POP-PL: A Patient-Oriented Prescription Programming Language

SPENCER P. FLORENCE and BURKE FETSCHER, Northwestern University, Department of Electrical Engineering and Computer Science
MATTHEW FLATT, University of Utah, School of Computing
WILLIAM H. TEMPS, Northwestern University Feinberg School of Medicine, Department of Dermatology
VINCENT ST-AMOUR, Northwestern University, Department of Electrical Engineering and Computer Science
TINA KIGURADZE, Northwestern University Feinberg School of Medicine, Department of Dermatology
DENNIS P. WEST, Northwestern University Feinberg School of Medicine, Department of Dermatology & Northwestern University Feinberg School of Medicine, Department of Pediatrics
CHARLOTTE NIZNIK, Northwestern University Feinberg School of Medicine, Department of Obstetrics and Gynecology
PAUL R. YARNOLD, Optimal Data Analysis LLC
ROBERT BRUCE FINDLER, Northwestern University, Department of Electrical Engineering and Computer Science
STEVEN M. BELKNAP, Northwestern University Feinberg School of Medicine, Department of Dermatology & Northwestern University Feinberg School of Medicine, Department of Medicine

A medical prescription is a set of health care instructions that govern the plan of care for an individual patient, which may include orders for drug therapy, diet, clinical assessment, and laboratory testing. Clinicians have long used algorithmic thinking to describe and implement prescriptions but without the benefit of a formal programming language. Instead, medical algorithms are expressed using a natural language patois, flowcharts, or as structured data in an electronic medical record system. The lack of a prescription programming language inhibits expressiveness; results in prescriptions that are difficult to understand, hard to debug, and awkward to reuse; and increases the risk of fatal medical error.

This article reports on the design and evaluation of Patient-Oriented Prescription Programming Language (POP-PL), a domain-specific programming language designed for expressing prescriptions. The language is based around the idea that programs and humans have complementary strengths that, when combined properly, can make for safer, more accurate performance of prescriptions. Use of POP-PL facilitates automation of certain low-level vigilance tasks, freeing up human cognition for abstract thinking, compassion, and human communication.

This work is supported by the National Science Foundation.

Authors' addresses: S. P. Florence, B. Fetscher, M. Flatt, W. H. Temps, V. St-Amour, T. Kiguradze, D. P. West, C. Niznik, P. R. Yarnold, R. B. Findler, and S. M. Belknap; emails: {spencer.florence, burke.fetscher}@eecs.northwestern.edu, mflatt@cs.utah.edu, william.temps@northwestern.edu, stamourv@eecs.northwestern.edu, TinatinKiguradze2013@u.northwestern.edu, {dwest, c-niznik}@northwestern.edu, paul@planetyarnold.com, robby@eecs.northwestern.edu, sbelknap@northwestern.edu.

https://doi.org/10.1145/3210256

We implemented this language and evaluated its design attempting to write prescriptions in the new language and evaluated its usability by assessing whether clinicians can understand and modify prescriptions written in the language. We found that some medical prescriptions can be expressed in a formal domain-specific programming language, and we determined that medical professionals can understand and correctly modify programs written in POP-PL. We also discuss opportunities for refining and further developing POP-PL.

CCS Concepts: • **Software and its engineering → Domain specific languages**;

Additional Key Words and Phrases: DSL design, medical programming languages, medical prescriptions, empirical evaluation

ACM Reference format:
Spencer P. Florence, Burke Fetscher, Matthew Flatt, William H. Temps, Vincent St-Amour, Tina Kiguradze, Dennis P. West, Charlotte Niznik, Paul R. Yarnold, Robert Bruce Findler, and Steven M. Belknap. 2018. POP-PL: A Patient-Oriented Prescription Programming Language. *ACM Trans. Program. Lang. Syst.* 40, 3, Article 10 (July 2018), 37 pages.
https://doi.org/10.1145/3210256

1 PRESCRIBING PROGRAMS

Prescription-writing is effectively a parallel discipline that uses many constructs common to computer programming but without benefit of the extensive conceptual framework provided by computer science. Physicians and other prescribers write and repeatedly modify complex, algorithmic prescriptions, some the length of a journal article (Handelsman et al. 2011). These algorithms may be expressed in reference works or published clinical trial reports in the form of natural language or flow charts; remarkably the detailed written protocols used for conduct of clinical trials are often elided and are unavailable to the prescriber. Some parts of prescriptions are not directly stated but instead are delegated to standard operating procedures of institutionsor clinical guidelines of professional societies or are left to the judgment of individual clinicians, leaving much room for uncertainty. Nurses, pharmacists, physicians, medical technicians, patients, caretakers, and medical devices endeavor to perform the instructions in a prescription to the best of their understanding and ability. As prescriptions are written in natural language, prescription instructions unavoidably contain errors and ambiguities, cause miscommunication, and often specify or include vigilance tasks that exceed the capacity of unaided humans to perform reliably. Prescription bugs and performance errors may result in patient injury or death. Previous attempts to "computerize" prescriptions have floundered, because the algorithmic nature of prescriptions was unrecognized. Singh et al. (2009) have shown that use of structured data in Computerize Physician Order Entry (CPOE) often cannot clearly express the meaning of prescriptions; they conclude that improvements in the expression of prescriptions are needed to reduce error and improve health care safety.

In response to this important, widespread, and unmet need, we have designed, implemented, and tested a prototype prescription programming language. This Patient-Oriented Prescription Programming Language (POP-PL, pronounced "pop-pee-ell") leverages the power of software design and program development to help physicians express, model, test, debug, reuse, and iteratively refine prescriptions. POP-PL aims to express all of the instructions relating to wellness promotion, disease prevention, and medical care of a patient, including drug therapy, diet, activity, clinical evaluation, laboratory testing, medical imaging, and so on. By automating the monitoring and management of low-level health care tasks, programs written in POP-PL should reduce cognitive load on patients, caregivers, and clinicians, freeing up time and resources for communication and analysis.

POP-PL represents prescriptions as event-driven programs that interact with a system of actors to provide health care for patients—including patients themselves, caregivers, prescribers, other clinicians, clinics, hospitals, and, ultimately, entire health care systems. Based on our observations, clinicians[1] readily grasp this conceptualization of health care; indeed, this is how the clinician-researchers on our team think of their healthcare work. A POP-PL program views the health care system context as a generator of a stream of messages describing events and tasks related to patient care. A prescription responds to particular signals or signal patterns by issuing instructions to command, control, and coordinate relevant actors, including other prescriptions, medical personnel, and medical devices. POP-PL's prescription/program paradigm provides a means of coordinating complex interactions among actors (whether human or machine) that perform the essential operations of care (e.g., nurses, pharmacists, or infusion pumps).

The premise of this work is that **prescriptions are programs** and that such a perspective on the nature of prescriptions can mitigate medical error, even when the prescriptions are still expressed on paper (Belknap et al. 2008). Programs and programming languages potentially have much more to contribute, but only if there exists an adequate prescription programming language. Such a language must possess two key properties: It must be able to represent prescription tasks in a manner that human or machine actors can perform and that clinicians can understand and modify. This article focuses on creating a language that meets these two criteria.

To inform the construction of POP-PL, we have analyzed the structure and function of numerous existing English-language prescriptions and clinical protocols, and we have observed clinician teams while performing medical work in hospitals. This article reports our evaluation of the expressiveness and comprehensibility of prescriptions written in POP-PL.

The article includes a discussion of programs that have been translated to POP-PL and a survey assessing the ability of physicians, pharmacists, nurses, and other clinicians to interpret and modify prescriptions written in POP-PL. We found that prescriptions can be well represented in POP-PL and that prescribers and clinicians readily understand programs written in this prescribing language. We also present the concrete syntax of POP-PL, a reduction semantics, and some example prescriptions, including those used in our survey.

Section 2 covers related work. Section 3 provides some of the relevant medical background and motivation for addressing the medical error problem as well as our perspective on the nature of prescriptions. Section 4 explains the design principles and requirements of POP-PL. Section 5 provides some detail on how POP-PL meets these requirements. Section 6 works through an example prescription written in POP-PL. Section 7 discusses POP-PL's formal evaluation model. Section 8 describes how the language's concrete syntax maps to the model. Section 9 describes execution of POP-PL programs in more depth. Section 10 presents our analysis of POP-PL, including an evaluation of example prescriptions translated into POP-PL and an empirical study of how well prescribers can understand and express prescriptions written in the language. Section 11 covers anticipated future work, and Section 12 concludes.

2 RELATED WORK

This work is inspired by a number of different streams of work, originating both in the medical world and in computer science.

2.1 Algorithmic Medicine

Medical practitioners have long recognized the value of using algorithmic descriptions to codify mechanistic aspects of clinical practice. One of the most successful instances of algorithmic

[1]Throughout this article, the word "clinician" refers to the most highly trained medical workers present in hospitals, namely physicians, nurses, and pharmacists.

medicine is the Ottawa Ankle Rules (Stiell et al. 1994), a sensitive algorithm for identifying which injuries are unlikely to involve a fracture of the ankle or foot, and therefore do not require further evaluation with X-ray imaging. The Ottawa Ankle Rules expresses the essential elements and logic of the clinical evaluation of an injured ankle as a simple algorithm. It has led to better outcomes at lower cost than ankle radiography and is now widely used in routine clinical practice. There have been entire textbooks that collect medical algorithms and explain how to use them (Healey and Jacobson 1994; Mushlin and Greene 2010). These algorithms are typically expressed using flow-charts and treelike decision diagrams and thus fail to bring the full power of algorithmic expression that programming languages offer. The work described in this article is the logical next step: bringing programming language know-how to the design of a language for the clear expression of medical algorithms.

2.2 Computerized Physician Order Entry

Computers and software have long been proposed as a means of reducing medical error. The American Recovery and Reinvestment Act of 2009 allocated more than $19 billion to support adoption of health information technology; the national investment was estimated at $115 billion distributed over 15 years (Hillestad et al. 2005). These efforts include placing computers at the point of care, providing Electronic Medical Record (EMR) systems to manage and store data, using Computerized Prescriber Order Entry (CPOE) or ePrescribing systems for handling prescriptions and using Clinical Decision Support (CDS) systems to automatically detect potential errors and alert clinicians.

There is scant high-quality evidence that current attempts to computerize health care have resulted in significant improvement in patient safety in clinical practice (Landrigan et al. 2010). Some research studies have shown that computerization of health care can reduce medical error, but there is often no evaluation of the effect on patient outcomes. When clinically meaningful outcomes are evaluated, the demonstrated effect on reducing severe or fatal patient injury is may be small or nil (Jones et al. 2014). While CPOE does reduce or eliminate some kinds of error, like illegible handwriting, but introduces new types of medical error. In one report, CPOE systems caused 22 new types of medical error, including fragmented display of drugs, inventory display mistaken for dose guidelines, duplicate and incompatible orders, and inflexible formats (Koppel et al. 2005). Because prescribers find CPOE to be limited in expressiveness, they often include free-text instructions that cannot be represented in the structured forms of CPOE systems. A large study of CPOE systems found that 1% of prescriptions contained free-text instructions that were inconsistent with the contents of the structured data fields, and 20% of these inconsistencies risked moderate or severe patient injury (Singh et al. 2009).

2.3 Process Modeling Languages

There already exist several programming languages that attempted to model and automate processes, several of them in the medical context. For a survey of the area, we refer the interested reader to van der Aalst et al. (2003).

POP-PL's main contribution to this body of work is the creation of a patient-oriented language that clinicians can use directly to express and modify clinical algorithms so as to address the needs of the patient.

The Little-JIL (Chen et al. 2006) process definition language also aims to represent medical processes with a graphical language. Mertens et al. (2012) shows that such representation of medical processes is possible and can result in clearer communication and a reduction in errors. Little-JIL emphasizes modeling hospital processes (and processes in general) to use static analysis to find potential errors and prove properties about the prescription. This is orthogonal with POP-PL's attempt to coordinate the actual execution of these prescriptions. In addition Little-JIL takes a process-oriented view of a prescription. It attempts to model and understand prescriptions as the central—immutable—object of study. Patients flow through the prescription, instead of the prescription being adapted to the patient. This helps enable Little-JIL's static analysis. Little-JIL also stands out in that it is not specific to the medical domain; it attempts to be a generic process definition language.

Another medical language is TNest (Combi et al. 2012), a graphical workflow language for modeling temporal data-centric medical processes. It is designed to express the temporal constrains that can exist between concurrent tasks in the hospital. It does not focus on error detection, nor is it patient-oriented, but rather it focuses on a particular technical challenge: the expression and scheduling of medical processes with data and temporal dependencies. This is one area where POP-PL is weak. For instance, the heparin program in Figure 2 does not express that the lowering of the heparin dosage on line 24 must occur before the restarting of the heparin on line 24. This dependency is also not expressed in the original protocol; its existence is inferred.

Asburu (Sharar et al. 1998) and GLARE (Molino et al. 2006) are languages designed to represent clinical protocols using AI techniques. They are meant to generate plans from abstract specifications and ontologies. The GLIF language (Peleg et al. 2000) attempts to provide a process definition language that can standardize guidelines between hospitals. It does not attempt to tackle the problem of medical error directly. The Eon (Tu and Musen 1999) language attempts to model a much broader range of clinical protocols, including those of clinical trials, and attempts to tackle the problem of modularity. Ruffolo et al. (2005) attempts to monitor clinical processes during execution and suggest solutions to problems using data-mining rather than controlling processes directly.

iTasks (Jansen et al. 2010) is a generic task management language, expressed as an embedded DSL in the language Clean (Brus et al. 1987). iTasks distinguishes itself from other workflow languages by focusing on dynamic control flow and better abstractions, as opposed to the static control flow of the languages discussed so far. iTasks uses composable procedures to represent tasks for external entities. In contrast, POP-PL directly passes instructions and information between the prescription and the actors in the system.

The iALARM (Kilmov and Shahar 2013) language attempts to describe and respond to patterns of events in the hospital. POP-PL has a similar, though less expressive, querying language.

The Declare framework (van der Aalst et al. 2009) aims to provide flexibility in workflow execution. In the medical context this language is especially useful, because it allows the programmer to specify what must, what must not, and what may happen. This approach would be very useful for implementing orders that, e.g., prevent use of penicillin-type drugs in a patient with severe penicillin allergy or require anticoagulation in a patient at risk for thrombosis. However Declare suffers from readability problems in complex workflows with many interacting constraints.

2.4 Design

Pane et al. (2002) call for a human-centered approach to language design, using HCI principles. They demonstrate this by designing a language, HANDS, with a focus on the principle of closeness of mapping between the programmers envisioned solution to programming problems and their actual implementation (Green and Petre 1996).

POP-PL's design requirements, specifically the consideration of the interaction with other components of the hospital system and the errors that could arise from the interaction, was inspired by Furnas (2000)'s mosaic of responsive adaptive systems (MoRAS) perspective. However, we did not use the full design framework.

POP-PL is a clear example of end-user programming, a phrase attributed to Nardi (1993). In her definition, end-user programmers are not "not 'casual,' 'novice,' or 'naive' users." They are simply people who need to program but are not, and have no reason to become, professional programmers. This perspective clearly fits our notion of clinicians expressing computation in prescriptions. See Ko et al. (2011) for a recent survey of this area.

2.5 Evaluation of Domain-Specific Language Design

There is a long history of empirical evaluations of language design. These studies date back to the study of Psychology of Programming Languages. It includes studies such as Sime et al. (1972), which directly compares different kinds of conditionals in "micro-languages" designed specifically for the evaluation, and Green and Petre (1992), which compares an existing visual programming language to a custom text-based language. These studies tend to focus on evaluating usability by giving subjects programming tasks and analyzing aspects of the resulting programs or the process of creating the programs.

This plethora of usability studies eventually led to the Cognitive Dimensions of Notations framework (Green 1989), which attempts to replace these studies with a generalizable framework. This framework has received several extensions (Blackwell et al. 2001; Green and Petre 1996) and standard evaluation techniques (Blackwell and Green 2000). Notable uses of this framework include LabVIEW and Prograph (Green and Petre 1996), Vega (Satyanarayan et al. 2014), and HANDS (Pane et al. 2002). While it does not provide a framework to tell if a domain-specific language meets its goals directly, Cognitive Dimensions does provide a framework for classifying and comparing languages.

This kind of non-experimental usability study is often used to evaluate domain-specific languages. For example Little-JIL was evaluated using unstructured interviews in which participants helped modify a process for chemotherapy delivery to determine if a program in the language could find ambiguities and safety violations in the protocol (Mertens et al. 2012). Little et al. (2010) examine several programs written in TurKit, a language for programming Mechanical Turk. While not providing strong statistical evidence for the claims about the language's efficacy, these studies do give a good intuitive understanding of where the language succeeds and fails. Pane et al. (2002) compared several programming tasks between several variants of the HANDS language. The evaluation of POP-PL presented in this article is based on a similar non-experimental usability design.

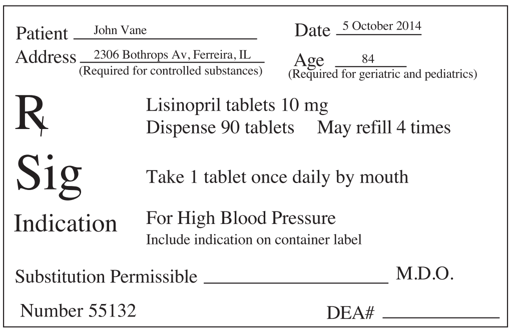
2.6 Actors and Time

POP-PL uses a previously established paradigm in medical software by modeling hospitals as actor systems. For example, Belknap (1991) provides a system for simulating hospital drug therapy by modeling a patient as a set of functions that are affected by interactions with the outside world. In addition, the Eon system (Tu and Musen 1999) conceptualizes the hospital "as consisting of multiple agents—health care providers, patients, and decision-support systems—interacting with each other at different points along a temporal continuum."

POP-PL's handling of time is similar to how Esterel (Boussinot and Simone 1991) handles time: each unit of time being a discrete message, although POP-PL's approach is less principled. In addition the way POP-PL handles synchronicity and determinism—receiving one message, then running the program to quiescence, and then handling the next message—is similar to both Esterel and Marketplace (Garnock-Jones et al. 2014), as well as Hewitt et al. (1973)'s original actor model. The way POP-PL delivers messages to relevant parties is similar to the way message delivery in Callsen and Agha (1994)'s Actorspaces works.

3 THE MEDICAL DOMAIN

This section provides context on the nature of prescriptions and the difficulty of preventing and mitigating medical error.



Patient John Vane Date 5 October 2014
Address 2306 Bothrops Av, Ferreira, IL Age 84
(Required for controlled substances) (Required for geriatric and pediatrics)

℞ Lisinopril tablets 10 mg
Dispense 90 tablets May refill 4 times

Sig Take 1 tablet once daily by mouth

Indication For High Blood Pressure
Include indication on container label

Substitution Permissible ____________ M.D.O.

Number 55132 DEA# ____________

Fig. 1. Lisinopril prescription.

3.1 What is a Prescription?

A medical prescription is a program whose instructions govern the plan of care for an individual patient (Belknap et al. 2008). The complexity of prescriptions may not be immediately evident to the casual observer. Even apparently simple prescriptions are more than just lists of drugs and dosages. They instead encompass all the instructions and logic that a prescriber deploys to safely and effectively coordinate the care of a particular problem. Most medical problems involve treatments that require control loops, clinical assessments, therapeutic monitoring for efficacy and toxicity, and one or more interventions; many patients have multiple medical problems. The code for coordinating each medical treatment typically includes both explicit and implicit instructions.

Consider the medical form in Figure 1. This is an order for a prescription that includes instructions for the pharmacist after the "℞"and for the patient after the "Sig." However, this does not constitute the full prescription. In addition to the text of the note, there are usually additional instructions—either explicit or implicit, written or verbal—for the patient, caregiver, pharmacist, nurse, dietician, and other clinicians. Examples of these instructions include the following:

— For the patient: "Check blood pressure daily, contact physician if blood pressure is above or below target range."; "Follow DASH2 Diet."
— If the patient is a woman: "Avoid pregnancy while taking lisinopril, as this drug can cause serious birth defects."
— And for prescribers and pharmacists: "Prevent prescription of drugs known to have harmful interactions with lisinopril."

So a prescription comprises *the entire set of health care instructions* that are in place for a particular patient, including complex, contingent, iterative instructions for how to manage the patient; how to modify drug dosing and sequencing; criteria specifying when to notify a doctor; scheduling of laboratory tests; monitoring of relevant biomarkers of treatment efficacy; and management of potential adverse effects. These instructions may apply to the prescriber, nurse, pharmacist, or other clinicians, as well as to the patient or caregiver. Some instructions may be refused, conflict with other instructions, or be impossible because of missing resources. Currently, these instructions are expressed as free-text pseudocode, as pre-printed fill-in-the-blank style order sets, and (imperfectly) as structured data in CPOE systems.

3.2 Why is Reducing Error Important?

Over the past half-century, medical research has consistently identified medical error as a major major health care risk and thereby a serious impediment to the practice of patient-oriented[2] medicine (Gaba et al. 1987; Leape 1994; Schimmel 1964). Errors are surprisingly common in medicine; among hospitalized patients, errors occur at the rate of one error per patient per day (Committee on Identifying and Preventing Medication Errors 2007). Despite extensive and expensive efforts to improve patient safety, medical error continues to contribute to roughly one-sixth of all deaths in the United States—approximately 440,000 deaths per year (James 2013). Unfortunately, Landrigan et al. (2010) found that there had been little evidence of improvement in the rate of medical errors over the previous decade.

3.3 Why Target Prescriptions?

The form, severity, and origin of individual medical errors is varied. Leape et al. (1993) break down medical errors into 14 different categories ranging from inadequate monitoring to equipment failures. So why, given such a spectrum of errors, are prescriptions the correct place to intervene?

While the manifestation of error is varied, every action that directly affects a patient must go through a prescription. This puts prescriptions on the critical path between the intent behind how to care for a patient and the actuation of that care. This makes prescriptions a natural place to augment the medical system to reduce medical error.

4 THE DESIGN OF POP-PL

Clinicians and software have complementary and synergistic strengths. Clinicians are capable of obtaining and analyzing patient narratives; making and testing hypotheses about diagnoses; applying general principles and mental models of human physiology, pathology, and pharmacology to plan a course of treatment; and communicating in natural language with patients, caregivers, and other clinicians. In contrast, software can reliably perform repetitive, tedious tasks such as searching large datasets, continuously monitoring for out-of-range values, precisely measuring time intervals, task auditing, and general bookkeeping. Safe, effective medical care requires that many small tasks be issued, scheduled, and performed. Individually, these tasks are usually not difficult. Problems arise because there are many tasks for each patient, multiple patients for each clinician, and competition among these tasks and actors for constrained resources. Also, many simple tasks are vigilance tasks, which are beyond human capacity to perform flawlessly over extended periods of time. Software systems excel at this sort of banal task management. The precise set of tasks may vary in small or large ways among patients. Even among patients with similar conditions, the most effective treatment for different patients may diverge in a process of trial and error, as clinicians observe and adjust the plan of care. Investigating and understanding the variations in patients' lives and their conditions and modifying the set of tasks to handle these variations is a natural fit for a clinician's skills.

Based on this understanding of health care delivery and our clinical experience, we have derived three primary design principles. First, POP-PL must be able to accurately represent the control flow of prescriptions in a machine executable manner. Any design decision in the language must help represent how prescriptions perform and how this performance integrates into both the patient's experience and the workflow of the clinician's practice. This allows POP-PL to act as a helpful teammate, whose purpose is to help remind clinicians what they wanted to happen and when.

[2]Patient-oriented medicine has the goal of improving or maintaining health while being responsive to the patient's needs and values (Kindig 1971). This contrasts with task-centric, disease-centric, or provider-centric orientations. These perspectives sometimes conflict with the patient-oriented perspective.

Second, POP-PL must be a language clinicians can use and understand. The behavior of programs should be clear, and both using and modifying POP-PL programs must fit into clinicians' workflows. Clinicians already have highly accurate mental models of medical algorithms and desired prescription behavior. What they lack is a suitable means of expressing these mental models. To be understandable, POP-PL must allow clinicians to accurately and efficiently express and communicate these mental models.

Finally, POP-PL must enhance the human/computer synergy with a singular goal: the mitigation of medical error. Each design requirement must consider how it can reduce or mitigate medical error. Two kinds of error must be considered. The first are the existing kinds of error that already make hospitals dangerous. The second are errors that use of POP-PL may introduce. Hospitals comprise a large number of complex interacting systems. Adding a new component can have far-reaching effects on such systems. We must consider what these effects might be, if they could introduce new kinds of errors, and, in our language design efforts, work to prevent or mitigate these errors.

From these principles we have derived several specific design requirements for POP-PL. Each requirement is necessary either to represent prescriptions or to make language usable to clinicians. Each requirement is paired with those error modes that it aims to mitigate. For existing categories of error, specific instances of each were observed by the team that developed a prescription for opioids (Belknap et al. 2008) and during our observation of clinical teams.

Representing Connections and Forgotten Monitoring. The language must be able to represent connections between portions of a medical protocol. For example, it must provide an easy-to-use mechanism to build abstractions over common patterns of orders and logically bind monitoring of drug effects to decisions about drug administration.

The associated error category is forgotten monitoring. Safe administration of drugs typically requires commensurate monitoring. The output of the monitoring is often relevant to decision-making about therapy, including initiation or discontinuation or dose adjustment of drug therapy or initiation of other corrective treatments. For example, diabetic patients will often check their blood glucose and adjust their insulin dose based on the result. However, the relevant monitoring instructions, diet plan, and drug orders usually exist independently. Monitoring orders are typically not explicitly linked to the drug(s) whose effect(s) they are monitoring, even when the instructions are intended by the prescriber to be conceptually linked together. Accordingly, logically related subroutines may be decoupled, creating hazards and leading to unintended outcomes. For example, we are aware of instances where a patient with diabetes mellitus is receiving insulin injections, blood glucose monitoring, and continuous enteral tube feedings. The tube becomes obstructed or gets dislodged at the same time that glucose monitoring is omitted, while insulin injections continue. The outcome can be (and has been) fatal hypoglycemia due to inappropriate insulin administration in the setting of inadequate caloric intake. The missing value problem is common and potentially fatal yet particularly difficult to address with current systems.

Representing Questions and Alarm Fatigue. The language must be able to express physicians' requests for complex information about the patient's condition that are necessary to drive the patient's care. These questions range from checking the specific value of a lab test to whether the patient's treatment has reached or deviated from some therapeutic threshold to the detection of a dangerous adverse drug effect.

One of the most pernicious problems with error detection is the problem of alarm fatigue. The abundance of medical devices and automated alert systems in a hospital produce a cacophony of monitoring noise, as false alarms are common. Cvach (2012) found that clinicians may have to deal with as many as 700 monitor alarms per patient per day. A monitor may accurately detect a serious problem but fail to communicate this problem in a way that elicits an effective mitigating response from the people and systems involved in the patient's care. A language that cannot sufficiently gather and summarize information relevant to care would require a large amount of human intervention to guide patient care, increasing the number of alerts and alarms clinicians must sift through. On the flip side, a language with an adequately powerful query facility may help reduce errors caused by missed alerts by expressing complex yet common failure conditions and their commensurate responses programmatically.

Handling Common Situations and Delayed Reactions. The language must be able to specify how the machines and people are to react to the variety of states that may occur. These can range from reacting to parts of the normal operation of prescription (e.g., a diabetic patient receiving a meal), to dangerous situations (e.g., a diabetic patient becoming hypoglycemic).

The associated error category is a delayed response to a dire problem. With some prescriptions, even when monitoring is ordered and dangerous situations are detected, transmitted, noticed, and understood, life-saving measures are delayed or not taken. In some cases, the active prescription does not contain enough detail regarding the conditions that ought to trigger the mitigation (e.g., threshold values) or does not adequately specify how the mitigation instructions are to be performed (e.g., antidote dose). In other cases, the recipient lacks the training or authorization to perform the life-saving instructions. For example, if the nurse is not provided with precise criteria as to what conditions ought to trigger the use of an antidote or is not instructed as to the appropriate dose of the antidote to give the patient, then it becomes difficult to perform even this seemingly simple lifesaving intervention correctly. In such cases, the nurse must contact the prescriber, which may result in a dangerous delay. By expressing the criteria that ought to trigger the desired response as well as the details necessary to perform the response in a program, the computer can issue the correct tasks immediately and raise alarms if they are not performed expeditiously.

Unambiguity and Improper Task Management. The language must be unambiguous when read by humans. This lack of ambiguity must extend to both the programs and their outputs.

The associated error category comprises mistakes and delays in care caused by ambiguities in existing prescriptions. Having a prescription that monitors events and issues tasks will help ensure that the correct tasks happen at the correct time but will not fully solve the ambiguity problem. When nurses receive tasks that do not match their understanding of the plan for patient care, they often will delay performance of that task until obtaining confirmation from the prescriber. Such a response is considered necessary to safely care for the patient. Thus, to avoid unnecessary communication overhead and corresponding delays in care, the language must be unambiguous to clinicians.

Predictable Behavior and Reliance on the Machine. The language must express a model of the prescription that allows a clinician to predict the behavior of the prescription. This goes beyond the requirement of **unambiguity** in that the language must provide mechanisms to help explain portions of a prescription that are inherently confusing or complex.

The associated error category is an over-reliance on the computer system. All computer systems experience faults. Today's medical systems allow for paper backup so that hospitals can still function when computers are unavailable. Even when the IT systems are functioning, a pause to check the computer system may be overly burdensome in situations that require a fast reaction. Therefore, medical staff must be able to function in the absence of the computer system. This means that the medical staff should be able to execute a prescription/program just by reading it on paper.

Modifiability and Inflexibility. The language must allow for clinicians to modify existing protocols to suit the needs of their patients. Physicians frequently reuse pre-written and pre-validated prescriptions, modifying them for the specific needs of their patient. As such, we must focus on creating a language that is accessible enough to allow the average clinician to make patient-oriented modifications as needed.

The associated error category is a hospital system that cannot adapt to the needs of a specific patient. A patient on a pain management protocol who is addicted to opioids will likely need a higher starting dose to handle their higher tolerance; a patient on a blood pressure protocol might have their target blood pressure range change because of concerns about kidney function. If clinicians cannot freely make these changes, then they cannot adequately treat these patients.

While these are not a minimal set of non-functional requirements (Roman 1985) for POP-PL, they are a starting point for the design of the language. The requirements were elicited via extended observation of clinical teams in hospital and analysis of existing prescriptions.

5 MEETING THE REQUIREMENTS

This section gives an overview of the design of POP-PL from the perspective of the requirements discussed in the previous section. In this section and throughout the rest of the article, we use boldface to call attention to specific requirements.

POP-PL models the hospital as a collection of actors. To POP-PL, each entity in the clinical setting (prescriptions, nurses, pharmacists, patients, programmable infusion pumps, etc.) is an actor. Each actor subscribes and publishes messages that are relevant to their own role for each of the patients they have been assigned. All messages and actions are recorded in a log, which describes the task and event history of clinical care for a cohort of patients. In principle, the log is a time-ordered list of all events that have occurred and task requests that have been issued with respect to each individual patient's care, including information about when events occurred, plus other event-specific data.

A prescription consists of a set of handlers that react to the addition of entries to the log, by looking at the most recent message or querying the log for complex information. As such, an instance of a POP-PL program "runs" by launching new actors into the clinical network. These prescription actors issue tasks to relevant caretakers of their patients.

The message log gives us the tools to **represent questions** as patterns of events in the log. The reactive model gives us the tools to **handle common situations**. Each handler, which is analogous to an individual order in current prescribing systems, receives answers to questions presented. Each handler issues messages to coordinate reacting to the new information.

The reactive model matches prescribers' mental model of how prescriptions execute, which makes the prescriptions more **predictable**. The syntax of POP-PL also aids in meeting this requirement by having an explicit notation for examples and test cases in the language. These help explain complicated portions of prescriptions.

Based on the HCI principle of closeness of mapping between the programming language and the language of problem domain (Green and Petre 1996; Pane et al. 2001), the syntax of POP-PL is designed to look as much like existing article prescriptions as possible. We choose this familiar syntax to make the language more readable, **modifiable**, and **unambiguous**.

The syntax also **represents connections** by spatial proximity in the written code. Medication orders and their monitoring orders belong to the same protocol and are part of the same file. This presents the orders in a manner that is more intuitive for clinicians and helps to prevent medication orders, therapeutic interventions, and relevant monitoring from being decoupled.

6 AN EXAMPLE POP-PL PROGRAM

This section presents a program for administering an anticoagulant called heparin in a hospital setting. The program is a partial translation of the Washington Adventist Hospital (2009) protocol for continuous intravenous administration of heparin. Heparin is given to patients to prevent or treat thrombosis, the formation of a blood clot inside a blood vessel, which may impede blood flow and cause organ damage or dysfunction. Heparin dosing is challenging, because both the duration and intensity of heparin's anticoagulant effect change disproportionately and unpredictably with drug dose. If the heparin dose is too low, then treatment may fail, and the patient may have a catastrophic event, such as an embolism to the brain or lungs; too high, and the patient may experience life-threatening hemorrhage. To maintain the correct, individualized, amount of anticoagulation, the protocol requires frequent monitoring of heparin's anticoagulant effect and corresponding adjustments of its dosing rate.

Figure 2 contains the example POP-PL program. The first line of the program tells the Racket runtime (Flatt and PLT 2010) that our program is written in the POP-PL language. The third line tells POP-PL to load a library with definitions for the communication protocols of the hospital where this protocol is used.

When a prescriber starts this program, it initially—lines 5 to 7—sends a message to generate a task for the nurse on duty. The message requests a one-time intravenous starting dose of heparin, called a bolus, to increase the concentration of heparin in the patient's blood up to the desired levels. It then sends a message to start a continuous intravenous infusion of heparin at 18 units[3] per kilogram of body weight per hour to maintain the heparin concentration in the body at the desired level.

Intravenous pumps are configured to dispense fluids in units of volume per time, generally milliliters per hour. Converting from units per kilogram per hour to milliliters per hour is a two-step process. First the units per hour is calculated from the patients body weight. This computation is currently done by the physician and double checked by the pharmacist before a solution is prepared. Second the units per hour is converted to milliliters per hour based on the particular form of heparin being used, namely the concentration of the heparin in solution.[4] The error rate for initiation of heparin dosing is typically 2 errors per 1,000 doses ordered (Harder et al. 2005); this may seem low but is still of concern as the consequences of a heparin dosing error may be fatal. POP-PL could calculate the pump setting with access to the electronic medical record.

The next section of the prescription, labeled `infusion:`, is a handler that continuously modifies the heparin dosage by either changing the rate of the infusion, giving another bolus, or stopping the infusion altogether. It does so based on the value of a laboratory assay called the activated partial thromboplastin time (aPTT). This aPTT measurement is the number of seconds it takes for blood to clot under conditions specified by the aPTT laboratory assay. Line 10 reacts to new aPTT messages, running lines 11 through 24 when a new aPTT value is present. The identifier `aPTTResult` is an externally referent identifier, bound in the library required on line 3 and encoding information about messages containing aPTT values. Using it with `whenever new` tells the handler to wait for a message indicating a new `aPTTResult` and to bind its payload to the variable `aPTT`. A conditional dispatch is then performed based on the value of `aPTT`. The test expressions are on the left of the pipes, and their corresponding bodies are to the right. For instance, if `aPTT` is between `101` and `123`, then the program asks the nurse to decrease the heparin infusion by 1 unit/kg body weight per hour. The prescription's target range for aPTT values is between `59` and `101`; when

[3]A "unit" is a standard measurement of heparin and is 0.002mg. This is the amount of heparin needed to keep 1ml of cat's blood fluid for 24 hours at 0°C.

[4]For example, consider a patient weighing `85kg`, and a solution concentration of `100 units/mL`. The physician would calculate `18 units/kg/hour x 85 kg = 1,530 units/hr`. The pharmacist would then prepare a solution of `100 units/mL`. The nurse would then calculate `1,530 units/hr ÷ 100 units/mL = 15.3mL/hr`. The pump likely will not accept fractional units, resulting in a final setting of `15 mL/hr`.

results within this range are reported, then the program issues no heparin dose adjustments. The comment on line 17 documents the gap in the conditional.

Finally, lines 26 through 28 schedule aPTT readings. It requests that the nurse check the aPTT value daily if the last two aPTT readings were in the target range, and every six hours otherwise. The `aPTTResult` after the `whenever` tells POP-PL to query the log for aPTT results, the `in range` and `outside of` tell POP-PL what range of values to look at, and the `x2`'s requires that the event happened twice. There is some syntactic trickery[5] here: If there are fewer than two aPTT values present, then the 6-hour case is run. This is because

```
aPTTResult outside of 59 to 101, x2
```

on line 27 is not equivalent to

```
aPTTResult not(in range 59 to 101), x2
```

but rather equivalent to

```
not(aPTTResult in range 59 to 101, x2)
```

that is, the negation applies to the entire query, not just the range portion.

The `every` form takes two expressions: a time frame and a message. It queries the log to see if that message has been sent within its time frame and, if not, sends it. So line 27 reads in English as "If there has not been an aPTT value read in 6 hours, and either the last two aPTT values are not between 59 and 101 seconds or there are fewer than two aPTT values, then check the aPTT value."

In our prototype implementation, when a program is run it starts a simulation of prescription which can be interacted with via a read-eval-print loop (REPL). The REPL first prints out the messages that would be sent from the prescription in a deployed version. It then prints a prompt where the user can submit messages to simulate responses from the world. This simulation does not display timestamps with the messages, as it has no real notion of time. Instead, time is advanced manually with a special command.

When the program in Figure 2 is run, it sends the messages:

```
[givebolus 80 units/kg HEParin I.V.]
[start 18 units/kg/hour HEParin I.V.]
[checkaptt]
```

which tell the nurse to start the IV and give the initial bolus and ask for the initial aPTT reading.[6] We can then enter messages at the REPL to see how the program responds. For example, an aPTT response of 46s is sent by typing the text following the ">":

```
> aPTTResult 46 seconds
[givebolus 40 units/kg HEParin I.V.]
[increase HEParin 1 unit/kg/hour]
```

The program responds by increasing the heparin drip and giving another bolus. A little over 6 hours later, the program requests another aPTT test:

```
> wait 6 hours
[checkaptt]
```

[5]This trickery backfired, violating the requirement of **predictability**. We revisit the issue in Section 10.
[6]The outgoing messages may not match the case of what appears in the program, because POP-PL is case insensitive. See Section 8 for why this is beneficial.

```
 1  #lang pop-pl
 2
 3  used by JessieBrownVA
 4
 5  initially
 6     giveBolus 80 units/kg of: HEParin by: iv
 7     start 18 units/kg/hour of: HEParin by: iv
 8
 9  infusion:
10    whenever new aPTTResult
11      aPTT < 45            | giveBolus 80 units/kg of: HEParin by: iv
12                           | increase HEParin by: 3 units/kg/hour
13
14      aPTT in 45 to 59     | giveBolus 40 units/kg of: HEParin by: iv
15                           | increase HEParin by: 1 unit/kg/hour
16
17   // aPTT in 59 to 101    | Continue current HEParin dose
18
19      aPTT in 101 to 123   | decrease HEParin by: 1 unit/kg/hour
20
21      aPTT > 123           | hold HEParin
22                           | after 1 hour
23                           |     restart HEParin
24                           |     decrease HEParin by: 3 units/kg/hour
25
26  aPTTChecking:
27    every 6 hours checkaPTT whenever aPTTResult outside of 59 to 101, x2
28    every 24 hours checkaPTT whenever aPTTResult in range 59 to 101, x2
29
30  --- Tests ---
31
32  [giveBolus 80 units/kg of: HEParin by: iv]
33  [start 18 units/kg/hour of: HEParin by: iv]
34  [checkaPTT]
35
36  > aPTTResult 240
37  [hold HEParin]
```

Fig. 2. Example of a POP-PL program.

Lines 30 and after contain unit tests for the program. They are written just like REPL interactions; a message in brackets denotes what should be the outgoing message, and a message preceded by a ">" is a message sent to the prescription. Unlike REPL interactions, both portions are inputs to the program; they are run and the results are checked. If the outputs are other than expected, then the program prints out a message saying that the test cases failed. In addition to checking correctness, the tests work towards keeping programs in the language **predictable**. When portions of the program are complex the test cases can act as clarifying examples.

7 A FORMAL MODEL FOR EVALUATION

The interactions between POP-PL's handlers and how they affect the program's future is not present in most languages. Thus, we define an operational semantics to make the handlers' interactions and behavior mathematically precise. To formally model how POP-PL programs behave, we define an evaluator that handles one new message at a time. This evaluator takes the current event log (including the new message), and an actor, which is a list of named handlers. These

```
  e ::= (add name e) | (remove name) | (send e)
      | (λ (x ...) e) | (e e ...) | (o e ...)
      | x | m | log | void
      | (begin e e) | (if0 e e e)
log ::= (list m ...)
  o ::= time-of | most-recent | ...
      | message-type-is | message-payload | make-hold-message
      | time-passed? | make-restart-message | make-start-message
```

Fig. 3. POP-PL model syntax.

handlers can send messages to the outside world and add or remove handlers from the actor. This produces a set of outgoing messages and a new actor.

This section presents the formal model for this evaluator. It then extends the model with a DSL for querying complex information from the event log. The abstract syntax for POP-PL (shown in Figure 3) is the λ-calculus with sequencing plus three new forms: two for adding and removing handlers from the state, and one for emitting messages. In addition, it has three primitive data types: messages (m), logs, and void. We use the λ-calculus here not because POP-PL is higher order but because the λ-calculus is a convenient lingua franca for programming language models.

It may not be readily apparent how the model in this section maps to the program in Figure 2. We ask the reader's indulgence here. The next sections describe how the language's concrete syntax compiles to this abstract model.

7.1 Language Basics

A log is a complete representation of the history of events. It has a chronologically ordered sequence of messages plus the time these messages were sent. The language has primitive functions for getting the most recent message from a log and getting the time of that message and other similar functions (not mentioned here). Logs as first class immutable values allow closures to capture logs and compare them against logs they receive later. This is useful for **representing questions** about changes in patient's state.

Messages are arbitrary network-serializable data. For the simplicity of the model, we leave messages opaque, but, in our implementation, messages have structure and a human-readable form.

Time is represented with heartbeat messages. Each heartbeat means that a time interval has passed. Programs use these heartbeat messages to react to changes in time the same way they react to other events. While this could be done with a more elegant system, like an alarm service, heartbeats are used for simplicity.

7.2 The Evaluator

The evaluator—whose type is at the top of Figure 4—takes in a set of handlers and the current log. Each handler is a named function of one argument: the log to handle. The evaluator invokes each handler with the current log. When a message to which the prescription is subscribed appears, the evaluator is invoked with the new state and a log containing the new message.

Figure 4 extends POP-PL's syntax with the machine state (s) and evaluation contexts (E and S). The machine has three registers: the current expression being evaluated, the set of handlers being computed for the next state, and the outgoing messages. Given H_0 = (h_1 h_2 ...), an initial, non-empty set of handlers, and *log*, the current view of history, EVAL starts computation in the machine state ⟨(h_1 *log*), (h_2 ...), ()⟩.

$$\textsc{eval} : H \times log \rightarrow H \times (m\ \ldots)$$

$$\begin{array}{lcl}
s & ::= & \langle e,\ H,\ (m\ \ldots)\rangle \\
H & ::= & (h\ \ldots) \\
h & ::= & \langle name,\ (\lambda\ (x)\ e)\rangle \\
v & ::= & m \mid log \mid \mathtt{void} \mid (\lambda\ (x\ \ldots)\ e) \\
S & ::= & \langle E,\ H,\ (v\ \ldots)\rangle \\
E & ::= & (\mathtt{add}\ name\ E) \mid (\mathtt{begin}\ E\ e) \mid (\mathtt{send}\ E) \\
 & \mid & (v\ \ldots\ E\ e\ \ldots) \mid (o\ v\ \ldots\ E\ e\ \ldots) \\
 & \mid & (\mathtt{if0}\ E\ e\ e) \mid [\,]
\end{array}$$

$$\begin{array}{ll}
\langle E[(\mathtt{send}\ m_n)],\ H,\ (m\ \ldots)\rangle & [\textsc{send}] \\
\longrightarrow \langle E[\mathtt{void}],\ H,\ (m_n\ m\ \ldots)\rangle &
\end{array}$$

$$\begin{array}{ll}
\langle E[(\mathtt{add}\ name_a\ v)],\ (h\ \ldots),\ (m\ \ldots)\rangle & [\textsc{add}] \\
\longrightarrow \langle E[\mathtt{void}],\ (\langle name_a,\ v\rangle\ h\ \ldots),\ (m\ \ldots)\rangle & \\
\quad \text{where } (\langle name,\ v_h\rangle\ \ldots) = (h\ \ldots), & \\
\quad\quad name_a \notin (name\ \ldots), & \\
\quad\quad (\langle name_o,\ v_o\rangle\ \ldots) = H_0, & \\
\quad\quad name_a \notin (name_o\ \ldots) &
\end{array}$$

$$\begin{array}{ll}
\langle E[(\mathtt{remove}\ name)],\ H,\ (m\ \ldots)\rangle & [\textsc{rem}] \\
\longrightarrow \langle E[\mathtt{void}],\ (h_1\ \ldots\ h_2\ \ldots),\ (m\ \ldots)\rangle & \\
\quad \text{where } (h_1\ \ldots\ \langle name,\ v_1\rangle\ h_2\ \ldots) = H, & \\
\quad\quad (h_3\ \ldots\ \langle name,\ v_2\rangle\ h_4\ \ldots) = H_0 &
\end{array}$$

$$\begin{array}{ll}
S[((\lambda\ (x\ \ldots)\ e)\ v\ \ldots)] & [\beta\textsc{v}] \\
\longrightarrow S[e\{x := v,\ \ldots\}] &
\end{array}$$

$$S[(\mathtt{begin}\ v\ e)] \longrightarrow S[e] \qquad [\textsc{begin}]$$

$$S[(o\ v\ \ldots)] \longrightarrow S[\delta[\![o,\ v,\ \ldots]\!]] \qquad [\delta]$$

$$S[(\mathtt{if0}\ 0\ e_1\ e_2)] \longrightarrow S[e_1] \qquad [\textsc{if0}]$$

$$\begin{array}{ll}
S[(\mathtt{if0}\ v\ e_1\ e_2)] \longrightarrow S[e_2] & [\textsc{if!0}] \\
\quad \text{where } v \neq 0 &
\end{array}$$

Fig. 4. POP-PL model of evaluation.

During evaluation, a handler can change the machine state via the first three rules in Figure 4. The [SEND] rule puts a new message in the outgoing queue. The [ADD] and [REM] rules change the set of handlers used for the next message by adding a new handler or removing an existing one. Note that [ADD] and [REM] modify the *next* set of handlers. They cannot interfere with any of the current handlers. These rules also cannot make conflicting changes. They can add only handlers that are not present in both the next set of handlers and the original set of handlers passed to EVAL, and vice versa for [REM]. This guarantees that the order in which handlers are run cannot be determined by any handler. The rules use the initial set of handlers, H_0, to make sure that only suitable handlers are added or removed. The remaining rules are standard. The modifications to the set of handlers are not added to the log, as they are not clinically relevant but rather a detail of program execution.

```
          e ::= ....
              | (query e (qp ...))
         qp ::= (query-param e)
query-param ::= cut:
              | filter:
              | get-consec:
              | subseq:
              | length>=:
```

Fig. 5. Model for the querying DSL.

After each handler completes, the second and third components of the machine state it computed are used by EVAL for the next handler. When all handlers have been evaluated, EVAL produces the new handlers and the outgoing messages. In essence, EVAL folds over the current list of handlers, producing outgoing messages and the new set of handlers.

7.3 Querying

The querying extension in Figure 5 adds a series of operations to check if the current state of the world matches some criteria. The allowed operations are as follows: cut history at a certain point, filter out unneeded information from history, determine the consecutive events that match some criteria, find the largest subsequence of these events that matches some predicate, and decide if the number of events left passes some threshold. The operations are always applied in that order, and any operation may be a no-op.

$$\text{CUT} : (\mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

The CUT operation returns a log of all elements after (but not including) the most recent message for which the given predicate holds. Its purpose is to ask for history only after some event.

$$\text{FILTER} : (\mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

The FILTER operation removes information irrelevant to subsequent operations. For instance, it removes any non `aPTTResult` messages in the example in Figure 2.

$$\text{GET-CONSEC} : (\mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

The GET-CONSEC operation returns a log of all the most recent elements in the log that match the given predicate after the first element that does not match the predicate. This operation is similar in form to CUT, but its purpose is different. It is used when determining how many times an event has occurred without deviation. For instance, it is used when asking "How many consecutivee blood tests had a glucose value of above 45?" after filtering out all non-glucose messages from the log.

$$\text{SUBSEQ} : (\mathit{Message} \times \mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

The SUBSEQ operation finds the longest subsequence of the log such that every pair of consecutive elements in the output matches the given predicate. Its purpose is to remove elements that are too similar to their neighbors to be relevant. This is often used to remove test results that are so close together in time that they do not represent unique data points. For example, SUBSEQ: of 2-HOURS-APART?, which checks if two messages are at least 2 hours apart, and a log whose messages have the timestamps (in hours) `{1 4 5 6 7 8}` evaluates to a log with the messages whose times are `{1 4 6 8}`.

$$\text{LENGTH>=} : \mathit{Natural} \times \mathit{Log} \to \mathit{Log\ or\ Fail}$$

```
(query a-log ((length>=: 3)
              (cut: dose-change?)
              (filter: pain-score?)
              (get-consec: painscore>8?)
              (subseq: 1-hour-apart?)))
```

Fig. 6. A query in the syntax of the model.

The LENGTH>= operation asks if there are at least N elements in the log. Its purpose is to determine if there are enough events left to consider the findings relevant.

As an example that uses all of the features of the querying language, consider a question from a prescription for analgesia for hospitalized adults (Belknap et al. 2008): "Are there three consecutive pain scores with a severity greater than or equal to 8 cm?"[7] The formulation of this question takes into account the training of the nurses and leaves out information that cannot be left out in a program. Written less ambiguously the question is "With the current dosage, has the pain score been consistently high enough to raise the dosage?" We codify this as "are there at least three pain scores that are an hour apart and 8 cm or above since the last dose change in fentanyl, with no dosage readings between them below 8 cm?"[8] This translates to the abstract syntax in Figure 6.

8 THE CONCRETE SYNTAX

The concrete syntax for POP-PL is designed to abstract over the complex, unambiguous ideas doctors have, while resembling the natural language prescriptions that doctors write today. The syntax is designed for the language to be usable with little or no programming training on the part of the clinician. To ease a prescriber's transition into using POP-PL, we make the syntax support existing conventions for expressing prescriptions on article. This section covers the concrete syntax and describes how some forms map to the abstract syntax.

Literal Data. The only form of literal data in POP-PL is numbers, with associated units where appropriate. Any other data needed—such as a type of drug or the name of a mode of drug delivery—are bound to identifiers by the language implementation. This allows for arbitrary complexity in these data structures without increasing the programming burden on the physician and keeping the language more **unambiguous**. In addition, this makes certain program analyses easier. For example, by having a known list of identifiers that represent drugs, the language can generate a list of drugs that a prescription is prescribing, or, by looking at how these drugs are given, a language could determine exactly how many intravenous lines are needed to accommodate all prescribed drugs.

Case Insensitivity. POP-PL is case insensitive. In a medical context, this allows for Tall Man lettering (Filik et al. 2006). A common mode of medical error is confusion by a clinician of two drugs with similar names (Gerrett et al. 2009). For instance, the drug carboplatin is written as CARBOplatin to avoid confusion with cisplatin, which is written CISplatin. These capitalizations are not always consistent. In addition a case-sensitive language may be less **modifiable**. Therefore, the case of identifiers is ignored, and all identifiers are bound to data that will render with the

[7] An explanation of units and methods for assessing the level of pain can be found in Section 10.2.
[8] It is worth noting that medical professionals do not always arrive at the more precise meaning.

correct case to keep the output **unambiguous**. For example, `heparin`, `hepaRIN`, and `HEPARIN` will all render as `HEParin`.

Keyword Arguments. Functions in POP-PL allow variations on their keyword arguments. In addition to allowing keyword arguments in any order, the keywords may be synonyms from a fixed set of alternatives. For example,

```
giveBolus 80 units/kg of: HEParin by: iv
```

is equivalent to

```
giveBolus 80 units/kg of: HEParin via: iv
```

because `giveBolus` declares `via:` and `by:` as synonyms. The variation in arguments makes easier to **modify** prescriptions by letting physicians pick the words they would use in plain English orders. This way of handling multiple names for name parameters is inspired by Hypertalk (Apple Computer, Inc 1988), which was inspired by Smalltalk (Goldberg and Robson 1983).

After. The `after` form corresponds to the **common situation** where tasks must be executed with a time delay. It adds a handler that, after some amount of time, performs an action and then removes itself:

```
(add <some-unique-name>
  (λ (new-log)
    (if0 (time-passed? (time-of new-log)
                       (time-of old-log)
                       time)
         (begin <body-of-after>
                (remove <that-same-name>))
         void)))
```

Initially. The `initially` form puts a handler in the initial state that removes itself after execution:

```
⟨<some-unique-name>,
 (λ (a-log)
   (begin
     <body-of-initially>
     (remove <that-same-name>)))⟩
```

Basic Handlers. The `name:` form corresponds to a handler of the given name in the initial state.

Whenever. The `whenever` form is a triple-purpose form: It serves as conditional dispatch, it matches the current message against a pattern, and it hosts the querying DSL. The `whenever new` form checks the current message against some known message pattern and binds using that pattern. A `whenever` form whose expression is a query tests against the querying DSL. A `whenever` form with pipes in its body acts as a multi-armed conditional that runs inside the scope of the outer `whenever`. This is designed to look like the tables that are often used to represent conditions in current order sets (Washington Adventist Hospital 2009).

Querying with Whenever. A `whenever` form in the concrete syntax—without a `new`—equates to an `if0` whose expression is a query. So the query from Figure 6 would be written as

```
whenever painscore > 8 cm, 1 hour apart, x3,
since the last change in: fentanyl
```

The `x3` form maps to the LENGTH>= operation. The choice of using `x3` instead of something like `3 times` is because in the prescriptions we have seen most doctors use the first form, with the hope of keeping the language **unambiguous**.[9] The `since last change in: fentanyl` form maps to the CUT: operation. The `1 hour apart` form maps to the SUBSEQ: operation. These three may or may not appear, in any order. The first expression of the `whenever` corresponds to both the FILTER and GET-CONSEC operations and is an arbitrary expression. This expression may reference the name of up to one message. That message is the one filtered for. The GET-CONSEC: operation then uses a function that binds the message name and evaluates the expression.

9 PUTTING IT ALL TOGETHER

To give a sense of how programs evaluate, this section works through some interesting parts of the REPL interactions in Section 6 and inspects the intermediate states of the evaluator. Specifically, this section inspects how POP-PL evaluates multiple handlers when responding to a message and looks at how the `after:` form modifies the handler set to schedule actions in the future. These are looked at in the context of the program in Figure 2.

9.1 Stepping Through Handlers

The program in Figure 2 begins execution with three handlers, one for the `initially` block, one for the infusion, and one for the monitoring orders. In the first interaction, the evaluator runs each handler and outputs the three messages seen in Section 6—two messages to set up the infusion and one to begin the monitoring:

To illustrate how the evaluator loops through the handlers, we step through this interaction, which handles the program startup, one handler at a time. The initial machine state of the evaluator is

```
⟨(INITIALLY (list)),
 (INITIALLY INFUSION APTTCHECKING),
 ()⟩
```

In this machine, INITIALLY is the handler for the code on lines 5–7 in Figure 2, APTTCHECKING is the handler for lines 26–28, and INFUSION is the handler for lines 9–24. The start handler set H_0 is (INITIALLY APTTCHECKING INFUSION). The initial expression invokes the INITIALLY handler on the initial log, which is empty.

The INITIALLY handler generates the messages to give a bolus and start the IV drip and then deletes itself. The eval loop will then call the next handler on the current log. The new machine state is

```
⟨(INFUSION (list)), (INFUSION APTTCHECKING), (<start> <bolus>)⟩
```

This state is invoking INFUSION on the log and has a new set of output handlers that has INITIALLY removed and the two new messages in the output message set.

The INFUSION handler does nothing, as it only responds to `aPTTResult` messages. The eval loop then calls the final handler on the current log:

```
⟨(APTTCHECKING (list)), (INFUSION APTTCHECKING), (<start> <bolus>)⟩
```

The APTTCHECKING handler emits a message telling the nurse to check the patient's aPTT value, because there are neither two stable aPTT values nor has there been a measurement in the last 6 hours. Thus, once this handler completes, the final state is

```
⟨void, (INFUSION APTTCHECKING), (<check> <start> <bolus>)⟩
```

There are no more handlers to run, so the evaluator returns the new set of handlers (APTTCHECKING INFUSION) and three messages (`<check> <start> <bolus>`).

[9] Section 10 shows that it in fact made the language more confusing.

9.2 Handlers and `after:`

To illustrate how the `after:` form handles scheduling, the next example shows two sequential interactions: one where a new aPTT value is read and another after 1 hour has passed. The result of receiving the aPTT message is as shown in Section 6:

```
> aPTTResult 200 seconds
[hold HEParin]
```

The output is the message to hold the heparin from line 21. In addition, in 1 hour, the heparin will be resumed. This fact is encoded in the new set of handlers. The eval loop runs much like before, calling each handler on the new log. The initial state is

```
⟨(APTTCHECKING (list <aPTT-of-200> <check> <start> <bolus>)),
 (INFUSION APTTCHECKING),
 ()⟩
```

which will call the APTTCHECKING on the current log. This log contains the three messages from the last run of the program, as well as the new incoming message. After evaluating all of the handlers the machine state is

```
⟨void,  (INFUSION APTTCHECKING INFUSIONS-AFTER),  (<hold>)⟩
```

This state has a single output message in its output set (`<hold>`) and three output handlers (INFUSION APTTCHECKING INFUSIONS-AFTER). The first two handlers are the same as the previous states. The third handler, INFUSIONS-AFTER corresponds to lines 22–24 of the program. Once an hour has passed, the handler will execute and remove itself.

We can see this if we wait an hour:

```
> wait 1 hour
[decrease HEParin 3 units/kg/hour]
[restart HEParin]
```

The `wait 1 hour` command sends one hour's worth of heartbeat messages to the program, which are processed one at a time. With the exception of the last message, these message result in no changes, so their output message set is (), and their output handler set is (INFUSION APTTCHECKING INFUSIONS-AFTER). When last heartbeat message message arrives, the initial machine state is

```
⟨(APTTCHECKING (list <heartbeat> ... <aPTT-of-200> <check> <start> <bolus>)),
 (INFUSION APTTCHECKING INFUSIONS-AFTER),
 ()⟩
```

The handlers are as before. The log is the previous log with 1 hour of heartbeat messages at the head. Only the INFUSION-AFTER responds to these message. It will emit messages to restart the heparin and decrease its dosage. Therefore, once all handlers have been evaluated, the final state is

```
⟨void,  (INFUSION APTTCHECKING),  (<restart> <decrease>)⟩
```

The final state has the expected output messages, and the new set of handlers, which still contains the INFUSION and APTTCHECKING handlers, but with the INFUSION-AFTER handler removed.

10 ASSESSING THE LANGUAGE

To assess POP-PL we performed two evaluations. First, we translated several prescriptions to POP-PL to understand how well the language can express them. This first evaluation focuses on the requirements related to representing prescriptions, specifically how well POP-PL **represents questions** and **handles common situations**. Second, we evaluated how well physicians can use POP-PL via a survey. The second evaluation focuses on the requirements related to usability, specifically on whether POP-PL is **unambiguous**, is **modifiable**, and has **predictable behavior**.

10.1 An ICU Insulin Prescription

Insulin is an important and necessary drug for controlling blood sugar. However, if the insulin dose is too high, the blood sugar can fall to dangerously low levels that can cause organ damage and can be fatal if not corrected. The effect of insulin on blood sugar depends on diet, activity, disease, and physiologic stress. Therefore, the correct dose of insulin varies considerably among patients and also varies over time in individual patients as the clinical situation changes. The program listed in Figure 7 and Figure 8 is a prescription for administering IV insulin to patients in an Intensive Care Unit[10] (Loyola Division of Endocrinology 2004). The program has been shortened for clarity in two ways: Entries in the blood glucose lookup table have been collapsed, and four other lookup tables have been elided.

The program consists of five handlers. The first is an `initially` block that starts the insulin IV. The second handles responding to the patient's blood glucose (`BG`) levels. It consists of a table that maps blood glucose measurements to fixed insulin doses. This differs from the heparin prescription in Figure 2, where the table mapped aPTT values to *changes* in the heparin dose. While it would be possible to collapse the heparin and insulin dosing tables to simple formulas, tables are generally used when describing prescriptions.

This mapping of blood glucose values to fixed insulin values demonstrates an design flaw in POP-PL. If two consecutive blood glucose values are within the same range, then the program will generate two successive messages to change the insulin dose to the *same* value, as seen on lines 40 and 44 of Figure 8. Fixing this would involve POP-PL comparing new blood glucose values with previous ones to avoid redundant instructions. POP-PL has no mechanism for **representing this question**, and such redundant messages are expected to increase alert fatigue.

The third handler covers the `monitoring` orders, indicating how often to recheck blood glucose levels. This handler will check the blood glucose every 2 hours if the glucose is stable and check every hour otherwise.

This handler demonstrates another another flaw in POP-PL. The original prescription has three different monitoring schedules with overlapping execution conditions. While the core querying language could check these conditions, the surface syntax cannot.

The fourth handler, `mealBolus`, delivers an extra dose of insulin whenever the patient eats a meal. Like the `aPTT` identifier in the heparin prescription, `carbs` is bound based on the payload of the `meal` message. It is the number of grams of carbohydrates in the meal. Therefore, this handler gives one unit of insulin for every 10g of carbohydrates.

The last handler covers `hypoglycemia`, the state where the patient's blood glucose is dangerously low. If the patient's glucose is below 60mg per deciliter, then the patient's doctor is notified and the blood glucose is redrawn every 15 minutes until the issue is resolved. The `latest` keyword on line 32 has the `whenever` dispatch on every time step based on the last `BG` message, instead

[10] In other clinical settings, a typical insulin prescription checks the patient's blood glucose a few times a day, usually some time before and after meals. Because the prescription is meant for intensive care, it checks the blood glucose and changes the insulin dosage frequently, independently of when meals are given.

```
#lang pop-pl
used by Loyola

initially
  start insulin at: 1 unit/hour by: iv

insulinUpdating:
  whenever new BG
     BG < 60                    | hold insulin
     BG between 60 and 79       | change insulin to: 0.1 units/hour
     BG between 80 and 109      | change insulin to: 0.2 units/hour
     BG between 110 and 119     | change insulin to: 0.5 units/hour
     BG between 120 and 149     | change insulin to: 1.0 units/hour
     BG between 150 and 179     | change insulin to: 1.5 units/hour
     BG between 180 and 209     | change insulin to: 2.0 units/hour
     BG between 210 and 239     | change insulin to: 2.0 units/hour
     BG between 240 and 269     | change insulin to: 3.0 units/hour
     BG between 270 and 299     | change insulin to: 3.0 units/hour
     BG between 300 and 329     | change insulin to: 4.0 units/hour
     BG between 330 and 359     | change insulin to: 4.0 units/hour
     BG > 360                   | change insulin to: 6.0 units/hour
                                | notifyDoctor
monitoring:
  every 2 hours checkBG whenever BG between 70 and 180, x4
  every 1 hour checkBG whenever BG outside 70 to 180, x4

mealBolus:
  whenever new meal
    giveBolus 1 unit * carbs / 10 grams of: insulin by: iv

hypoglycemia:
  whenever latest BG < 60
    every 15 minutes checkBG
  whenever new BG and BG < 60
    notifyDoctor

```

Fig. 7. Insulin prescription, Part 1.

of waiting for a new one. The `insulinUpdating` handler will also withhold the administration of insulin when the blood glucose is low.

10.2 A Patient–Oriented Prescription for Analgesia

The program in Figure 9 is a prescription for the safe administration of the opioid fentanyl (Belknap et al. 2008). The original on-paper English prescription is shown in Figure 10. Opioids are used to treat pain but also can suppress respiratory drive if given in too high of a dose; this can be fatal. The opioid drug dose that causes fatal respiratory depression may be not much higher than the dose needed to achieve adequate control of pain. The safe and effective dose of an opioid drug can vary eightfold among patients (Mather and Glynn 1982). As a further challenge, the safe dose may vary substantially in an individual patient, depending on pain severity, which can vary

```
--- Tests ---
[start insulin at: 1 unit/hour by: iv]
[checkbg]
> BG 120
[change insulin to: 1 units/hour]
> wait 1 hour
[checkBG]
> BG 120
[change insulin to: 1 units/hour]
> wait 1 hour
[checkBG]
> BG 59
[notifyDoctor]
[hold insulin]
> wait 15 minutes
[checkBG]
```

Fig. 8. Insulin prescription, Part 2.

according changes in the disease that is causing the pain. Additionally, concomitantly administered drugs can reduce pain severity or increase the analgesic effect of the opioid. If the severity of the patient's pain suddenly drops and the opioid dosage is not adjusted accordingly, then respiratory depression may ensue, and, if severe, the patient may die.[11]

As before, the first handler is an `initially` block. It sets up a fentanyl infusion and initializes a Patient Controlled Analgesia (PCA) pump. A PCA pump allows the patient to administer doses of painkiller on demand.[12] This corresponds to the "Set-up" section marked in the right-hand margin of the original prescription.

The next section, `breakthroughPain`, handles increasing the fentanyl dose delivered by the PCA pump if the patient is in severe pain. Pain is measured on a Visual Analog Scale (VAS), an example of which can be found in Figure 11.[13] The VAS used here measures pain scores on a scale of 0 to 10cm. When a VAS reading over 8cm occurs, the PCA pump output is increased by 10mg, and another VAS reading is scheduled for 1 hour later. In addition, the doctor is notified whenever three consecutive pain scores are above 8cm.

The penultimate handler, `minimalPain`, decreases the opioid dosage if the patient has consistently low pain to avoid an opioid overdose due to a reduction in pain. If three consecutive pain scores are below 2cm, then the PCA pump output is decreased by 10mg. The consecutive readings must be taken between the PCA pump dosage changes, otherwise the prescription could change the dose based on pain score information from a different dosage of opioid. It is important to note that this `since last decrease onDemandFentaNYL` clause is not present in the original prescription—it is left implicit. It is possible that most nurses actually follow this instruction, or it is possible that acute events caused by disregarding this instruction are incredibly rare. Although this is unknown, the author of this prescription states that this is the intended behavior.

The final handler contains the `monitoring` orders. Whenever a nurse does a vital sign recording on a patient (checking blood pressure, drawing blood for labs, etc.) the program will request that

[11]This tolerance shift occurs because pain stimulates parts of the nervous system that opioids depress.
[12]Although this administration route seem may dangerous, PCA pumps have safety features to prevent overdosing and are a standard form of opioid administration.
[13]Remarkably, Prince et al. (1983) and Prince et al. (1994) have shown that visual analog scales have ratio scale properties and are valid and useful for clinical measurement. This, along with the ease of administration and scoring in clinical settings make the VAS a simple yet powerful pain measurement technology in both research and health care settings.

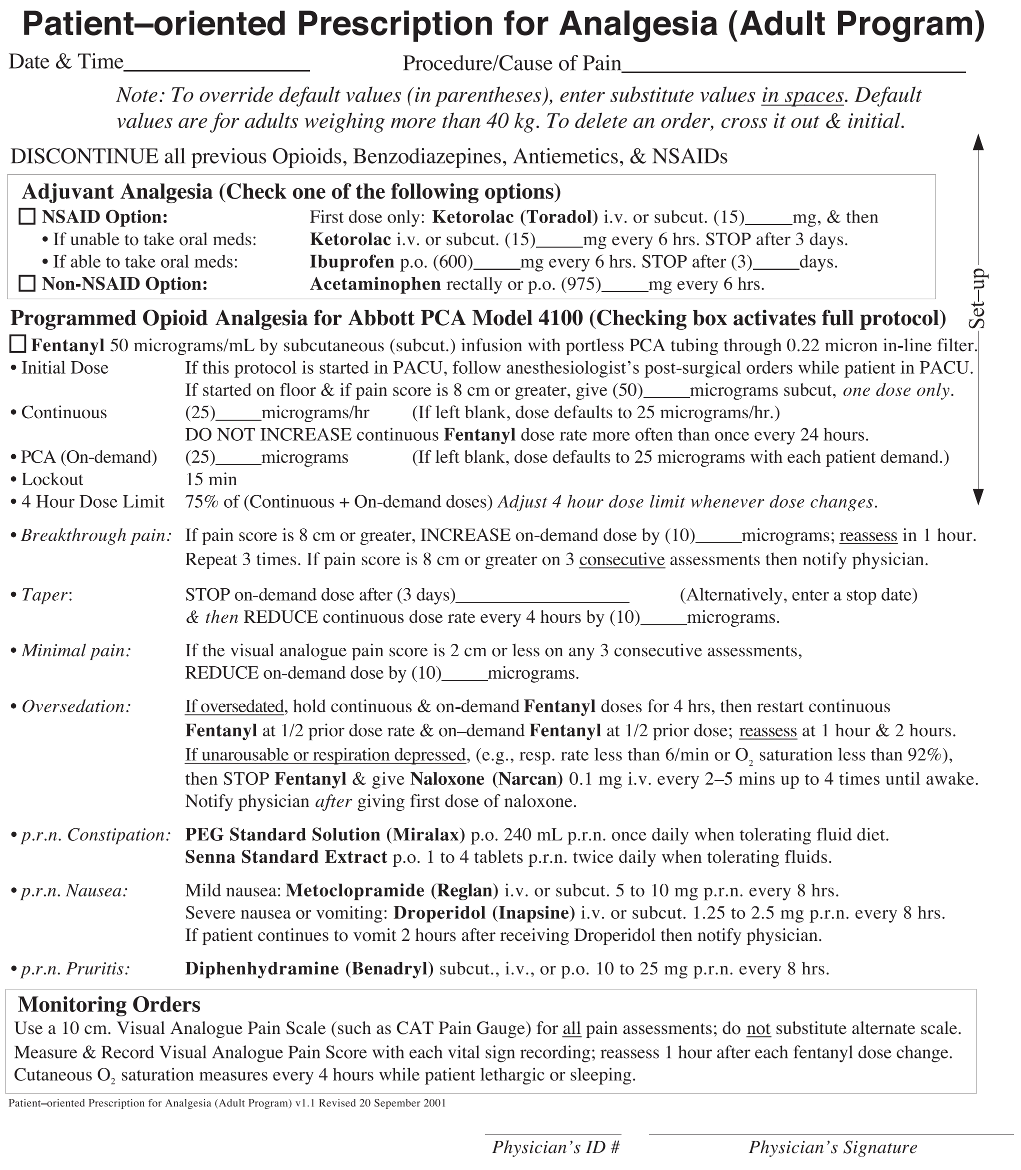
```
#lang pop-pl

used by OSFSaintFrancis

initially
  start 25 micrograms/hour of: fentaNYL
  set onDemandFentaNYL to: 25 micrograms

breakthroughPain:
  whenever new painscore and score > 8 cm
      increase onDemandFentaNYL by: 10 micrograms
      after 1 hour
         checkPainScore

  notifyDoctor whenever painscore > 8, x3

minimalPain:
  whenever painscore < 3 cm, x3, 1 hour apart,
           since last decrease onDemandFentaNYL
      decrease onDemandFentaNYL by: 10 micrograms

monitoring:
  whenever new vitalSignRecording
    checkPainScore

--- Tests ---

[start 25 micrograms/hour of: fentaNYL]
[set onDemandFentaNYL 25 micrograms]
> painscore 10
[increase onDemandFentaNYL 10 micrograms]
> wait 61 minutes
[checkPainScore]
> painscore 2
> painscore 2
> painscore 2
> wait 2 hours
> painscore 2
> wait 2 hours
> painscore 2
[decrease onDemandFentaNYL by: 10 micrograms]
```

Fig. 9. Prescription for opioids from Belknap et al. (2008).

the patient's pain be assessed. This vital sign recording will occur either at the request of another prescription or in the course of standard hospital care.

POP-PL cannot represent three key features of this prescription. First, this prescription contains a negative order that, in vernacular English, means: "give no other pain medication while this prescription is active."[14] POP-PL has no mechanism for expressing such an order.

[14]The phrasing in the prescription is "DISCONTINUE all previous Opioids, Benzodiazepines, Antiemetics, & NSAIDs."



Patient–oriented Prescription for Analgesia (Adult Program)

Date & Time________________ Procedure/Cause of Pain________________________

Note: To override default values (in parentheses), enter substitute values in spaces. Default values are for adults weighing more than 40 kg. To delete an order, cross it out & initial.

DISCONTINUE all previous Opioids, Benzodiazepines, Antiemetics, & NSAIDs

Set–up

Adjuvant Analgesia (Check one of the following options)

☐ **NSAID Option:**	First dose only: **Ketorolac (Toradol)** i.v. or subcut. (15)_____mg, & then
• If unable to take oral meds:	**Ketorolac** i.v. or subcut. (15)_____mg every 6 hrs. STOP after 3 days.
• If able to take oral meds:	**Ibuprofen** p.o. (600)_____mg every 6 hrs. STOP after (3)_____days.
☐ **Non-NSAID Option:**	**Acetaminophen** rectally or p.o. (975)_____mg every 6 hrs.

Programmed Opioid Analgesia for Abbott PCA Model 4100 (Checking box activates full protocol)

☐ **Fentanyl** 50 micrograms/mL by subcutaneous (subcut.) infusion with portless PCA tubing through 0.22 micron in-line filter.

• Initial Dose	If this protocol is started in PACU, follow anesthesiologist's post-surgical orders while patient in PACU. If started on floor & if pain score is 8 cm or greater, give (50)_____micrograms subcut, *one dose only*.
• Continuous	(25)_____micrograms/hr (If left blank, dose defaults to 25 micrograms/hr.) DO NOT INCREASE continuous **Fentanyl** dose rate more often than once every 24 hours.
• PCA (On-demand)	(25)_____micrograms (If left blank, dose defaults to 25 micrograms with each patient demand.)
• Lockout	15 min
• 4 Hour Dose Limit	75% of (Continuous + On-demand doses) *Adjust 4 hour dose limit whenever dose changes.*
• *Breakthrough pain:*	If pain score is 8 cm or greater, INCREASE on-demand dose by (10)_____micrograms; reassess in 1 hour. Repeat 3 times. If pain score is 8 cm or greater on 3 consecutive assessments then notify physician.
• *Taper*:	STOP on-demand dose after (3 days)________________ (Alternatively, enter a stop date) *& then* REDUCE continuous dose rate every 4 hours by (10)_____micrograms.
• *Minimal pain:*	If the visual analogue pain score is 2 cm or less on any 3 consecutive assessments, REDUCE on-demand dose by (10)_____micrograms.
• *Oversedation:*	If oversedated, hold continuous & on-demand **Fentanyl** doses for 4 hrs, then restart continuous **Fentanyl** at 1/2 prior dose rate & on–demand **Fentanyl** at 1/2 prior dose; reassess at 1 hour & 2 hours. If unarousable or respiration depressed, (e.g., resp. rate less than 6/min or O_2 saturation less than 92%), then STOP **Fentanyl** & give **Naloxone (Narcan)** 0.1 mg i.v. every 2–5 mins up to 4 times until awake. Notify physician *after* giving first dose of naloxone.
• *p.r.n. Constipation:*	**PEG Standard Solution (Miralax)** p.o. 240 mL p.r.n. once daily when tolerating fluid diet. **Senna Standard Extract** p.o. 1 to 4 tablets p.r.n. twice daily when tolerating fluids.
• *p.r.n. Nausea:*	Mild nausea: **Metoclopramide (Reglan)** i.v. or subcut. 5 to 10 mg p.r.n. every 8 hrs. Severe nausea or vomiting: **Droperidol (Inapsine)** i.v. or subcut. 1.25 to 2.5 mg p.r.n. every 8 hrs. If patient continues to vomit 2 hours after receiving Droperidol then notify physician.
• *p.r.n. Pruritis:*	**Diphenhydramine (Benadryl)** subcut., i.v., or p.o. 10 to 25 mg p.r.n. every 8 hrs.

Monitoring Orders

Use a 10 cm. Visual Analogue Pain Scale (such as CAT Pain Gauge) for all pain assessments; do not substitute alternate scale.
Measure & Record Visual Analogue Pain Score with each vital sign recording; reassess 1 hour after each fentanyl dose change.
Cutaneous O_2 saturation measures every 4 hours while patient lethargic or sleeping.

Patient–oriented Prescription for Analgesia (Adult Program) v1.1 Revised 20 September 2001

Physician's ID # *Physician's Signature*

Fig. 10. Opioid prescription from Belknap et al. (2008).

Second, the "Adjuvant Analgesia" portion of the prescription is, logically, a state machine that controls which non-opioid painkillers are given and in what dosage. It currently cannot be represented in POP-PL.

Finally, the prescription contains several orders that are marked as "*p.r.n.,*" which stands for *pro re nata*, latin for "as the circumstance arises." It is commonly used in medicine to mean "as needed," and the corresponding orders are executed at the nurse's discretion, without the need for a physician's order. We do not yet understand how to best represent the interaction between the prescription and the rest of the hospital for such orders.

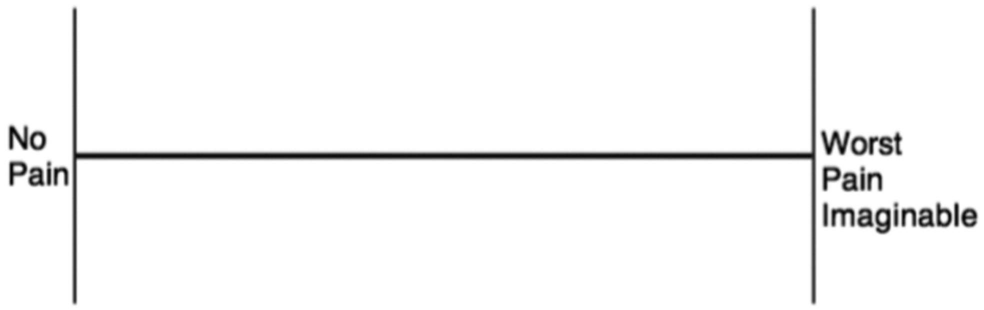


Fig. 11. Example of a VAS for pain. Patients mark a point on the scale that represents a self assessment of their pain level. The nurse would then measure the distance of that point from zero.

Overall, we were able to implement 7 of the 15 bullet-point and boxed orders in this prescription. At a higher level, POP-PL was able to effectively **handle common conditions** and **represent questions** for prescriptions that do not need to consider their own history. However, prescriptions that did need to consider their own history, e.g., to avoid repeating messages, are difficult to express. In addition POP-PL cannot represent *p.r.n.* orders or negative orders.

10.3 A Usability Survey

In addition to representing prescriptions, POP-PL must also be comprehensible to clinicians as they conduct their management of their patient. This evaluation focuses on the requirements of **unambiguity**, **predictability**, and **modifiablility**. To evaluate POP-PL based on this use criteria, we administered a survey to medical professionals that asked them to identify specific parts of a prescription and to make minor modifications to it. The Northwestern University Institutional Review Board (NUIRB) granted approval to conduct this research with one of the authors as Principal Investigator (SMB). This NUIRB-approved project was conducted under the auspices of the Northwestern-based Research on Adverse Drug events And Reports (RADAR) Program under the direction of one of the authors (D.P.W.).

The survey began with a 5-minute explanation of an insulin protocol written in POP-PL.[15] The participants were then presented with a survey containing the protocol in Figure 2 (but without the test cases). They were asked the following questions (with typos preserved from the original survey):

(1) Circle the part of the program that handles appt values of 50 seconds.
(2) What happens when we get a an appt of 50 seconds?
(3) Modify the part of the protocol you have circled so that it will instead increase the heparin dosage by 2 units/kg/hour.
(4) Circle the part of the protocol that controls how often an aptt test is run.
(5) What is the least frequent the test is run?
(6) Given each of the following test results, when would the next aPTT check be done if the protocol is accurately followed?
aPtt = 50 seconds at 6am
aPtt = 80 seconds at noon
aPtt = 90 seconds at 6pm

[15]The script for the survey introduction, the full survey and the tallied results, and implementation of POP-PL are available in the supplementary materials.

Role	Count
Attending Physician	2
Resident Physician	9
Medical Student	7
Post-Training Pharmacist	6
Resident Pharmacist	11
Pharmacy Student	11
Advanced Practice Nurse	4
Blank	1
Total	51

Fig. 12. Demographics of the survey.

Type	Count
Comfortable with Programming	14
Programming Training	6

Fig. 13. Programming experience of survey participants.

In addition, each participant was asked about his or her medical training, formal programming training, and comfort with programming.

Questions 1, 4, and 5 focus on evaluating **ambiguity**. Questions 2, 5, and 6 focus on **predictability**. Question 3 looks at basic **modifiablility**.

Understanding the Demographics. The survey was given to 51 respondents. Each respondent had one of seven types of medical training: attending physician, resident physician, medical student, post-training pharmacist, resident pharmacist, pharmacy student, or advanced practice nurse. These represent three stages of training (student, resident, post-training) across two tracks (pharmacist or physician) plus advanced practice nurses.

The physicians' role is to observe and communicate with patients, make diagnoses, and coordinate patients' medical care. The formal training of physicians begins in medical school, which is typically a 4-year post-graduate program. After medical school, many physicians undergo additional training in residency programs, typically 2 to 7 years (depending on specialty) and usually based in hospitals or clinics. Once physicians complete their residency, they become attending physicians. The pharmacists' role is to provide expertise about drug therapy, coordinate the process of drug delivery, and monitor the efficacy and toxicity of drugs in patients. The pharmacists' track is similar to the doctors' but with a difference in degree: five years of pharmacy school followed by an optional 2-year residency. Advanced practice nurses (APNs) are nurses with additional post-graduate education; depending on the state and institution, APNs can write prescriptions and provide other clinical care to patients without the direct supervision of a physician.

In addition, respondents reported their programming training in free-form text and their relative comfort with programming on a scale ranging from 0 to 10, with 0 being the least. Of the 51 respondents, 6 had some formal training in programming (which at most amounted to a single undergraduate course), and 14 had a non-zero comfort with programming. The remaining respondents in each category either claimed no training/zero comfort or left the question unanswered.

The survey was administered in three settings. Physicians completed the survey in small groups in the hospital, typically right after finishing rounds. All of the pharmacists completed the survey at the same time after listening to a seminar on an unrelated topic. The APNs completed the survey in a meeting scheduled for this purpose. The breakdown of the survey participants by demographics is in Figure 12, and a breakdown by programming experience is in Figure 13.

question	yes	partial	no
1st	98.03%	0.00%	1.96%
2nd	82.35%	11.76%	5.88%
3rd	86.27%	1.96%	11.76%
4th	94.11%	1.96%	3.92%
5th	82.35%	0.00%	17.64%
6th	39.21%	45.09%	15.68%

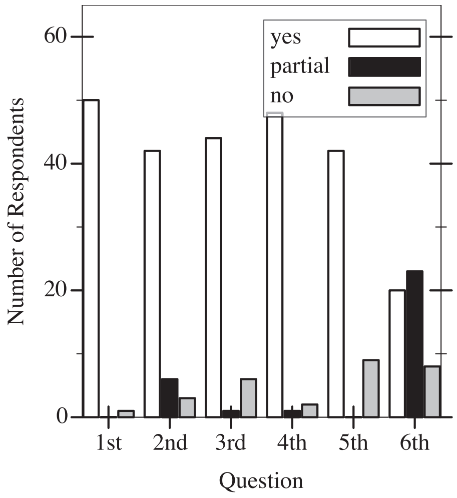


Fig. 14. Survey data.

Analyzing the Data. Three graders evaluated each survey. The questions were graded with three possibilities: "yes," meaning the respondent answered the question completely correctly; "partial," meaning the respondent was neither completely correct nor completely incorrect; and "no," meaning the respondent was completely incorrect. The results for each question are displayed both by percentage and graphically in Figure 14. The graphs are broken down by participant specialty in Figure 15.

Data were analyzed using univariate optimal discriminant analysis (UniODA), a non-parametric statistical methodology for which no distributional assumptions are required (model parameters and exact Type I error rates are always valid) that explicitly maximizes model classification accuracy for each specific sample and hypothesis. For UniODA, the index of classification accuracy is effect strength for sensitivity (ESS), a normed index on which ESS = 0 indicates the accuracy that is expected by chance, and ESS = 100 indicates perfect, errorless prediction: By convention, ESS < 25 is a relatively weak effect; ESS < 50 is a moderate effect; ESS < 75 is a relatively strong effect; and ESS > 75 is a strong effect (Yarnold and Soltysik 2004, 2016).

UniODA was performed to assess inter-rater agreement for all 18 pairings of three independent raters scoring six separate test questions. Due to absence of variability in scores assigned by at least one rater, no model was possible for 7 pairings: For these combinations of raters and questions inter-rater agreement was 91.5% or greater. For the remaining 11 pairings, 5 indicated perfect agreement, 2 indicated strong agreement, and 4 indicated moderate agreement: Inter-rater agreement was 75.5% or greater for these analyses, and all results were statistically significant with Bonferroni adjusted $p < 0.05$. For all questions except the question about programming comfort, inter-rater agreement was 90.2% or greater. All instances of rater disagreement were resolved to consensus in post-rating discussion.

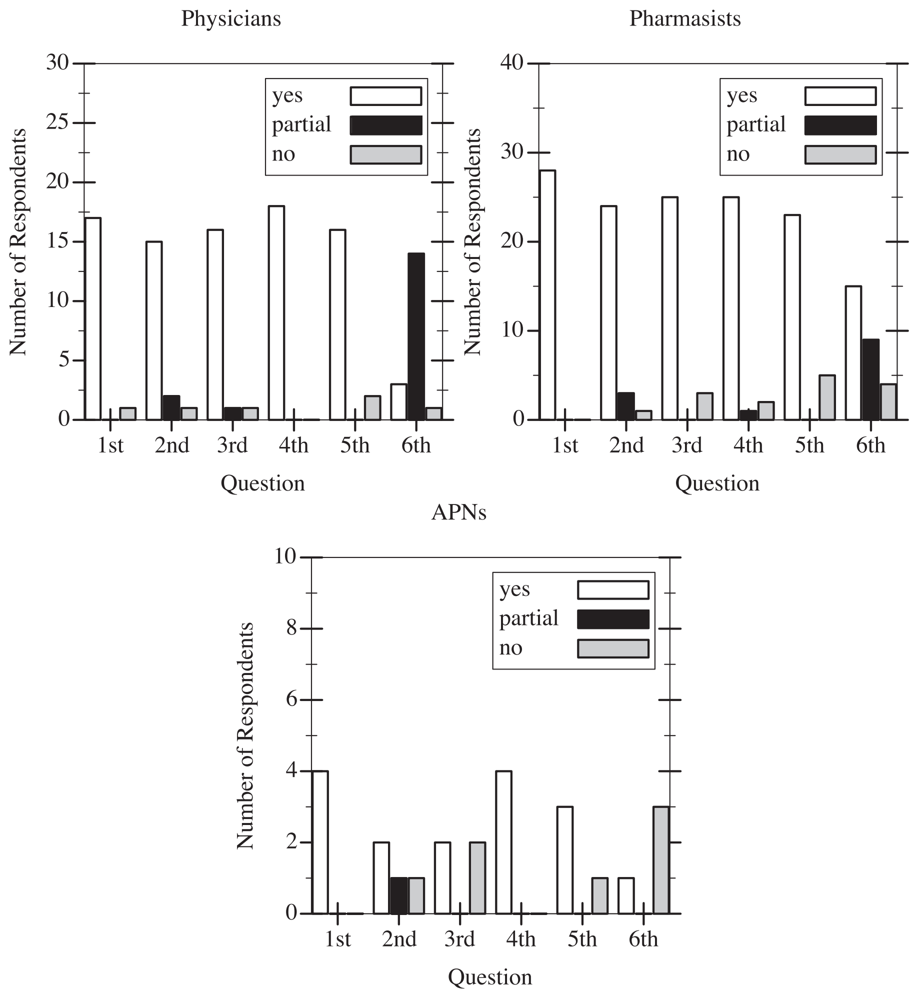


Fig. 15. Survey data by track.

The class (dependent) variable was whether or not the respondent obtained a perfect score across all six questions. Obtaining a perfect score was not predicted by score on any of the six individual questions (all $p < 0.13$) or by track ($p < 0.73$), level ($p < 0.32$), or years of experience ($p < 0.23$). However, obtaining a perfect score was very strongly related to having formal training in programming: $p < 0.0004$, ESS = 95.0. Non-zero comfort with programming was a strong predictor of obtaining a perfect score: $p < 0.016$, ESS = 53.3. No multivariable model was possible after accounting for computer programming training.

Interpreting the Analysis. The nearly perfect association between an errorless survey score and programming experience and comfort suggests that physicians without programming experience may be uncomfortable adopting these methods. However, examination of the data in Figure 14 clarifies that the majority of people only had problems with the last question, which dealt with the querying DSL. The first five questions had reasonably high scores. In addition, only prior programming training was predictive, which implies that experience or role in the hospital did not affect how easy it was to use the language.

From this we draw two conclusions. First, the querying DSL and its syntax are confusing to medical practitioners. A closer inspection of the survey results revealed that the most common mistakes on the last question were that the respondent believed the aPTT test needed to be in the effective range only once or that the test should be run on the faster schedule after two aPTT tests were in range. The first of these misunderstandings show that the x2 syntax is confusing. This is supported by anecdotal evidence from several physicians we have spoken to, who find "x2" confusing when used in conventional prescriptions. The second of these misunderstandings would seem to imply that something in the querying language is confusing. Given that experience with programming predicts a perfect score, perhaps the querying language strays too far from the natural language doctors are accustomed to and should be simplified or at least hidden with some form of helper function.

Second, the high scores on the first five questions, in addition to experience and role not predicting overall score, provides strong evidence that clinicians can interpret and modify prescriptions written as programs with little training, with the exception of the querying language, which is ambiguous.

10.4 Threats to Validity

The primary threat to the validity of the analysis of program representation is the small number of prescriptions that we evaluated. While in the medical experience of the clinical pharmacologist on our team, the insulin and heparin prescriptions are representative of insulin and heparin prescriptions, the opioid prescription is novel, and the prescriptions may not be representative of medical protocols overall.

There are two major threats to validity to the survey. First, all of the pharmacists and nurses were from a single institution. This may bias the results, as the participants were all familiar with the same hospital system and operating protocols. It is possible that variations in this could affect the usability of the language.

Second, it is likely that all participants were familiar with similar protocols for administering insulin and heparin, as these protocols are fairly common. This may have aided in their understanding of the protocols presented in the survey.

11 THE FUTURE OF POP-PL

POP-PL's design continues to evolve. The evaluation in this article has revealed four major enhancements we expect to support in future versions: the ability to express constraints requiring or preventing the occurrence of certain tasks, the ability to express orders requiring state-machine like behavior, a querying language that is easier for clinicians to understand, and support for *p.r.n.* orders.

Some existing systems, like the Declare framework (van der Aalst et al. 2009), can represent positive or negative constraints. However, in POP-PL, constraints will likely need to be handled in two ways: the first statically checking that the prescription could violate the order, and the second watching at runtime to ensure that no action taken by hospital staff or automated processes violates the order.

Because the underlying semantics of POP-PL can represent state machines, it may be possible to simply add a suitable representation to the surface syntax. The next step will be to design an appropriate syntax that meets the design requirements. One potential solution would be to use a graphical syntax, which is used in some existing medical protocols (e.g., The SPRINT Research Group (2015)). However, usability studies of languages with visual syntax have produce mixed results (Green and Petre 1996; Green and Petre 1992; Whitley 1997), suggesting that they not appropriate in all situations. As such, any visual syntax will have to be carefully designed and evaluated.

While the difficulties clinicians have understanding the querying language are, at least partially, related to the syntax of the language, it remains possible that the underlying semantics of the querying language do not match the clinicians thought process. As shown by the existence of languages like iAlarm, creating a language for this is difficult and useful in its own right—the ability to automatically recognize patterns of events in a hospital is powerful.

The final flaw is POP-PL's inability to represent *p.r.n.* orders. We have not been able to distill design requirements for these orders or determine error conditions that may occur when they are not handled properly. However, our team has observed errors in the execution of these orders in the hospital, generally relating to the order not being performed when expected by the physician. These errors seem to be related to both ambiguous orders and to different understandings among patients, nurses, pharmacists, and physicians as to how clinical processes work. One potentially useful effect of using POP-PL to express *p.r.n.* orders may be to facilitate a common understanding as to the meaning of these orders.

In addition to the problems the evaluation revealed, more design requirements can be derived by careful study of how POP-PL will interact with a prescriber's workflow. For example, consider how POP-PL's representation of prescriptions interacts with the hospital over time. Plans of care evolve as new knowledge about the patient is gained or as new events change the patient's needs. This leads to the following requirement.

Adaptability and a Failure to Appropriately React. The language must allow for the plan of care to be adapted to changing circumstances without interrupting the execution of the prescription. For example, if a patient on blood pressure control experiences lightheadedness, the prescription may be adapted to have a higher target blood pressure for a time.

The associated error condition is a failure to appropriately react to changes in the patient state. This error could manifest in several ways: no reaction, reaction that restarts prescriptions, a delayed response, and so on.

Once we add this requirement, we must consider its effect on the rest of the system. For instance, consider the following scenario: A patient is on a blood pressure medication with monitoring orders to adjust the medication if the blood pressure leaves the target range. Soon after reaching the target range, the patient experiences lightheadedness. To account for this the physician changes the prescription to lower the dose of blood pressure medication and increase the target blood pressure range. The new prescription also includes an alert to remind the physician to reassess the patient and possibly return the target blood pressure to a lower target after 1 week. However the physician may fail to adjust the monitoring orders for the patient's blood pressure; despite the physical proximity, there is no system that prevents the physician from making this error once the protocol has been executed. This implies that the current system may not meet the **representing connection** requirement if the language is extended for the **adaptability** requirement. However, one could imagine that this requirement could be met, at least partially, at the IDE level, with prompts reminding physician to edit some parts of a protocol when others are modified. This leads to an interesting point: The language itself may not, and in fact likely cannot, meet all of the requirements on its own. Its development environment and integration with the hospital systems needs to be carefully designed as well.

To take this idea further, consider the previous scenario from the perspective of the usability of the language in the larger context of the hospital. Specifically, consider the nurse's view of the system. If the nurse is not notified of the change to the protocol, then he or she will receive a task he or she was not expecting, violating the **predictability** requirement. However, this is likely not a linguistic problem: Instead, this violation lives entirely in the integration of POP-PL with its environment.

Considering POP-PL from the nurses perspective does, however, lead to a new requirement that is likely linguistic. Consider the following instruction from the opioid prescription in Belknap et al. (2008): "If [the patient is] unarousable or respiration depressed, (e.g., resp. rate less than 6/min or O2 saturation less than 92%), then STOP **Fentanyl** & give **Naloxone(Narcan)** 0.1 mg i.v. every 2–5 mins up to 4 times until awake. Notify physician after giving first dose of naloxone." This order is meant to counteract an overdose of opioids without leaving the patient in pain by giving frequent small does of the antidote to fentanyl. A naive implementation of this loop would pester the nurse with a task multiple times within that 5-minute span, during a critical time when the nurse cannot afford to be distracted. This leads to a new design requirement:

Unburdensome and Alarm Fatigue. The language must allow for prescriptions that, while detailed, do not cause a high overhead for the humans executing the prescription. In the above example, it may be sufficient to give "every 2-5 mins" part of the order as one task instead of issuing many tasks. However, if this is done, then the program may not know what the patient's state is, or if the task has been delayed or missed, until it is too late to escalate the intervention.

The associated error condition is, again, alarm fatigue. A language that expresses tasks at too low a level may greatly increase the number of notifications clinicians receive in their day-to-day activities, a burden that is already excessive. If too many notifications for tasks are generated, then clinicians will likely ignore the output of POP-PL to manage their mental load and fall back to their current methods of keeping track of tasks, which would negate any benefits POP-PL could bring.

11.1 Future Evaluation

The current evaluation of POP-PL has several limitations. For one, it may not generalize to future versions of POP-PL, especially if the language changes radically. This is a classic problem with usability studies. The Cognitive Dimensions framework (Green 1989) was designed to tackle this problem by providing a generalizable framework for studying programming languages on several orthogonal dimensions. It seems likely that POP-PL's design requirements could be mapped to these dimensions; for example, **modifiablility** maps closely to the Viscosity dimension. Evaluating POP-PL against this framework, using existing methodology such as Blackwell and Green (2000), will give us a way to evaluate against future versions of POP-PL. In addition, a comparative study like Stefik et al. (2011) will help identify areas for and show improvement in the language over time.

Beyond notation, a more thorough evaluation of how the syntax and semantics work together is needed. There remains a need to evaluate if the mental model the language evokes is the same as the clinician's mental model of the prescription and how it will work in the hospital. This is necessary to meet the requirements of **unambiguity** and **predictability**. If **predictability** is not obtained, then the requirement of **adaptability** cannot be met. Such an evaluation would need to have both case study (to examine how the model acts with respect to clinicians in detail to help refine the design) and formal experimentation.

Making this evaluation more difficult is the fact that clinicians in the hospital have different mental models of how the system works depending on their clinical role. For instance, Singer et al. (2009) found that nurses felt deficiencies in the hospital infrastructure more than physicians. Listyowardojo et al. (2011) found that physicians perceived more of an institutional commitment to safety than nurses or lab workers. A study focusing on these differences and how they affect the perception and execution of orders is crucial to meet the requirements of **umambiguity** and **predictability**.

We expect to evaluate whether physicians and other prescribers are capable of independently writing and debugging prescriptions in POP-PL without the assistance of programmers. A new discipline of prescription validation will emerge, where development of POP-PL prescriptions will proceed in parallel with the clinical trials of new medical tests and treatments. When made generally available for use in routine clinical care, these clinical trial prescriptions will inform the creation of library routines that clinicians can invoke. We also expect that prescribers will stitch together library routines and existing patterns and abstractions to create a customized plan of care for each patient.

We also expect to evaluate the impact of a future, more mature, version of POP-PL on the risk of medical error and patient injury. Such evaluations of POP-PL's effect on processes and patients will need to look at several aspects of the system: Can POP-PL detect errors as they arise? Can POP-PL mitigate these errors when detected? Does adding POP-PL to the system introduce new errors? These questions can be first explored in realistic medical simulation environments. This will help to validate POP-PL in preparation for deployment in clinical trials.

When evaluating the detection and mitigation of errors, different evaluations will be needed for different kinds of error. For example, evaluating delayed reactions could be done by running POP-PL programs in simulation against logs derived from the medical administration record, and evaluation of forgotten monitoring could be done by having prescribers write prescriptions for patients pulled from the medical record.

Finding and evaluating errors introduced by POP-PL will likely require more involved experimentation, as errors could arise from secondary or tertiary effects of POP-PL in systems that it does not directly touch. Such evaluations would require both directed user studies with the different groups of clinicians at the hospital as well as POP-PL programs being written and run in a simulation of a hospital environment, where only the patient is fake.

12 CONCLUSION

This article connects the problem of medical error to a lack of an executable and clinician-usable specification of algorithmic medical protocols. From this, it generates a set of design requirements for such a tool by tying each requirement to a reduction in a specific kind of medical error. It then describes an evaluation of a prototype language based on that design. The evaluation found that, while the language is flawed, it appears that making a usable clinical language is possible. The understanding of these flaws gives a path forward in our efforts to design and implement a better prescription programming language.

With this path to making a better prescription programming language at our feet, we envision a future where prescriptions can be implemented as programs. Once prescriptions have been implemented as such, the programming language research communities can bring decades of tools and techniques for addressing software bugs to bear on the problem of medical error. Imagine a world where a type-system prevents a patient's weight in kilograms from being used with dosing instructions in pounds, where static analysis prevents coadministration of incompatible drugs to a patient, where a regression test suite prevents buggy modifications to a program as it is ported from one hospital to another. We expect that the full range of technology that the programming language community has already created—and will create—can find application in health care to reduce medical error, increase treatment efficacy, and improve patient outcomes. This work is our attempt to take the next step towards lifting the purely mechanistic cognitive burdens of patient care from clinicians, freeing their time to truly care for their patients.

ACKNOWLEDGMENTS

We thank Asumu Takikawa, Matthias Felleisen, Jesse Tov, Max New, Jonathan Schuster, Daniel Feltey, Shu-Hung You, and Christos Dimoulas for their feedback on drafts of this article. We also thank our excellent reviewers for their critical and constructive feedback, which has greatly improved this work.

REFERENCES

Apple Computer, Inc. 1988. *Hypercard Script Language Guide: The Hypertalk Language.* Addison-Wesley.

S. M. Belknap, H. Moore, S. A. Lanzotti, P. R. Yarnold, M. Getz, D. L. Deitrick, A. Peterson, J. Akeson, T. Maurer, R. C. Soltysik, G. A. Storm, and I. Brooks. 2008. Application of software design principles and debugging methods to an analgesia prescription reduces risk of severe injury from medical use of opioids. *Nature Clin. Pharmacol. Therapeut.* 84, 3 (2008), 385–392.

Steven M. Belknap. 1991. The Chicago Kinetic Simulator. *Math. J.* 1, 4 (1991), 68–86.

A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Curr, G. Kadoda, M. S. Kutar, C. L. Nehaniv, M. Petre, C. Roast, C. Roes, A. Wong, and R. M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of Cognitive Technology: Instruments of Mind.*

Alan F. Blackwell and Thomas R. G. Green. 2000. A cognitive dimensions questionnaire optimised for users. In *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group.*

Frederic Boussinot and Robert De Simone. 1991. The Esterel language. In *Proceedings of the Institute of Electrical and Electronics Engineers.*

T. Brus, M. C. J. D. van Eekelen, M. van Leer, and M. J. Plasmeijer. 1987. CLEAN—A language for functional graph rewriting. In *Proceedings Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, pp. 364–384.

Chrisian J. Callsen and Gul Agha. 1994. Open heterogeneous computing in ActorSpace. *J. Parallel Distrib. Comput.* 21 (1994), 289–300.

Bin Chen, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. 2006. Automatic fault tree derivation from Little-JIL process definitions. In *Proceedings of the Conference on Software Process Change (SPW'06).*

Carlo Combi, Mauro Gambini, Sara Migliorini, and Roberto Posenato. 2012. Modeling temporal, data-centric medical processes. In *Proceedings of the SIGHIT International Health Informatics Symposium.* 141–150.

Committee on Identifying and Preventing Medication Errors. 2007. *Preventing Medication Errors: Quality Chasm Series.* National Academies Press.

Maria Cvach. 2012. Monitor alarm fatigue: An integrative review. *Biomed. Instrum. Technol.* 46, 4 (2012), 268–277.

Division of Endocrinology, Department of Medicine and Multidisciplinary ICU Committee. 2004. Guideline for Intravenous Insulin Infusion in the Adult ICU Patient. Retrieved from http://www.hospitalmedicine.org/CMDownload.aspx?ContentKey=046c625a-839f-4731-beb8-5149c4c8f978&ContentItemKey=262ad349-0970-45ce-89d7-37de88728c7c.

R. Filik, K. Purdy, A. Gale, and D. Gerret. 2006. Labeling of medicines and patient safety: Evaluating methods of reducing drug name confusion. *Hum. Factors* 48, 1 (2006), 39–47.

Matthew Flatt and PLT. 2010. Reference: Racket. PLT, TR-1. Retrieved from http://racket-lang.org/tr1/.

George W. Furnas. 2000. Future Design Mindful of the MoRAS. *Hum.-Comput. Interact.* 15, 2–3 (2000), 205–261.

David M. Gaba, Mary Maxwell, and Abe DeAnda. 1987. Anesthetic mishaps: Breaking the chain of accident evolution. *Anesthesiology* 66, 5 (1987), 670–676.

Tony Garnock-Jones, Sam Tobin-Hochstadt, and Matthias Felleisen. 2014. The network as a language construct. In *Proceedings of the European Symposium on Programming (ESOP'14).*

David Gerrett, Alastair G. Gale, Iain T. Darker, Ruth Filik, and Kevin J. Purdy. 2009. Tall Man Lettering: Final report of the use of tall man lettering to minimise selection errors of medicine names in computer prescribing and dispensing systems. Retrieved from http://www.connectingforhealth.nhs.uk/systemsandservices/eprescribing/refdocs/tallman.pdf.

Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc.

T. R. G. Green and M. Petre. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput.* 7 (1996), 131–174.

T. R. G. Green and M. Petre. 1992. When Visual Programs are Harder to Read than Textual Programs. In *Proceedings of the 6th European Conference on Cognitive Ergonomics, Human-Computer Interaction: Tasks and Organisation.*

Thomas R. G. Green. 1989. Cognitive Dimensions of Notations. *People and Computers V.* 433–460.

Yehuda Handelsman, Jeffrey I. Mechanick, Lawrence Blonde, George Grunberger, Zachary T. Bloomgarden, George A. Bray, Samuel Dagogo-Jack, Jaime A. Davidson, Daniel Einhorn, On Ganda, Alan J. Garber, Irl B. Hirsch, Edward S. Horton, Faramarz Ismail-Beigi, Paul S. Jellinger, Kenneth L. Jones, Lios Jovanovič, Harold Lebovitz, Philip Levy, Etie S. Moghissi, Eric A. Orzeck, Aaron I. Vinik, and Kathleen L. Wyne. 2011. American Association of Clinical Endocrinologists medical guidelines for clinical practice for developing a diabetes mellitus comprehensive care plan. *Endocr. Pract.* 17, 5 (2011), 826–831.

Kathleen A. Harder, John R. Bloomfield, Sue E. Sendelbach, Michele F. Shepherd, Pam S. Rush, Jamie S. Sinclair, Mark Kirschbaum, and Durand E. Burns. 2005. *Improving the Safety of Heparin Administration by Implementing a Human Factors Process Analysis.* Agency for Healthcare Research and Quality.

Patrice M. Healey and Edwin J. Jacobson. 1994. *Common Medical Diagnoses: An Algorithmic Approach.* Saunders.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial intelligence (IJCAI'73).* 235–245.

Richard Hillestad, James Bigelow, Anthony Bower, Federico Girosi, Robin Meili, Richard Scoville, and Roger Taylor. 2005. Can electronic medical record systems transform health care? Potential health benefits, savings, and costs. *Health Affairs* 24, 5 (2005), 1103–1117.

John T. James. 2013. A new, evidence-based estimate of patient harms associated with hospital care. *J. Patient Safety* 9, 3 (2013), 122–8.

Jan Martin Jansen, Rinus Plasmeijer, Pieter Koopman, and Peter Achten. 2010. Embedding a web-based workflow management system in a functional language. In *Proceedings of the Language Descriptions, Tools and Applications Conference (LDTA'10)*, 2010.

Spencer S. Jones, Robert S. Rudin, Tanja Perry, and Paul G. Shekelle. 2014. Health information technology: An updated systematic review with a focus on meaningful use. *Ann. Intern. Med.* 60, 1 (2014), 48–54.

Denis Kilmov and Yuval Shahar. 2013. iALARM: An intelligent alert language for activation, response, and monitoring of medical alerts. In *Proceedings of the Revised Selected Papers of the AIME 2013 Joint Workshop on Process Support and Knowledge Representation in Health Care.*

David A. Kindig. 1971. Some implications of patient-oriented health care. In *Proceedings of the Health Conference of the New York Academy of Medicine.*

Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Brad Myers, Mary Beth Rosson, Gregg Rothermel, and Susan Wiedenbeck. 2011. The state of the art in enduser software engineering. *ACM Comput. Surv.* 43, 3 (2011), Article No. 21.

Ross Koppel, Metlay Cohen, Brian Abaluck, Russell Localio, Stephen E. Kimmel, and Brian L. Strom. 2005. Role of computerized physician order entry systems in facilitating medication errors. *J. Am. Med. Assoc.* 293, 10 (2005), 1197–1203.

Christopher P. Landrigan, Gareth J. Parry, Catherine B. Bones, Andrew D. Hackbarth, Donald A. Goldmann, and Paul J. Sharek. 2010. Temporal trends in rates of Ppatient harm resulting from medical care. *New Engl. J. Med.* 363 (2010), 2124–2134.

Lucian L. Leape. 1994. Error in medicine. *J. Am. Med. Assoc.* 272, 23 (1994), 1851–1857.

Lucian L. Leape, Ann G. Lawthers, Troyen A. Brennan, and William G. Johnson. 1993. Preventing medical injury. *Qual. Rev. Bull.* 19, 5 (1993), 144–149.

Tita Alissa Listyowardojo, Raoul E. Nap, and Addie Johnson. 2011. Variations in hospital worker perceptions of safety culture. *Int. J. Qual. Health Care* 24, 1 (2011), 9–15.

Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. 2010. TurKit: Human Computation Algorithms on Mechanical Turk. In *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology.*

Laurence E. Mather and Christopher J. Glynn. 1982. The minimum effective analgetic blood concentration of pethidine in patients with intractable pain. *Br. J. Clin. Pharmacol.* 14, 3 (1982), 385–390.

Wilson C. Mertens, Stefan C. Christov, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil, Lucinda J. Cassells, and Jenna L. Marquard. 2012. Using process elicitation and validation to understand and improve chemotherapy ordering and delivery. *Joint Commis. J. Qual. Patient Safety* 38, 11 (2012), 497–505.

Gianpaolo Molino, Paolo Terenziani, Stefania Montani, Alessio Bottrighi, and Mauro Torchio. 2006. GLARE: A Domain-Independent System for Acquiring, Representing and Executing Clinical Guidelines. In *Proceedings of the American Medical Informatics Association Annual Proceedings (AMIA'06).*

Stuart B. Mushlin and Harry L. Greene II. 2010. *Decision Making in Medicine* (3rd ed.). Mosby.

Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing.* MIT Press, Cambridge, MA.

John F. Pane, Brad A. Myers, and Leah B. Miller. 2002. Using HCI Techniques to Design a More Usable Programming System. In *Proceedings Symposia on Human Centric Computing Languages and Environments.*

John F. Pane, Chotirat Ann Ratanamahatana, and Brad A. Myers. 2001. Studying the language and structure in nonprogrammers' solutions to programming problems. *Int. J. Hum.-Comput. Stud.* 54, 2 (2001), 237–264.

Mor Peleg, Aziz A. Boxwala, Omolola Ogunyemi, Qin Zeng, Samson Tu, Ronilda Lacson, Elmer Bernstam, Nachman Ash, Peter Mork, Lucila Ohno-Machado, Edward H. Shortliff, and Robert A. Greenes. 2000. GLIF3: The evolution of a guideline representation format. In *Proceedings American Medical Informatics Association (AMIA'00).* 645–649.

Donald D. Prince, Francis M. Bush, Stephen Long, and Stephen W. Harkins. 1994. A comparison of pain measurement characteristics of mechanical visual analogue and simple numerical rating scales. *Pain* 56, 2 (1994), 217–226.

Donald D. Prince, Patricia A. McGarth, Amir Rafii, and Barbara Buckingham. 1983. The validation of visual analogue scales as ratio scale measurments for chronic and experimental pain. *Pain* 17, 1 (1983), 45–56.

Gruia-Catalin Roman. 1985. A taxonomy of current issues in requirements engineering. *Computer* 4.

Massimo Ruffolo, Rosario Curia, and Lorenzo Gallucci. 2005. Process Management in Health Care: A System for Preventing Risks and Medical Errors. In *Proceedings of the Business Process Management.* 334–343.

Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology.*

E. M. Schimmel. 1964. The hazards of hospitalization. *Ann. Intern. Med.* 60, 1 (1964), 100–110.

Yuval Sharar, Silvia Miksch, and Peter Johnson. 1998. The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artif. Intell. Med.* 14, 1–2 (1998), 29–51.

M. E. Sime, T. R. G. Green, and D. J. Guest. 1972. Psychological evaluation of two conditional constructs used in computer languages. *Int. J. Hum.-Comput. Stud.* 5, 1 (1973), 105–113.

Sara J. Singer, David M. Gaba, Alyson Falwell, Shoutzu Lin, Jennifer Hayes, and Laurence Baker. 2009. Patient safety climate in 92 US hospitals: Differences by work area and discipline. *Medical Care* 47, 1 (2009), 23–31.

Hardeep Singh, Shrinidi Mani, Donna Espadas, Nancy Petersen, Veronica Franklin, and Laura A. Petersen. 2009. Prescription errors and outcomes related to inconsistent information transmitted through computerized order entry: A prospective study. *Arch. Intern. Med.* 169, 10 (2009), 982–989.

Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. 2011. An empirical comparison of the accuracy rates of novices using the Quorum, Perl, and Randomo programming languages. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools.*

Ian G. Stiell, R. Douglas McKnight, Gary H. Greenberg, Ian McDowell, Rama C. Nair, George A. Wells, Cristine Johns, and James R. Worthington. 1994. Implementation of the ottawa ankle rules. *J. Am. Med. Assoc.* 271, 11 (1994), 827–832.

The SPRINT Research Group. 2015. A randomized trial of intensive versus standard blood-pressure control. *New Engl. J. Med.* 373, 22 (2015), 2103–2116.

Samson W. Tu and Mark A. Musen. 1999. A flexible approach to guideline modeling. In *Proceedings of the American Medical Informatics Association Conference (AMIA'99).* 420–424.

W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. 2009. Declarative workflows: Balancing between flexibility and support. *Comput. Sci.-Res. Dev.* 23, 2 (2009), 99–113.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. 2003. Busisness Process Management: A Survey. In *Proceedings of the 2003 International Conference on Business Process Management.*

Washington Adventist Hospital. 2009. Weight-Based beparin orders. Retrieved from https://extranet.adventisthealthcare.com/LinkClick.aspx?fileticket=rroECsjCLnY%3D&tabid=649&mid=1813.

K. N. Whitley. 1997. Visual Programming Languages and the Empirical Evidence For and Against. *J. Vis. Lang. Comput.* 8, 1 (1997), 109–142.

Paul R. Yarnold and Robert C. Soltysik. 2004. *Optimal Data Analysis: A Guidebook With Software for Windows.* APA Books.

Paul R. Yarnold and Robert C. Soltysik. 2016. *Maximizing Predictive Accuracy.* ODA Books.

Received October 2016; revised October 2017; accepted February 2018