POP-PL: A Patient-Oriented Prescription Programming Language

Spencer P. Florence*,†† Burke Fetscher* Matthew Flatt† William H. Temps‡ Tina Kiguradze‡
Dennis P. West‡ Charlotte Niznik‡ Paul R. Yarnold§ Robert Bruce Findler* Steven M. Belknap‡,¶

*Northwestern University, Department of Electrical Engineering and Computer Science, USA
†University of Utah, School of Computing, USA
‡Northwestern University Feinberg School of Medicine, Department of Dermatology, USA
§Optimal Data Analysis LLC, USA
¶Northwestern University Feinberg School of Medicine, Department of Medicine, Division of General Internal Medicine and Geriatrics, USA
††Corresponding Author: spencer.florence@eecs.northwestern.edu

Abstract

Medical professionals have long used algorithmic thinking to describe and implement health care processes without the benefit of the conceptual framework provided by a programming language. Instead, medical algorithms are expressed using English, flowcharts, or data tables. This results in prescriptions that are difficult to understand, hard to debug, and awkward to reuse.

This paper reports on the design and evaluation of a domain-specific programming language, POP-PL, for expressing medical algorithms. The design draws on the experience of researchers in two disciplines, programming languages and medicine. The language is based around the idea that programs and humans have complementary strengths, that when combined can make for safer, more accurate performance of prescriptions.

We implemented a prototype of our language and evaluated its design by writing prescriptions in the new language and administering a usability survey to medical professionals. This formative evaluation suggests that medical prescriptions can be conveyed by a programming language's mode of expression and provides useful information for refining the language. Analysis of the survey results suggests that medical professionals can understand and correctly modify programs in POP-PL.

Categories and Subject Descriptors D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—specialized application languages; J.3 [*LIFE AND MEDICAL SCIENCES*]: Medical information systems

General Terms Design, Human Factors, Languages

Keywords DSL Design, Medical Programming Languages, Medical Prescriptions, Empirical Evaluation

GPCE'15, October.2015, Pittsburgh, Pennsylvania, USA.

ACM 978-1-4503-3687-1/15/10. . . $15.00.
http://dx.doi.org/10.1145/2814204.2814221

1. Prescribing Programs

Physicians and other prescribers are programmers in search of a programming language. They write and repeatedly modify complex, algorithmic prescriptions, some the length of a journal paper (Handelsman et al. 2011). Currently, prescribers use flowcharts, structured data tables, or natural language to express their prescriptions; and pharmacists, nurses, physicians, medical technicians, patients, caretakers, and medical devices perform the instructions in the prescription to the best of their understanding and ability. These instructions unavoidably contain errors and ambiguities, cause miscommunication, and sometimes exceed human capacity for reliable performance, any of which may result in patient injury or death. Previous attempts to "computerize" prescriptions have floundered because the algorithmic nature of prescriptions was not recognized. Singh et al. (2009) have shown how physicians struggle with the limited data-field entry programs that current Computerized Physician Order Entry (CPOE) and ePrescribing systems provide.

In response, we have designed, implemented, and tested a prototype for a prescription programming language. This domain-specific language, POP-PL, leverages the power of programming and program development to help physicians articulate, model, test, debug, and iteratively refine prescriptions. POP-PL is designed to express all of the instructions relating to the care of a patient, including drug therapy, diet, exercise, laboratory monitoring, imaging studies, etc.

The nature of prescriptions is more complex than it might first appear. Prescriptions require both detailed task accounting and human action. To handle this, POP-PL represents prescriptions as reactive programs that interact with a system of actors that perform healthcare–including patients, prescribers, other clinicians, clinics, hospitals, and, ultimately, entire healthcare systems. Based on our observations, clinicians readily grasp this conceptualization of healthcare; indeed, this is how the clinician-researchers on our team think of their healthcare work. The prescription itself views the health care system context as a stream of messages describing events and tasks. In response to particular messages or patterns of messages, a prescription issues instructions to command, control, and coordinate relevant actors, including other prescriptions, medical devices, or nurses. Unlike other systems for managing humans to perform complex tasks—e.g. Jabberwocky (Ahmad et al. 2011) and Automan (Barowy et al. 2012)—POP-PL is not a system for distributed computation. Instead, its focus is on remote actuation, which treats humans as actuators for performing the primitive operations written in the prescription program.

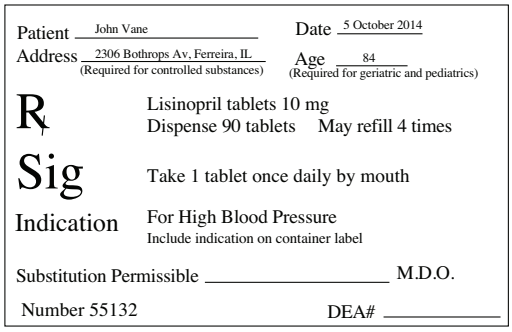
Patient John Vane Date 5 October 2014
Address 2306 Bothrops Av, Ferreira, IL (Required for controlled substances) Age 84 (Required for geriatric and pediatrics)
℞ Lisinopril tablets 10 mg
Dispense 90 tablets May refill 4 times
Sig Take 1 tablet once daily by mouth
Indication For High Blood Pressure
Include indication on container label
Substitution Permissible _____ M.D.O.
Number 55132 DEA# _____

Figure 1: Lisinopril Prescription

We have analyzed the structure and function of numerous existing English-language prescriptions and clinical protocols to inform the creation of POP-PL. This paper reports on a formative evaluation, in the form of a survey of medical professionals. The survey assesses their ability to understand and modify POP-PL programs after a 10 minute training session. Our results suggest that prescribers and clinicians can understand programs in this language without much difficulty.

This paper presents the concrete syntax of our language, sample prescriptions, and a semantics model. Section 2 provides some of the medical background on our motivation, and on the nature of prescriptions. Section 3 explains the design principles and requirements of POP-PL, and provides some detail on how they are met. Section 4 works through an example prescription written in POP-PL. Section 5 discusses POP-PL's formal model of evaluation. Section 6 describes how the model maps to the concrete syntax of the language. Section 7 presents an empirical formative study of how well prescribers can understand the language. Section 8 covers related work, and section 9 concludes.

2. The Medical World

To fully understand this prescribing language, one must understand a little about the medical world and, in particular, the nature of prescriptions and the difficulty of mitigating medical errors.

2.1 What is a Prescription?

The nature and complexity of prescriptions may not be immediately evident to the casual observer. Even seemingly simple prescriptions are more than just lists of drugs and dosages, and contain hidden complexities. 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". But this is not 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:

- 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."

All of these instructions are part of the prescription. So, to be precise: a prescription comprises *the entire set of healthcare instructions* that are issued about 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 be for a patient, caregiver, nurse, pharmacist, or physician.

Therefore, a prescription is a program whose instructions govern the plan of health care for an individual patient (Belknap et al. 2008). 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.

2.2 Why is Reducing Error Important?

Over the past half-century, medical research has consistently identified medical error as a major impediment to the practice of patient-oriented[1] 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 still contributes to roughly one-sixth of all deaths in the United States—approximately 440,000 people per year (James 2013). Unfortunately, Landrigan et al. (2010) found that there had been little improvement in the rate of medical errors in the previous decade.

3. The Design of POP-PL

Based on our clinical experience, we have derived a primary design principle and several requirements for POP-PL. This section covers these principles and requirements, then presents the design concepts POP-PL uses to meet them.

3.1 Design Principle

Medical professionals and software have complementary and synergistic strengths. Software can reliably perform repetitive, tedious tasks such as searching large data sets, continuously monitoring for out-of-range values, precisely measuring time intervals, task auditing, and general bookkeeping. In contrast, medical professionals are naturally good at obtaining and analyzing patient narratives, making and testing hypotheses about diagnosis, 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. Both kinds of skills are crucial for effective patient care.

POP-PL is designed to enhance this synergy. By giving prescribers a mechanism to describe tedious tasks and let a machine carry them out, we hope to avoid error, while leaving high-level control of health-care in the hands of the prescriber.

3.2 Requirements

Providing medical care to a patient requires that many small tasks be issued, scheduled, and performed. Individually, these tasks are usually not difficult. Problems arise, however, because there are many tasks for each patient, multiple patients for each clinician, and competition among these tasks and actors for multiple constrained resources. The precise set of tasks can vary in small or large ways between patients that have similar conditions, and the sheer number of these tasks can sometimes overwhelm healthcare workers.

[1] Patient-oriented medicine has the goal of improving or maintaining health while being responsive to the patient's needs and values (Kindig 1971).

The remainder of this section describes general categories of task-related error and how they have become POP-PL design requirements. Specific instances of all of these general categories were witnessed by the team who developed a prescription for opioids (Belknap et al. 2008); indeed, the development of that prescription was the germination of many of the ideas presented here.

Forgotten Monitoring. Safe administration of dangerous drugs typically requires a commensurate form of monitoring. Because the specifics of the monitoring may vary across patients, however, the monitoring is typically not explicitly coupled with the drug, even when they are commonly ordered together. Accordingly, monitoring may be omitted or removed, leading to bad outcomes. To fix this, a programming language must provide an easy-to-use mechanism to build abstractions over common patterns of orders.

Alarm Fatigue. Even when monitoring is ordered, an alert may go unnoticed or unaddressed, presumably due to alarm fatigue. Many devices produce many noises in the hospital setting and false alarms are common. Cvach (2012) found that clinicians may have to deal with as many as 700 monitor alarms per patient per day. To fix this, a programming language must generate better targeted alarms. A program should take the entire medical context into account before sounding an alert and then generate a limited number of messages targeted directly to those who need to perform the required tasks.

Delayed Reaction. With some prescriptions, even when monitoring is ordered, and dangerous situations are recorded, noticed, and understood, life saving measures are not taken. In these cases the active prescription does not contain an order that provides specific instructions for when these measures are to be taken and how to perform them correctly (e.g., precise names and dosage information for antidotes for dangerous drugs). For example, if the nurse is not told the dose of antidote to give the patient, it becomes difficult to perform even this seemingly simple lifesaving intervention correctly. In such cases, the nurse must contact the physician or other prescriber, which may result in dangerous delay. To fix this, a programming language for prescriptions must provide a way for the prescription to identify dangerous situations and issue the precise set of tasks required for such life-saving measures.

Task Overload. The tempo required for performance of medical tasks is unremitting and can be overwhelming. Schubert et al. (2013) found that 98% of nurses had to drop at least one task in the span of a week. The capacity of humans for vigilance is limited. To fix this a programming language must reduce the overall number of, or simplify the existing, human-dependent tasks (e.g., by automating simple computations or bookkeeping), and help track missing tasks to ensure critical tasks are not forgotten.

3.3 Concepts

The design of POP-PL handles these requirements by dividing its world model into actors. Each entity in the clinical setting (prescriptions, nurses, pharmacists, patients, programmable infusion pumps, etc.) is modeled as an actor. Each actor subscribes to messages about their patients. All messages and actions are recorded in a log, which forms the history of the entire healthcare system. The log is, in principle, a time-ordered list of all events that have occurred and tasks that have been issued with respect to a patients 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.

4. An Example POP-PL Program

To illustrate POP-PL's design concepts, figure 2 contains an example POP-PL program for administering an anticoagulant called heparin in a hospital setting. The program is a partial translation of the Washington Adventist Hospital (2009) protocol. 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. Heparin dosing is challenging because both the duration and intensity of heparin's effect change disproportionately and unpredictably with drug dose. If the heparin dose is too low, treatment may fail and the patient may have a catastrophic embolism to the brain or lungs; too high, and the patient may hemorrhage to death. To maintain the correct amount of anticoagulation, the protocol requires frequent monitoring of the heparin's effect and adjustment of its dose.

The first line of the program tells 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 Jessie Brown VA Medical Center.

When a prescriber starts this program, it initially—lines 5 to 7—sends a message to generate a task for the nurse on duty to give a one-time 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 continuous intravenous infusion of heparin at 18 units/kg of body weight per hour.

The 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 a measurement of the patient's blood called the activated partial thromboplastin time, or aPTT. This measurement is the number of seconds it takes for blood to clot under special lab conditions. 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 a 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 aPTT result, 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`, 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 results within this range come in the program issues no orders. 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` 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 insist that event happened twice. There is some syntactic trickery[2] here: if there are less than two aPTT values present the six 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.

[2] This trickery backfired. We revisit the issue in section 7.

```
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]
```

Figure 2: Example of a POP-PL Program

The `every` form takes two expressions: a time 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 both 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 prints out the messages that would be sent from the prescription in a deployed version. It also prints a prompt where the user can submit messages to simulate responses from the world. This is POP-PL's version of a REPL.

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

```
[checkaptt]
[start 18 units/kg/hour heparin iv]
[givebolus 80 units/kg heparin iv]
```

which ask for the initial aPTT reading, and tell the nurse to start the IV and give the initial bolus.[3] We can then enter messages at the REPL to see how the program responds. For example, we send an aPTT response of 46 seconds by typing the text following the ">":

```
> aPTTResult 46 seconds
[increase heparin 1 unit/kg/hour]
[givebolus 40 units/kg heparin iv]
```

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

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

[3] The outgoing messages are downcased because POP-PL is case insensitive. See section 6.

```
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)
o ::= time-of | most-recent | ...
```

Figure 3: POP-PL Model Syntax

Lines 30 and after contain unit tests for the program. They are written just like REPL interactions; a message in brackets denotes an outgoing message, and a message preceded by a ">" is a message sent to the prescription. Unlike REPL interactions, they are run and the results are checked. If they fail, the program prints out a message saying that the test cases failed.

5. A Formal Model for Evaluation

To formally model how POP-PL behaves, we define an evaluator that handles one new message at a time. This evaluator takes the current history of the hospital (including the new message), and an actor, which is nothing more than a list of handlers. These handlers can send messages to the outside world, add new handlers to the actor, or remove existing ones. This produces a set of outgoing messages and a new set of handlers for the actor.

In this section we present the formal model for this evaluator. We then extend this 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 named handlers from the state, and one for sending messages to the outside. In addition it has three primitive data types: messages (m), logs, and void. We use the λ-calculus here not because POP-PL is inherently 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. We will describe how the language's concrete syntax compiles to this abstract model in the next section.

5.1 Language Basics

A log is a complete representation of history. 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, 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.

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, we choose heartbeats for simplicity.

5.2 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 that the prescription is subscribed to appears, the evaluator is invoked with the new state and a log containing the new message.

```
EVAL : H × log → H × (m ...)

s ::= ⟨e, H, (m ...)⟩
H ::= (h ...)
h ::= ⟨name, (λ (x) e)⟩
v ::= m | log | void | (λ (x ...) e)
S ::= ⟨E, H, (v ...)⟩
E ::= (add name E) | (begin E e) | (send E)
    | (v ... E e ...) | (o v ... E e ...)
    | (if0 E e e) | []
```

$$\langle E[(\texttt{send } m_n)], H, (m \ldots)\rangle \qquad \text{[SEND]}$$
$$\longrightarrow \langle E[\texttt{void}], H, (m_n\ m \ldots)\rangle$$

$$\langle E[(\texttt{add } name_a\ v)], (h \ldots), (m \ldots)\rangle \qquad \text{[ADD]}$$
$$\longrightarrow \langle E[\texttt{void}], (\langle name_a, v\rangle\ h \ldots), (m \ldots)\rangle$$
$$\text{where } (\langle name, v_h\rangle \ldots) = (h \ldots),\ name_a \notin (name \ldots)$$

$$\langle E[(\texttt{remove } name)], H, (m \ldots)\rangle \qquad \text{[REM]}$$
$$\longrightarrow \langle E[\texttt{void}], (h_1 \ldots h_2 \ldots), (m \ldots)\rangle$$
$$\text{where } (h_1 \ldots \langle name, v\rangle\ h_2 \ldots) = H$$

$$S[((\lambda\ (x \ldots)\ e)\ v \ldots)] \qquad [\beta v]$$
$$\longrightarrow S[e\{x := v, \ldots\}]$$

$$S[(\texttt{begin } v\ e)] \longrightarrow S[e] \qquad \text{[BEGIN]}$$

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

$$S[(\texttt{if0 } 0\ e_1\ e_2)] \longrightarrow S[e_1] \qquad \text{[IF0]}$$

$$S[(\texttt{if0 } v\ e_1\ e_2)] \longrightarrow S[e_2] \qquad \text{[IF!0]}$$
$$\text{where } v \neq 0$$

Figure 4: POP-PL Model of Evaluation

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 inputs H_0 (the current handlers) and log (the current view of history) EVAL starts computation in the machine state $\langle(\texttt{h } log), H_0, ()\rangle$ where h is the first handler in H_0.

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. The remaining rules are standard.

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.

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

Figure 5: Model for the querying DSL

5.3 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: 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; 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.

$$\textsc{cut} : (\mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

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.

$$\textsc{filter} : (\mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

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

$$\textsc{get-consec} : (\mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

The GET-CONSEC operation returns the longest prefix of the log where the predicate holds. This operation is similar in form to CUT, but its purpose is very different. It is meant to ask how much something has happened without deviation. For instance "How many blood tests had a glucose value of above 45?"

$$\textsc{subseq} : (\mathit{Message} \times \mathit{Message} \to \mathit{Bool}) \times \mathit{Log} \to \mathit{Log}$$

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 two 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}.

$$\textsc{length>=} : \mathit{Natural} \times \mathit{Log} \to \mathit{Log}\ \mathit{or}\ \mathit{Fail}$$

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 Fentanyl (Belknap et al. 2008): "Are there three consecutive pain scores over 8 cm?". 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?".[4] This translates to the abstract syntax in figure 6.

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

Figure 6: A query, in the syntax of the model

6. The Concrete Syntax

The concrete syntax for POP-PL is designed to abstract over the complex, unambiguous ideas doctors have, while still looking like the prescriptions doctors write today. In fact, the syntax is designed so that the language should be usable with little or no programming training on the part of the clinician. Thus, we make several unconventional choices to reflect the vagaries of how prescriptions are currently written and to ease a prescriber's transition into using POP-PL. 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, possibly with attached units. 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. 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, the language could determine exactly how many IV lines are needed to deliver all the drugs.

Case insensitivity. The syntax of POP-PL is case insensitive. The merits of case-sensitivity are debatable, but in a medical context it allows for Tall Man lettering (Filik et al. 2006). A common medical error is the confusion of two different 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. POP-PL allows physicians to use any capitalization they choose, and thus always use a clear form of the drug name.

Keyword Arguments. Functions in POP-PL allow variations on their keyword arguments. In addition to allowing keyword arguments in any order, arguments list keywords that are allowed as synonyms. 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 lets physicians pick whatever words they would use in plain English orders. This way of handling name parameters is inspired by Hypertalk (Apple Computer, Inc 1988).

[4] It is worth noting that medical professionals do not always arrive at the more precise meaning, and mistakes are made.

After. The `after` form corresponds to adding 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. The `since the 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.

7. Assessing the Language

Day to day, doctors do not write prescriptions as complicated as figure 2. The majority of day to day prescriptions consist of simple orders like "give Lisinopril, 10 mg every day", or invoking complex pre-written protocols like the one in figure 2, perhaps with some minor modifications.

We evaluated POP-PL on this day to day use case. We gave a usability survey to medical professionals designed to assess if they could read and modify prescriptions-as-programs, and to assess our language design to ferret out weak points.

Type	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

Figure 7: Demographics of the Survey

7.1 Survey Design

The survey began with a short explanation of a translated insulin protocol.[5] The participants were then presented with a survey containing the protocol in figure 2 without the test cases, and 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

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

7.2 Understanding the Demographics

The survey was given to 51 respondents. Each respondent had one of 7 levels 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 four-year post-graduate program. After medical school, many physicians undergo additional training in residency programs, typically three to seven 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, usually starting with a high-school degree, and occasionally a two-year residency, before becoming a post-training pharmacist. Advanced

[5] The script for the survey introduction, the full survey and the tallied results are available at `http://eecs.northwestern.edu/~sfq833/resources/blobs/pop-pl-GPCE-2015.tar.gz`

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%

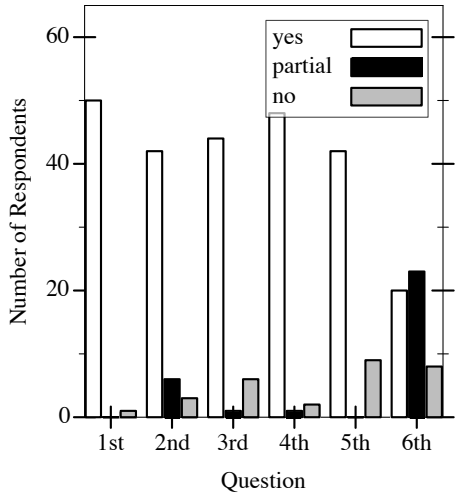

Figure 8: Survey Data

practice nurses (APNs) are nurses with additional post-graduate education; 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 of 0–10. 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 blank.

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 APN's completed the survey in a meeting scheduled for this purpose. The breakdown of the survey participants is in figure 7.

7.3 Analyzing the Data

Three graders evaluated each survey. From these, we calculated the inter-rater agreement, a score for the level of consensus between graders. The questions were graded with 3 possibilities: "yes", meaning the respondent answered the question completely correctly; "partial", meaning the respondent was neither completely correct nor completely incorrect; "no", meaning the respondent was completely incorrect. The results for each question are displayed both by percentage and graphically in figure 8.

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).

UniODA was performed to assess inter-rater agreement for all 18 pairings of three independent raters scoring six separate test questions (Yarnold 2014). 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.

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. And, 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.

7.4 Interpreting the Analysis

The correlation between perfect score and programming experience/comfort is discouraging. However from looking at the data in figure 8, it is clear 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.

From this we draw two conclusions. First, the querying DSL and its syntactic trickery 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 only needed to be in the effective range once, or that the test should be run on the faster schedule after two aPTT tests were in range. The first of these mistakes shows 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 the mistakes would seem to imply that something in the querying language is confusing. Given that experience with programming correlates to a perfect score, perhaps the querying language strays too far from the natural language doctors are used to and should be simplified, or at least hidden with some form of helper function.

Second, the high scores on the first five questions provide strong evidence that doctors can interpret and modify prescriptions written as programs with little training.

7.5 Anecdotes

In addition to the survey, we have collected several revealing comments from survey participants about POP-PL. Two survey participants asked if POP-PL would automatically update nurses' task lists. This illustrates a flaw in current systems, where new tasks are not automatically propagated to nurses. Physicians compensate for this flaw by verbally notifying a nurse when they have modified a patient's prescription, prompting the nurse to log in to the EMR and update their task list. Without this verbal communication between prescriber and other clinicians, a stale order may be performed in error or a new order may be neglected for several hours, either of

which may result in serious patient injury or death. POP-PL's focus on tasks and relative task-related sophistication suggests it naturally helps with this problem.

Another survey participant mentioned that POP-PL looked almost exactly like how she wrote free-text orders currently. Yet another participant mentioned that POP-PL would be useful for transmitting orders when engaged in telemedicine.

8. Related Work

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 now underway is 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.

Some research studies have shown that computerization of health care reduces medical error and patient injury, but the effect is often small and in some cases is nonexistent (Jones et al. 2014). 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 limited in expressiveness, they often include free-text instructions they cannot fit into 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 structured data fields, and 20% of these inconsistencies risked moderate or severe patient injury (Singh et al. 2009). There is little evidence that current attempts to computerize health care have resulted in significant improvement of patient safety in routine medical practice (Landrigan et al. 2010).

The work we describe here is inspired by POPA (Belknap et al. 2008), a protocol developed for management of severe pain in hospitalized patients. The authors of the protocol used software engineering and programming language ideas to develop and debug the protocol, which was expressed entirely on paper. The protocol was successful, reducing severe or fatal opioid-associated adverse drug events from an initial 3 to 7 per month to 0 per month.

Medical practitioners have long recognized the value of using algorithmic descriptions to codify other aspects of clinical practice. One of the most successful instances of algorithmic medicine is the Ottawa Ankle Rules (Stiell et al. 1994), an efficient algorithm for distinguishing between a broken ankle and a sprain. Distinguishing between a sprain and a break is something that can be done with a physical inspection with high accuracy, but only if you know precisely what to look for. It has led to better outcomes at lower cost and is now widely used. There have also been entire textbooks that collect medical algorithms and explain how to use them (Healey and Jacobson 1994; Mushlin and Greene 2010). These algorithms are, however, expressed using flow-charts and tree-like decision diagrams and thus fail to bring the full power of algorithmic expression that programming languages have to offer because the underlying programming language of flow-charts and decision trees is inexpressive. Our work is the logical next step: bringing programming language know-how to design languages for the clear expression of medical algorithms.

Computer scientists have already made several attempts to improve clinical practice; these efforts have come largely from the AI community (Peleg et al. 2000; Sharar et al. 1998; Tu and Musen 1999). Most of these efforts, at a high level, consist of building ontologies and decision support systems for medical practice, in an attempt to make more accurate diagnoses, generate treatment plans, or detect hazards. Unlike these approaches, we are attempting to build on the established strengths of software, relieving clinicians from the tedious, error-prone tasks that compromise patient safety. We leave the challenging tasks of interacting with patients, gathering evidence, and diagnosing to people who are well-trained for it and, perhaps more importantly, love to do it.

Several other bodies of work have modeled hospitals as actor systems. For example, Belknap (1991) provides a system for simulating in 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."

As already mentioned, POP-PL is not a system for distributed computation; rather, it handles remote actuation. In principle this is similar to the iTasks (Jansen et al. 2010) task management language, which has been used to model the Netherlands' Coast Guard's SAR operations (Lijnse et al. 2011). However, iTasks distinguishes the end user from the programmer (Lijnse et al. 2012), whereas POP-PL aims to have prescribers writing programs.

Hewitt et al. (1973) originated actor model. The delivery of messages to relevant parties we use is similar to Actorspaces from Callsen and Agha (1994).

9. Conclusion

Prescription errors are strikingly similar to software bugs; both are often shallow and regrettably common. The programming language research community has tools and techniques for reducing the number of software errors. Now that we know that prescriptions are programs, these techniques have clinical applications. Imagine a world where a type-system prevents a patient's weight in kilograms from being used with dosing instructions in pounds, where a static analysis prevents coadministration of incompatible drugs to a patient, or where a regression test suite prevents buggy modifications to a program as it is ported from one hospital to another. Our hope is that the full range of programming languages technology can find application in health care to reduce medical error, improve treatment efficacy, and improve patient outcomes. This paper is our attempt to take the next step towards a future where medical error is surprising and unusual rather than the norm.

Acknowledgements

We would like to thank Vincent St-Amour, Asumu Takikawa, Matthias Felleisen, Jesse Tov, Max New, and Jonathan Schuster for their feedback on drafts of this paper.

We would also like to thank the NSF for their support.

References

Salman Ahmad, Alexis Battle, Zahman Malkani, and Sepandar D. Kamvar. The Jabberwocky programming environment for structured social computing. In *Proc. User Interface Software and Technology (UIST)*, pp. 53–64, 2011.

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

Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor. AUTOMAN: a platform for integrating human-based and digital computation. In *Proc. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pp. 639–654, 2012.

SM Belknap, H Moore, SA Lanzotti, PR Yarnold, M Getz, DL Deitrick, A Peterson, J Akeson, T Maurer, RC Soltysik, GA Storm, and I Brooks. Application of software design principles and debugging methods to an analgesia prescription reduces risk of severe injury from medical use of opioids. *Nature: Clinical Pharmacology & Therapeutics* 84(3), pp. 385–392, 2008.

Steven M. Belknap. The Chicago Kinetic Simulator. *The Mathmatica Journal* 1, pp. 68–86, 1991.

Chrisian J. Callsen and Gul Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computer* 21, pp. 289–300, 1994.

Committee on Identifying and Preventing Medication Errors. Preventing medication errors: Quality chasm series. *National Academies Press*, 2007.

Maria Cvach. Monitor alarm fatigue: An integrative review. *Biomedical Instrumentation & Technology* 46, pp. 268–277, 2012.

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

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

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

David Gerrett, Alastair G. Gale, Iain T. Darker, Ruth Filik, and Kevin J. Purdy. 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. 2009. `http://www.connectingforhealth.nhs.uk/systemsandservices/eprescribing/refdocs/tallman.pdf`

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. American Association of Clinical Endocrinologists medical guidelines for clinical practice for developing a diabetes mellitus comprehensive care plan. *Endocrine Practice* 17, 2011.

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

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

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

John T. James. A new, evidence-based estimate of patient harms associated with hospital care. *Journal of Patient Safety* 8, pp. 122–128, 2013.

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

Spencer S. Jones, Robert S. Rudin, Tanja Perry, and Paul G. Shekelle. Health information technology: An updated systematic review with a focus on meaningful use. *Annals of Internal Medicine* 106(1), pp. 48–54, 2014.

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

Ross Koppel, Metlay Cohen, Brian Abaluck, Russell Localio, Stephen E. Kimmel, and Brian L. Strom. Role of computerized physician order entry systems in facilitating medication errors. *The Journal of the American Medical Association (JAMA)* 293, pp. 1197–1203, 2005.

Christopher P. Landrigan, Gareth J. Parry, Catherine B. Bones, Andrew D. Hackbarth, Donald A. Goldmann, and Paul J. Sharek. Temporal trends in rates of patient harm resulting from medical care. *New England Journal of Medicine* 363, pp. 2124–2134, 2010.

Lucian L. Leape. Error in medicine. *The Journal of the American Medical Association (JAMA)* 272, pp. 1851–1857, 1994.

Bas Lijnse, Jan Martin Jansen, Ruud Nanne, and Rinus Plasmeijer. Capturing the netherlands coast guards SAR workflow with iTasks. In *Proc. International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, 2011.

Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. Incidone: A task-oriented incident coordination tool. In *Proc. International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, 2012.

Stuart B. Mushlin and Harry L. Greene II. Decision Making in Medicine (Third Edition). Mosby, 2010.

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. GLIF3: The evolution of a guideline representation format. In *Proc. American Medical Informatics Association (AMIA)*, pp. 645–649, 2000.

E. M. Schimmel. The hazards of hospitalization. *Annals of Internal Medicine* 60, pp. 58–63, 1964.

M Schubert, D Ausserhofer, M Desmedt, R Schwendimann, E Lesaffre, B Li, and S De Geest. Levels and correlates of implicit rationing of nursing care in Swiss acute care hospitals—a cross sectional study. *International Journal of Nursing Studies* 50, pp. 230–239, 2013.

Yuval Sharar, Silvia Miksch, and Peter Johnson. The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine* 14, pp. 29–51, 1998.

Hardeep Singh, Shrinidi Mani, Donna Espadas, Nancy Petersen, Veronica Franklin, and Laura A. Petersen. Prescription errors and outcomes related to inconsistent information transmitted through computerized order entry: a prospective study. *Archives of Internal Medicine* 169, pp. 982–989, 2009.

Ian G. Stiell, R. Douglas McKnight, Gary H. Greenberg, Ian McDowell, Rama C. Nair, George A. Wells, Cristine Johns, and James R. Worthington. Implementation of the ottawa ankle rules. *The Journal of the American Medical Association (JAMA)* 271, pp. 827–132, 1994.

Samson W. Tu and Mark A. Musen. A flexible approach to guideline modeling. In *Proc. American Medical Informatics Association (AMIA)*, pp. 420–424, 1999.

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

Paul R. Yarnold. How to assess inter-observer reliability of ratings made on ordinal scales: Evaluating and comparing the Emergency Severity Index (Version 3) and Canadian Triage Acuity Scale. *Optimal Data Analysis* 3, pp. 42–49, 2014.

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