasons why such a regulatory framework could not be adopted. The real challenge is organizational in nature: all the various stakeholders would need to come together to form and manage an interoperable component ecosystem such as described in Chapter 2. In practice, this could be quite difficult as all the stakeholders would need to come to consensus on the interoperability protocol as well as the compliance criteria for devices, applications and platforms. Each of these issues can be quite complicated individually, together even more so. Additionally, different stake holders will have difference incentives affecting how willing they will be to compromise (*e.g.,* device manufacturers will likely try to promote device compliance criteria that favors their own devices).

Once an interoperability ecosystem has been formed, there is also the issue of creating processes that lead to *consistent enforcement*. In practice the compliance criteria agreed upon by the stakeholders will likely require some subjective judgement on the part of the certifier (*e.g.,* how do we know if the hazard list for a specific device is complete?). It will be difficult for stakeholders to anticipate the composite behavior of a system if the component compliance criteria is enforced inconsistently. We believe that this is primarily a social challenge. It will require that the stakeholders properly train and monitor the individuals tasked with certification.

Last but not least, the actual regulators will need to accept the certification practices of a given ecosystem. The best way to address this challenge is again through primarily social channels. For example, some regulatory agencies, such as the US FDA, allow interested parties to make a regulatory submission for a

non-existant or imaginary product (FDA calls this a "Pre-IDE" [93]). A regulatory submission, based on the assumptions and ecosystem framework described in this dissertation would help open a social channel to the regulator and help us get feedback from the regulator and identify regulatory & legal hurdles for the adoption of a interoperable ecosystem. Feedback from the mock submission could then be used to refine the overall design of the ecosystem and help the stakeholders propose changes to any existing laws that are limiting the adoption of interoperable medical systems.

7.2.2 Suitable Criteria for Device-Interface Compliance

This dissertation assumes that there exist "suitable" criteria for device-interface compliance. Indeed, in Chapter 6 we give some specific examples for specific device types. Unfortunately, our examples are clearly incomplete; we only assume as much is needed to help make the safety argument for the case study applications. In reality an ecosystem will need comprehensive certification criteria for all the different device types. The challenge here is both organizational *and* technical. The organizational aspect comes the fact the different stakeholders will likely need to compromise: Detailed and rigorous criteria are good in the sense that they may enable better prediction of device and composite system behavior. On the other hand, overly detailed and rigorous criteria may prove to onerous for certain stakeholders, *e.g.,* small device manufacturers without enough resources to meet the criteria.

There is also the very important technical and scientific question of how to relate device compliance evidence to assurance for device behavior. For example, one may require that device manufacturers follow a model-driven development

process where they first specify their device's behavior as a Timed Automata (TA). Then a tool like TIMES [9] would be used to generate code that exhibits the reactive and timing behavior specified by the TA. Now, if the device compliance criteria only required the manufacturer to provide evidence that TIMES was used to generate code from the model we probably shouldn't have much confidence that the specified timing behavior is perfectly achieved: timing divergence can arise due to CPU non-determinism and other effects. Therefore, the code should be tested on its hwardware platform to validate the timing behavior [96]. The question then becomes, how much testing is needed to achieve the desired level of confidence or assurance? Answering this question is a very important direction for future scientific work.

7.2.3 Physiologic Models for Model Based Reasoning

Many medical applications will likely phrase some or all of their safety requirements in terms of the patient's physiological state (*e.g.*,, our PCA case-study). If safety requirements will be phrased in terms of the patient's state and model based reasoning must be used to verify & validate system safety, then we need good patient models that can be incorporated into a model based reasoning framework. Good patient models are primarily a technical challenge. While there are physiologic models available for a wide variety of patient conditions and clinical scenarios, their accuracy, fidelity, or complexity may not be appropriate for the types of model based reasoning proposed in this dissertation. For example, the patient model used in our PCA case-study is simplified approximation of patient opiod pharmokinetics and likely misses out on certain safety relevant behaviors found in real patients. Future work should focus on developing models that have

enough fidelity to capture adequate amounts of safety related behavior while still being amenable to practical model based reasoning. Alternatively, the community could also investigate new model-based reasoning techniques that can effectively utilize many of the existing high-fidelity and complex models currently in the medical literature.

7.2.4 Expressive formal device interface language

TPMS as presented in this dissertation only allow developers to model "digital" system behavior with respect to real-valued clocks. When used as a behavior specification formalism in our framework this limitation forces the developer to informally associate idealized digital behavior with possible physical behaviors in the assurance argument itself. This reliance on informal reasnoning is one source of the assurance deficits we identified in our case studies. One important area of future work is to extend TPMS with the ability to model hybrid dynamics and non-functional properties including reliability and error.

The ability to fully model the hybrid dynamics of a device (*e.g.,* use a differential equation to model how the infusion rate of an infusion pump changes in response to a rate setting change) would allow application developers to reduce their reliance on informal reasoning. Likewise, support for non-functional properties such as reliability or error will help application developers make their device requirements more precise.

Furthermore, as mentioned earlier in the discussion, deciding refinement between two TPMSs can be expensive. In our case-studies, we discovered that deciding refinement is often quite quick, except when the discrete state-space of the specification was large. More efficient refinement checking procedures (*e.g.,*

exploiting efficient symbolic model-checking techniques) would be very useful in practice, and might be necessary to deal with complex specifications. Future work should focus on extending TPMS with support for more non-functional properties as well as more efficient models for analyzing TPMS specifications.

7.2.5 Trustworthy Devices

One critical assumption we make is that devices will behave "according" to their interface specifications. We rely on the certificaiton process to exclude devices that don't conform to their interface from the ecosystem. While we can rely on well understood crytographic techniques to automatically detect whether or not a device has been certified, it is much less clear how to check if a device has been mis-deployed, modified or *tampered* with after certification. A problem could be as simple as the nurse connecting a device to the wrong patient (*e.g.,* connecting the pulse-oximeter used in the PCA-interlock application to a different patient not actually undergoing PCA therapy). Or it could be as nefarious as an adversary that maliciously and subtly alters the behavior of a device to drive the patient to an unsafe zone. An important avenue for future technical work is to understand how we could enable the platform to automatically detect if a device is being operated outside its intended use (*i.e.,* on the wrong patient) or if its behavior is diverging from is specified interface.

7.2.6 Rigorously Verified & Validated Platforms

Our framework assumes that the underlying platform will operate correctly, *i.e.,* it will correctly perform the device application compatibility check and that it will

implement the LET semantics of the ODSDL. One important area of future work is to understand how the "timing guarantee" or timing enforcement features of a MAP could be formally verified. The separation property that separation kernels like seL4 have been formally verified to satisfy are fundamentally about the state of the system's *data* [111]. Formal verification is enabled because the semantics of the programming language used to implement the operating system, and the semantics of the underlying CPU's instruction set architecture (ISA) concern data-manipulation. Formally proving that an operating system guarantees the timing behavior of its hosted applications seems to be fundamentally more difficult. Neither the programming languages used to implement operating systems nor the underlying CPU ISA typically include any aspect of timing behavior in their semantics. While there has been some recent work formally verifying the worst case timing behavior of operating system operations [39], anyone hoping to formally verify an operating system that claims to offer application timing guarantees must find a way to bridge the current semantic gap.

Another important area of future work: In Chapter 5 we listed a number of features our prototype does not implement but a real MAP *ought* to have (Our prototype focuses mainly on timing guarantees). These features include enforcing time/space separation between applications, fault tolerant design, and a formally verified implementation. While all of these types of features have been studied extenstively in the literature it is not clear how all of these features would be implemented *in the same system*. Furthermore its not clear if these features might conflict. For example, our determinizing scheduler requires a single high priority task to orchestrate all the I/O events of the running applications. The determinizing scheduler must be able to preempt an application at any time to service I/O

events. Most separation kernels [55] enforce time/space isolation by executing each application partition for a pre-specified amount of time [181]. Incorporating a determinizing scheduler (or any sort of scheduling system that can change the system schedule dynamically) will impact the design of these operating systems and may introduce "knock-on" effects with regards to the verification of their separation features.

Bibliography

[1] The value of medical device interoperability. Technical report. 94

[2] Standard for iso/ieee health informatics - point-of-care medical device communication - part 10201: Domain information model. *ISO/IEEE 11073-10201:2004(E)*, pages 1–183, Jan 2005. 5

[3] Pica8 3290 Product Literature, 2013. 177

[4] Richard L Alena, Kenneth I Laws, Andre Goforth, and Fernando Figueroa. Communications for integrated modular avionics. 2006. 192

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 41, 89

[6] Rajeev Alur, Thomas A Henzinger, and Moshe Y Vardi. Parametric real-time reasoning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 592–601. ACM, 1993. 65

[7] Jim Alves-Foss, Paul W Oman, Carol Taylor, and W Scott Harrison. The mils architecture for high-assurance embedded systems. *International journal of embedded systems*, 2(3):239–247, 2006. 199

[8] Saeid Amini-Nik, Darren Kraemer, Michael L Cowan, Keith Gunaratne, Puviindran Nadesan, Benjamin A Alman, and RJ Dwayne Miller. Ultrafast mid-ir laser scalpel: protein signals of the fundamental limits to minimally invasive surgery. *PLoS One*, 5(9):e13053, 2010. 15

[9] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2003. 198, 243

[10] Étienne André. Imitator: A tool for synthesizing constraints on timing bounds of timed automata. In *Theoretical Aspects of Computing-ICTAC 2009*, pages 336–342. Springer, 2009. 65, 66, 209

[11] Étienne André. Imitator ii: A tool for solving the good parameters problem in timed automata. *arXiv preprint arXiv:1011.0223*, 2010. 66, 209

[12] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. Imitator 2.5: A tool for analyzing robustness in scheduling problems. In *FM 2012: Formal Methods*, pages 33–36. Springer, 2012. 66, 209

[13] Étienne André, Yang Liu, Jun Sun, and Jin-Song Dong. Parameter synthesis for hierarchical concurrent real-time systems. *Real-Time Systems*, 50(5-6):620–679, 2014. 65, 209

[14] Étienne André and Romain Soulat. *The Inverse Method: Parametric Verification of Real-time Unbedded Systems*. John Wiley & Sons, 2013. 65, 209

[15] Adam Antonik, Michael Huth, Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. 20 years of modal and mixed specifications. *European Association for Theoretical Computer Science. Bulletin*, (95), 2008. 36

[16] David B Apfelberg, Morton R Maser, Harvey Lash, and David N White. Benefits of the co2 laser in oral hemangioma excision. *Plastic and Reconstructive Surgery*, 75(1):46–50, 1985. 15

[17] ARINC653. Avionics application software standard interface - arinc 653, 2006. 192

[18] ARINC664. Aircraft data network - arinc 664, 2002. 192

[19] David Arney, Kunal Bhatia, Sanchit Bhatia, Michael Sutton, Tracy Rausch, Joel Karlinsky, and Julian M Goldman. Design of an x-ray/ventilator synchronization system in an integrated clinical environment. In *Conference proceedings:... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual Conference*, volume 2011, pages 8203–8206, 2010. 231

[20] David Arney, Julian M Goldman, Susan F Whitehead, and Insup Lee. Synchronizing an x-ray and anesthesia machine ventilator: A medical device interoperability case study. 2009. 7, 33, 231, 237

[21] David Arney, Miroslav Pajic, Julian M Goldman, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. Toward patient safety in closed-loop medical device systems. In *Proceedings of the 1st ACM/IEEE International Con-*

ference on Cyber-Physical Systems, pages 139–148. ACM, 2010. 7, 219, 232, 237

[22] David Arney, Krishna K Venkatasubramanian, Oleg Sokolsky, and Insup Lee. Biomedical devices and systems security. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 2376–2379. IEEE, 2011. 8

[23] Medical devices and medical systems - essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ice). http://enterprise.astm.org/filtrexx40.cgi?+REDLINE_PAGES/F2761.htm. 3, 32

[24] N.C. Audsley and Y Dd. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991. 173

[25] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. 158

[26] Neil C. Audsley. Deadline monotonic scheduling, 1990. 173

[27] Neil C Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 1994. 161

[28] Anaheed Ayoub, BaekGyu Kim, Insup Lee, and Oleg Sokolsky. A safety case pattern for model-based development approach. In *NASA Formal Methods*, pages 141–146. Springer, 2012. 34

[29] Sebastian S Bauer, Uli Fahrenberg, Line Juhl, Kim G Larsen, Axel Legay, and Claus Thrane. Quantitative refinement for weighted modal transition systems. In *Mathematical Foundations of Computer Science 2011*, pages 60–71. Springer, 2011. 89

[30] James M Beck and Elizabeth D Azari. Fda, off-label use, and informed consent: debunking myths and misconceptions. *Food & Drug LJ*, 53:71, 1998. 23

[31] Nikola Beneš, Ivana Černá, and Jan Křetínskỳ. Modal transition systems: Composition and ltl model checking. In *Automated Technology for Verification and Analysis*, pages 228–242. Springer, 2011. 88, 89

[32] Nikola Benes, Jan Kretínský, Kim Guldstrand Larsen, Mikael H. Møller, and Jirí Srba. Dual-priced modal transition systems with time durations. In *LPAR*, pages 122–137, 2012. 38, 90, 91

[33] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003. 43, 71, 74, 76, 226

[34] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. Modal event-clock specifications for timed component-based design. *Science of Computer Programming*, 77(12):1212–1234, 2012. 38, 90, 91

[35] Nathalie Bertrand, Sophie Pinchinat, and Jean-Baptiste Raclet. Refinement and consistency of timed modal specifications. In *Language and Automata Theory and Applications*, pages 152–163. Springer, 2009. 38, 90, 91

[36] E. Bini, Thi Huyen Chau Nguyen, P. Richard, and S.K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *Computers, IEEE Transactions on*, 58(2):279 –286, feb. 2009. 172

[37] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. 185

[38] Jennifer Black and Philip Koopman. System safety as an emergent property in composite systems. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 369–378. IEEE, 2009. 3

[39] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 339–348. IEEE, 2011. 246

[40] Gregory Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000. 183

[41] Anne Bouillard, Laurent Jouhet, and Éric Thierry. Tight performance bounds in the worst-case analysis of feed-forward networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010. 169

[42] Aaron R Bradley. Understanding ic3. 67

[43] Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011. 67

[44] Laurence L Brunton, Bruce Chabner, and Björn C Knollmann. *Goodman & Gilman's the pharmacological basis of therapeutics*, volume 12. McGraw-Hill Medical New York, 2011. 17

[45] Lei Bu, Qixin Wang, Xin Chen, Linzhang Wang, Tian Zhang, Jianhua Zhao, and Xuandong Li. Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior. *ACM SIGBED Review*, 8(2):7–10, 2011. 231

[46] CM Callan. An analysis of complaints and complications with patient-controlled analgesia. *Patient-controlled analgesia*, pages 139–150, 1990. 19

[47] R. Carroll, R. Cnossen, M. Schnell, and D. Simons. Continua: An interoperable personal healthcare ecosystem. *Pervasive Computing, IEEE*, 6(4):90–94, 2007. 5

[48] Kārlis Čerāns. Decidability of bisimulation equivalences for parallel timer processes. In *Computer Aided Verification*, pages 302–315. Springer, 1993. 78, 225, 238

[49] Kārlis Čerāns, Jens Chr Godskesen, and Kim G Larsen. Timed modal specificationtheory and tools. In *Computer Aided Verification*, pages 253–267. Springer, 1993. 89

[50] Karlis Cerans, Jens Chr. Godskesen, and Kim Guldstrand Larsen. Timed modal specification - theory and tools. In *CAV*, pages 253–267, 1993. 44

[51] Sanjian Chen, Matthew O'Kelly, James Weimer, Oleg Sokolsky, and Insup Lee. An intraoperative glucose control benchmark for formal verification. *IFAC-PapersOnLine*, 48(27):211–217, 2015. 237

[52] M. Clarke, D. Bogia, K. Hassing, L. Steubesand, T. Chan, and D. Ayyagari. Developing a standard for personal health devices based on 11073. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 6174–6176, 2007. 5

[53] FlexRay Consortium et al. Flexray communications system-protocol specification. *Version*, 2(1):198–207, 2005. 127

[54] A Cook. Arinc 653challenges of the present and future. *Microprocessors and Microsystems*, 19(10):575–579, 1995. 192

[55] Alan Cudmore. Current and future flight operating systems. 2007. 247

[56] Alexandre David, Kim G Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 91–100. ACM, 2010. 38, 90, 91

[57] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *TACAS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. 70, 226

[58] Patricia Derler, Thomas H Feng, Edward A Lee, Slobodan Matic, Hiren D Patel, Yang Zheo, and Jia Zou. Ptides: A programming model for dis-

tributed real-time embedded systems. Technical report, DTIC Document, 2008. 124, 193

[59] Patricia Derler, Edward A Lee, and Slobodan Matic. Simulation and implementation of the ptides programming model. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 330–333. IEEE Computer Society, 2008. 124, 193

[60] David L Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic verification methods for finite state systems*, pages 197–212. Springer, 1989. 226

[61] ECRI Institute. Surgical fire prevention. https://www.ecri.org/surgical_fires, 2014. Accessed: 2014-09-18. 16

[62] Kimberly K Egan and Lisa A Haile. Fda catches up with health it revolution. 2012. 4

[63] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003. 97

[64] Harald Fecher and Heiko Schmidt. Comparing disjunctive modal transition systems with an one-selecting variant. *The Journal of Logic and Algebraic Programming*, 77(1):20–39, 2008. 89

[65] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006. 96, 123

[66] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 39–48. ACM, 2006. 88

[67] Arkadeb Ghosal, Thomas A Henzinger, Christoph M Kirsch, and Marco AA Sanvido. Event-driven programming with logical execution times. In *Hybrid Systems: Computation and Control*, pages 357–371. Springer, 2004. 97, 193

[68] Jens Chr Godskesen. *Timed modal specifications*. PhD thesis, PhD thesis, Aalborg University, 1994. 38, 89, 91

[69] J Goldman. Advancing the adoption of medical device plug-and-play interoperability to improve patient safety and healthcare efficiency. *Medical Device "Plug-and-Play" Interoperability Program, Tech. Rep*, 2000. 2

[70] Julian M. Goldman. Getting connected to save lives. *Biomedical Instrumentation & Technology*, 39(3):174–174, 2005. 2

[71] Julian M Goldman. Medical device plug-and-play (md pnp) interoperability standardization program development. Technical report, DTIC Document, 2009. 3

[72] Julian M Goldman. Solving the interoperability challenge: Safe and reliable information exchange requires more from product designers. *Pulse, IEEE*, 5(6):37–39, 2014. 8

[73] Julian M Goldman, Richard A Schrenker, Jennifer L Jackson, and Susan F Whitehead. Plug-and-play in the operating room of the future. *Journal Information*, 39(3), 2005. 5

[74] K Grifantini. plug and play hospitals: Medical devices that exchange data could make hospitals safer, 2008. 14

[75] Matthew Grissinger. Misprogram a pca pump? its easy! *Pharmacy and Therapeutics*, 33(10):567, 2008. 18

[76] Cheryl S Hankin, Jeff Schein, John A Clark, and Sunil Panchal. Adverse events involving intravenous patient-controlled analgesia. *American journal of health-system pharmacy*, 64(14):1492–1499, 2007. 19

[77] John Hatcliff, Eugene Vasserman, Sandy Weininger, and Julian Goldman. An overview of regulatory and trust issues for the integrated clinical environment. *Proceedings of HCMDSS 2011*, 2011. 23, 34

[78] Constance L Heitmeyer, Myla Archer, Elizabeth I Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 346–355. ACM, 2006. 190

[79] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003. 97, 123, 193

[80] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, pages 166–184. Springer, 2001. 97, 123

[81] Thomas A Henzinger and Christoph M Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):33, 2007. 93, 193

[82] Rodney W. Hicks, Vanja Sikirica, Winnie Nelson, Jeff R. Schein, and Diane D. Cousins. Medication errors involving patient-controlled analgesia. *American Journal of Health-System Pharmacy*, 65(5):429–440, March 2008. 18, 19

[83] Vance Hilderman and Tony Baghi. *Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware)*. Avionics Communications, 2007. 26, 199

[84] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. 130

[85] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 109, 130

[86] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 157–171. Springer, 2012. 67, 70

[87] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. μz–an efficient engine for fixed points with constraints. In *Computer Aided Verification*, pages 457–462. Springer, 2011. 65

[88] Robert Matthew Hofmann. *Modeling medical devices for plug-and-play interoperability*. PhD thesis, Citeseer, 2007. 3, 5

[89] Ch Hornberger, Ph Knoop, H Matz, F Dörries, E Konecny, H Gehring, J Otten, R Bonk, H Frankenberger, P Wouters, et al. A prototype device for standardized calibration of pulse oximeters ii. *Journal of clinical monitoring and computing*, 17(3-4):203–209, 2002. 214

[90] Ch Hornberger, Ph Knoop, W Nahm, H Matz, E Konecny, H Gehring, R Bonk, H Frankenberger, Geert Meyfroidt, Patrick Wouters, et al. A prototype device for standardized calibration of pulse oximeters. *Journal of clinical monitoring and computing*, 16(3):161–169, 2000. 214

[91] Jana Hudcova, Ewan McNicol, Cheng Quah, Joseph Lau, and Daniel B Carr. Patient controlled intravenous opioid analgesia versus conventional opioid analgesia for postoperative pain control: A quantitative systematic review. *Acute Pain*, 7(3):115–132, 2005. 17

[92] Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. In *TACAS*, pages 189–203, 2001. 71

[93] Ide approval process. http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/, 2015. 242

[94] Jean-Bernard Itier. A380 integrated modular avionics. 2009. 193

[95] Jakub Jedryszek. *A model-driven development and verification approach for medical devices*. PhD thesis, Kansas State University, 2014. 232

[96] Eunkyoung Jee, Insup Lee, and Oleg Sokolsky. Assurance cases in model-driven development of the pacemaker software. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 343–356. Springer, 2010. 243

[97] Eunkyoung Jee, Shaohui Wang, Jeong Ki Kim, Jaewoo Lee, Oleg Sokolsky, and Insup Lee. A safety-assured development approach for real-time software. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 133–142. IEEE, 2010. 198

[98] Leslie A Johnson. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, 1998. 199

[99] Joint Commission. Sentinel event alert issue 33: Patient controlled analgesia by proxy. http://www.jointcommission.org/sentinelevents/sentineleventalert/, December 2004. 18, 19

[100] Joint Commission. Preventing patient-controlled analgesia overdose. *Joint Commission Perspectives on Patient Safety*, page 11, October 2005. 18

[101] Dilsun K Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed i/o automata. 43

[102] Tim Kelly and Rob Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*. Citeseer, 2004. 30

[103] Tim P Kelly. Concepts and principles of compositional safety case construction. *Contract Research Report for QinetiQ COMSA/2001/1/1*, 2001. 34

[104] Cheolgi Kim, Mu Sun, Sibin Mohan, Heechul Yun, Lui Sha, and Tarek F Abdelzaher. A framework for the safe interoperability of medical devices in the presence of network failures. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 149–158. ACM, 2010. 7, 33, 230, 237

[105] Yu Jin Kim, Sam Procter, John Hatcliff, Venkatesh-Prasad Ranganath, et al. Ecosphere principles for medical application platforms. In *Healthcare Informatics (ICHI), 2015 International Conference on*, pages 193–198. IEEE, 2015. 3

[106] Hiroaki Kimura. Laser scalpel, May 12 1981. US Patent 4,266,549. 15

[107] Andrew King, Kelsea Fortino, Nobby Stevens, Shalin Shah, Margaret Fortino-Mullen, and Insup Lee. Evaluation of a smart alarm for intensive care using clinical data. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 166–169. IEEE, 2012. 94

[108] Andrew King, Sam Procter, Daniel Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, and Sandy Weininger. An open test bed for medical device integration and coordination. In *Proceedings of the 31st International Conference on Software Engineering*, 2009. 3

[109] Andrew L King, Alex Roederer, David Arney, Sanjian Chen, Margaret Fortino-Mullen, Ana Giannareas, William Hanson III, Vanessa Kern, Nicholas Stevens, Jonathan Tannen, et al. Gsa: a framework for rapid prototyping of smart alarm systems. In *Proceedings of the 1st ACM International Health Informatics Symposium*, pages 487–491. ACM, 2010. 94

[110] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012. 97

[111] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. 190, 199, 246

[112] Benjamin A Kohl, Sanjian Chen, Margaret Mullen-Fortino, and Insup Lee. Evaluation and enhancement of an intraoperative insulin infusion protocol via in-silico simulation. In *Healthcare Informatics (ICHI), 2013 IEEE International Conference on*, pages 307–316. IEEE, 2013. 237

[113] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003. 127

[114] Hermann Kopetz and Günter Grunsteidl. Ttp-a time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-*

23. Digest of Papers., The Twenty-Third International Symposium on, pages 524–533. IEEE, 1993. 192

[115] Dina A Krenzischek, Colleen J Dunwoody, Rosemary C Polomano, and James P Rathmell. Pharmacotherapy for acute pain: implications for practice. *Pain Management Nursing*, 9(1):22–32, 2008. 17

[116] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 226

[117] L Landro. In just a flash, simple surgery can turn deadly. *The Wall Street Journal*, 2012. 16

[118] Paul B Langevin, Vashti Hellein, Susan M Harms, William K Tharp, C Cheung-Seekit, and S Lampotang. Synchronization of radiograph film exposure with the inspiratory pause: Effect on the appearance of bedside chest radiographs in mechanically ventilated patients. *American journal of respiratory and critical care medicine*, 160(6):2067–2071, 1999. 13

[119] K.G. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on*, pages 203–210, 1988. 88

[120] Kim G Larsen and Axel Legay. Quantitative modal transition systems. In *Recent Trends in Algebraic Development Techniques*, pages 50–58. Springer, 2013. 89

[121] Kim G Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In *Programming Languages and Systems*, pages 64–79. Springer, 2007. 37, 88

[122] Brian R Larson, John Hatcliff, and Patrice Chalin. Open source patient-controlled analgesic pump requirements documentation. In *Proceedings of the 5th International Workshop on Software Engineering in Health Care*, pages 28–34. IEEE Press, 2013. 232

[123] Lucian L Leape. Reporting of adverse events. In *N Engl J Med*. Citeseer, 2002. 19

[124] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. 238

[125] Kathy Lesh, Sandy Weininger, Julian M Goldman, Bob Wilson, and Glenn Himes. Medical device interoperability-assessing the environment. In *hcmdss-mdpnp*, pages 3–12. IEEE, 2007. 12

[126] Nancy Leveson. A new accident model for engineering safer systems. *Safety science*, 42(4):237–270, 2004. 3

[127] Nancy Leveson. Perspective: The drawbacks in using the term'system of systems'. *Biomedical Instrumentation & Technology*, 47(2):115–118, 2013. 3

[128] Nancy Leveson, Nicolas Dulac, Karen Marais, and John Carroll. Moving beyond normal accidents and high reliability organizations: a systems approach to safety in complex systems. *Organization Studies*, 30(2-3):227–249, 2009. 3

[129] JP Lewis and Ulrich Neumann. Performance of java versus c++. *Computer Graphics and Immersive Technology Lab, University of Southern California (January 2003)*, 2004. 185

[130] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973. 156

[131] Ann S Lofsky. Turn your alarms on. *APSF Newsletter: The Official Journal of the Anesthesia Patient Safety Foundation*, 19(4):43, 2005. 13

[132] Shiwei Luan. *Modularized PCA Pump Design for an ICE-informed Medical Device Coordination Framework*. PhD thesis, Kansas State University, 2015. 232

[133] P. E. Macintyre. Safety and efficacy of patient-controlled analgesia. *British Journal of Anaesthesia*, 87(1):36–46, 2001. 18

[134] Adam Marcus. Once a tech fantasy, plug-and-play or edges closer to reality. *Anesthesiology News*, 33(1), 2007. 2

[135] Petr Matoušek. Tools for parametric verification. a comparison on a case study. *Journal of Universal Computer Science*, 10(10):1469–1494, 2004. 66

[136] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009. 128

[137] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow:

enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. 138

[138] Kenneth L McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003. 67

[139] Medical device "plug-and-play" interoperability program. http://mdpnp.org/, 2008. 3, 7, 31

[140] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. A common criteria based security requirements engineering process for the development of secure information systems. *Computer standards & interfaces*, 29(2):244–253, 2007. 199

[141] Eva L Menger. Dual-wavelength laser scalpel background of the invention, December 20 1988. US Patent 4,791,927. 15

[142] Mike Mitka. Fda lays out rules for regulating mobile medical apps. *JAMA*, 310(17):1783–1784, 2013. 4

[143] Aloysius K Mok. Fundamental design problems of distributed systems for the hard-real-time environment. 1983. 156

[144] James D Mooney. Bringing portability to the software process. 93

[145] José Eduardo Moreira, Samuel P Midkiff, and Manish Gupta. A comparison of java, c/c++, and fortran for numerical computing. *Antennas and Propagation Magazine, IEEE*, 40(5):102–105, 1998. 185

[146] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *Software, IEEE*, 30(3):50–57, 2013. 199

[147] M Niskanen, S Purhonen, V Koljonen, A Ronkainen, and E Hirvonen. Fatal inhalation injury caused by airway fire during tracheostomy. *Acta anaesthesiologica scandinavica*, 51(4):509–513, 2007. 15

[148] Thomas E Nolan and Hilton J Klein. Methods in vascular infusion biotechnology in research with rodents. *ILAR Journal*, 43(3):175–182, 2002. 215

[149] Miroslav Pajic, Rahul Mangharam, Oleg Sokolsky, David Arney, J Goldman, and Insup Lee. Model-driven safety analysis of closed-loop medical systems. *IEEE Transactions on Industrial Informatics*, 2013. 33, 219, 232, 237

[150] Aircraft Data Network Part. 7-avionics full duplex switched ethernet (afdx) network. *ARINC Specification 664p7*, 2005. 7

[151] Stephen D Patek, Sanjian Chen, Patrick Keith-Hynes, and Insup Lee. Distributed aspects of the artificial pancreas. In *Communication, Control, and Computing (Allerton), 2013 51st Annual Allerton Conference on*, pages 543–550. IEEE, 2013. 237

[152] JE Paul, M sawhney, WS Beattie, and RF McLean. Critical incidents amongst 10033 acute pain patients. *Canadian Journal of Anesthesiology*, 51:A22, 2004. 19

[153] Nathan W Pearlman, Gregory V Stiegmann, Virginia Vance, Lawrence W Norton, Reginald CW Bell, Robin Staerkel, Charles W Van Way, and Ed-

ward J Bartle. A prospective study of incisional time, blood loss, pain, and healing with carbon dioxide laser, scalpel, and electrosurgery. *Archives of surgery*, 126(8):1018–1020, 1991. 15

[154] Charles P Pfleeger and Shari Lawrence Pfleeger. *Security in computing*. Prentice Hall Professional Technical Reference, 2002. 199

[155] Jeffrey Plourde, David Arney, and Julian M Goldman. Openice: An open, interoperable platform for medical cyber-physical systems. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 221–221. IEEE, 2014. 3

[156] Patricia Quigley. F2761 and the integrated clinical environment. *Standardization News*, 37(5):20, 2009. 3

[157] John Rushby. Proof of Separability—A verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 352–367, Turin, Italy, April 1982. Springer-Verlag. 190

[158] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986). 191, 199

[159] John Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992. 199

[160] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000. 192, 199

[161] John Rushby and Paul S Miner. Modular certification. 2002. 34

[162] John M Rushby. Design and verification of secure systems. In *ACM SIGOPS Operating Systems Review*, volume 15, pages 12–21. ACM, 1981. 189, 199

[163] Steven R Salbu. Off-label use, prescription, and marketing of fda-approved drugs: An assessment of legislative and regulatory policy. *Fla. L. Rev.*, 51:181, 1999. 23

[164] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004. 6

[165] Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Krithi Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989. 168

[166] David S Sheinbein and Robert G Loeb. Laser surgery and fire hazards in ear, nose, and throat surgeries. *Anesthesiology clinics*, 28(3):485–496, 2010. 15

[167] Lee P Smith and Soham Roy. Device-related risk of airway fire in oropharyngeal surgery. *Otolaryngology–Head and Neck Surgery*, 139(2 suppl):P70–P70, 2008. 15

[168] Randall S Stafford. Regulating off-label drug userethinking the role of the fda. *New England Journal of Medicine*, 358(14):1427–1429, 2008. 23

[169] Christos Stergiou. *Schedulability Analysis and Verification of Real-Time Discrete-Event Systems*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2013. 124

[170] Nicholas Stevens, Ana Rosa Giannareas, Vanessa Kern, Adrian Viesca, Margaret Fortino-Mullen, Andrew King, and Insup Lee. Smart alarms: multivariate medical alarm integration for post cabg surgery patients. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, pages 533–542. ACM, 2012. 94

[171] Katherine Summers. *Principles and Methods of Testing Infusion Devices*, 2014, (accessed February 3, 2016). xiii, 215

[172] Compliance Today. New Medical Equipment Safety of Interoperability Standard AAMI/UL 2800 Being Developed. http://www.metlabs.com/blog/product-safety/, 2013. [Online; accessed 2016]. 3

[173] Eugene Y Vasserman, Krishna K Venkatasubramanian, Oleg Sokolsky, and Insup Lee. Security and interoperable-medical-device systems, part 2: Failures, consequences, and classification. *Security & Privacy, IEEE*, 10(6):70–73, 2012. 8

[174] Krishna K Venkatasubramanian, Eugene Y Vasserman, Oleg Sokolsky, and Insup Lee. Security and interoperable medical device systems: Part 1. *IEEE security & privacy*, 10(5):61, 2012. 8

[175] Steve Vestal. Metah programmer\'s manual. 1996. 96

[176] Steve Vestal. Metah support for real-time multi-processor avionics. In *Parallel and Distributed Real-Time Systems, 1997. Proceedings of the Joint Workshop on*, pages 11–21. IEEE, 1997. 123

[177] Steve Vestal and Pam Binns. Scheduling and communication in metah. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 194–200. IEEE, 1993. 123

[178] Clinique veterinaire Villeray-Papineau. *Laser surgery*, (accessed February 3, 2016). 15

[179] Kim J. Vicente, Karima Kada-Bekhaled, Gillian Hillel, Andrea Cassano, and Beverley A. Orser. Programming errors contribute to death from patient-controlled analgesia: case report and estimate of probability. *Canadian Journal of Anesthesiology*, 50(4):328–32, 2003. 18

[180] C.B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, pages 2.A.1–1–2.A.1–10, Oct 2007. 193

[181] Christopher B Watkins. Integrated modular avionics: Managing the allocation of shared intersystem resources. In *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, pages 1–12. IEEE, 2006. 247

[182] Carsten Weise and Dirk Lenzkes. Weak refinement for modal hybrid systems. In *Hybrid and Real-Time Systems*, pages 316–330. Springer, 1997. 38, 78, 79, 89, 91

[183] Carsten Weise and Dirk Lenzkes. Weak refinement for modal hybrid systems. In *Proceedings of the International Workshop on Hybrid and Real-Time Systems*, HART '97, pages 316–330, London, UK, UK, 1997. Springer-Verlag. 44, 78, 79, 82

[184] Susan F Whitehead and Julian M Goldman. Hospitals issue call for action on medical device interoperability. *Patient Safety & Quality Healthcare*, 6:1, 2009. 3

[185] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993. 115

[186] David W Woodruff. A quick guide to vent essentials. 2005. 202

[187] Qizhi Zhang and Weidong Zhang. Priority scheduling in switched industrial ethernet. In *American Control Conference, 2005. Proceedings of the 2005*, pages 3366–3370. IEEE, 2005. 157

[188] Jia Zou, Joshua Auerbach, David F Bacon, and Edward A Lee. Ptides on flexible task graph: real-time embedded systembuilding from theory to practice. In *ACM Sigplan Notices*, volume 44, pages 31–40. ACM, 2009. 124

[189] Jia Zou, Slobodan Matic, Edward A Lee, Thomas Huining Feng, and Patricia Derler. Execution strategies for ptides, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 77–86. IEEE, 2009. 124, 193