A Semantics-First Approach to Safe Clinical Decision Support

Manasvi Saxena
msaxena2@illinois.edu
University of Illinois Urbana Champaign

1 Introduction

Preventable Medical Errors (PMEs) characterized by incorrect intended treatment, or incorrect executions of intended treatment present a significant challenge in Healthcare [1]. According to a seminal report on the subject [2], in 1997, between 44,000 and 98,000 deaths were estimated to have been caused by PMEs in the United States alone. A more recent study analyzed data from the eight-year period between 2000 and 2008, and estimated that in 2013, the number of deaths caused by PMEs was more than 250,000, making PMEs the third-leading cause of death in the United States [3]. The adverse effects of PMEs extend beyond patient outcomes. One study estimated the financial burden of PMEs to the United States to be 19.5 billion dollars in 2008 [4]. According to the authors of [1], PMEs caused psychological effects such as anger and guilt in healthcare providers (HCPs), adversely impacting their mental health.

One strategy to mitigate PMEs is to utilize evidence-based statements published by hospital and medical associations that codify recommended interventions for various clinical scenarios called Best Practice Guidelines (BPGs) [5]. High quality guidelines are routinely updated to account for results from clinical trials and advances in medicine, and make the latest diagnosis and treatment information accessible to providers [6].

While BPGs have the potential to reduce medical errors, their effectiveness hinges on the adherence of healthcare providers to them. For example, consider Advanced Cardiac Life Support (ACLS): a BPG published by the American Heart Association (AHA) for management of a life threatening condition called in-hospital cardiac arrest (IHCA) [7, 8]. Studies suggest that management of IHCA in 30% of adult, and 17% of pediatric cases deviates from the AHA-prescribed BPG, resulting in worse patient outcomes [9, 10, 11, 12, 13].

While BPG-adherence is difficult to achieve in practice [14, 15], integrating BPGs with existing patient care-flow, and making them readily-accessible when required can improve adherence [16]. To this end, hospitals commission computerized Decision Support Systems (CDSSs) that codify BPGs and support HCPs with situation-specific advice. Such systems have been shown to improve BPG-adherence [17, 18], and evidence from multi-center clinical trials suggests that they reduce PMEs [19, 20]. Thus, guideline-based CDSSs are now considered imperative to the future of medical decision making in general [21].

But despite their transformative promise, wider adoption of CDSSs is hindered by the challenges encoutered during their development:

Specification-Implementation Gap: Typically, to develop a CDSS, domain experts in medicine collaborate with software developers to translate medical knowledge encoded in a BPG to a computable medium, referred to as the CDSSs's knowledge-base. Additional software is then added to connect components such

as such as *medical sensors* and *User Interfaces* to complete the CDSS. Thus, the BPG acts as a functional specification for the knowledge-base, i.e. the implementation. However, as medical knowledge in a BPG is complex, it's possible for it to be inaccurately captured in the knowledge-base, or, for the implementation to diverge from the specification. Morover, as BPGs evolve to reflect latest clinical evidence, the knowledge-base has to be updated accordingly, further incresing the likelihood of a *specification-implementation* divergence.

Complexity: BPGs often involve *concurrent* workflows, with complex inter-workflow interactions. Consider the example of management of sepsis in pediatric cases from Fig. 1a. Sepsis is a life threatening condition characterized by the body's extreme's immune response to an infection. Once a patient is flagged as potential septic, several workflows, such as screening the patient for septic shock and administering fluids, antibiotics and inotropes have to be performed simultaneously, where the type of inotrope to be administered is influenced by whether the patient is cold or warm shock. Any CDSS must be capable of handling such concurrent workflows, and their interactions.

Moreover, a CDSS also must interact with *diverse external agents* such as sensor for patient parameters and databases holding electronic health records. Satisfying such a diverse set of requirements adds to the challenge of building CDSSs.

Formal Analysis: CDSS are *safety-critical* - bugs can have life-threatening consequences. Thus, its desirable to have (a) execution engines (interpreters and compilers) with *correctness guarantees*, and, (b) suite of formal analysis tools (such as model checkers, deductive verifiers) to verify desired *correctness* properties. This need has already been recognized in existing literature [22, 23]. *Complexity* of CDSSs adds to the challenge of verifying CDSSs as tools must be able to handle both the *concurrency* and *external agent* aspects of CDSSs.

1.1 Proposed Approach

This proposal aims to systematically address aforementioned challenges using a *semantics-first* approach. At the core of this approach is a novel language for writing BPGs in a *computer interpretable* manner called Medi$\mathbb{K}$. Medi$\mathbb{K}$ builds on the existing state-of-art for modeling large concurrent systems and other CIG languages to enable expressing *complex* BPGs naturally. By being *comprehensible* to domain experts in medicine, Medi$\mathbb{K}$-based Computer Interpretable Guidelines (CIGs) can serve as both the non-executable BPG, and the knowledge-base of a CDSS, eliminating any *specification-implementation gap*. Morover, any changes to the CIG are reflected immediately in the CDSS.

By *semantics-first*, we mean that our proposed has a *complete, formal semantics* from which tools such as an interpreter, model checker, and deductive verifier are derived in a *correct-by-construction* manner, addressing the *formal analysis* aspect of building CDSSs. This also enable rapid incorporation feedback from medical domain experts, as once the formal semantics are updated, the tools evolve automatically.

In essence, our approach can be broken down into the following research questions (RQs):

RQ 1 (Language Design): BPGs can vary greatly by scope and purpose. For instance, consider differences between the BPGs for managing cardiac arrest [24] and sepsis [25]. The BPG for cardiac arrest can be succinctly depicted by a single workflow. On the other hand, the BPG for screening and management of sepsis involves multiple workflows with complex inter-workflow interactions. Can Medi$\mathbb{K}$'s design can adequately accomodate diversity in BPGs, without compromising on comprehensibility?

RQ 2 (Applications): Can our approach be used to build real-world CDSSs for complex clinical scenarios? By real-world, we mean systems that are capable of consideration for use in hospitals.

RQ 3 (Ecosystem): Do semantics derived tools work well for CDSSs? If the tools derived from the semantics cannot be used to execute and analyze CDSS code, then the purpose of having said semantics defined in an executable manner is defeated. What additional tools/techniques do we need to develop or adapt for CDSS-specific functionalities?

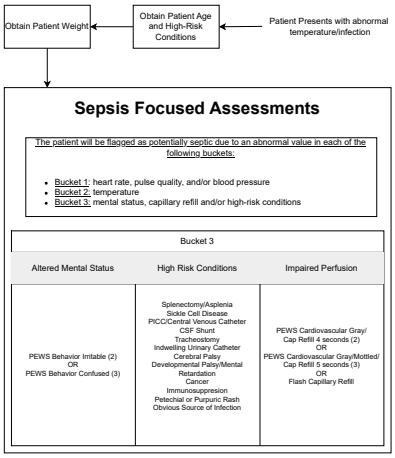
2 Background

We first introduce relevant background work before discussing research challenges. In section 2.1, we use a real-world Best Practice Guideline (BPG) for management of sepsis in pediatric cases is used to introduce Guidelines-based Clinical Decision Support Systems (CDSSs). In section 2.2, we desribe progress made towards addressing said challenges by existing approaches and their limitations. Central to our approach is the *semantics-first* philosophy, which is introduced in section 2.3.

2.1 Guidelines-based Clinical Decision Support Systems

We use a real-world BPG for management of sepsis in pediatric cases used at the Children's Hospital of Illinois at OSF St. Francis Medical Center (OSF) to introduce BPGs and Guidelines-based Clinical Decision Support Systems (CDSSs), and illustrate characteristics desirable of a framework for such system. Sepsis is life-threatening condition caused by the body's extreme response to an infection [25], and is a major cause of morbidity and mortality in children [26]. Adverse outcomes can, however, be mitigated through timely identification and prompt treatment with antibiotics and intravenous (IV) fluids [27, 28]. BPGs for screening and management of sepsis in pediatric Emergency Departments (EDs) have shown effectiveness in screening and management of sepsis [26], leading to their adoption in many pediatric EDs [29, 30].

In Fig. 1a, we present a simplified version of the screening section of OSF's sepsis management guideline. In essence, when a patient arrives at the emergency department ED with a fever or an infection, the healthcare provider (HCP) is supposed to obtain (a) the patient's age, (b) any conditions, such as cancer, immunosuppression, etc, that increase likelihood of sepsis, and (c) the patient's vital signs, such as heart rate, systolic blood pressure, respiratory rate, etc. This information is then used to check for abnormalities



(a) Pediatric sepsis screening BPG

Age	Heart Rate	Systolic BP	Temp
$0d-1m$	>205	<60	<36 or >38
$\geq 1m-3m$	>205	<70	<36 or >38
$\geq 3m-1y$	>190	<70	<36 or >38.5
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\geq 13y$	>100	<90	<36 or >38.5

(b) Vital Signs Chart

in clusters of linked information, called "buckets". For instance, if the patient's heart rate is abnormal, then "bucket 1" is said to have an abnormal value. Checking for such abnormalities often involves the use of tables, such as Table 1b, that contain normal ranges indexed by *age*. If the patient has at least one abnormal value in every "bucket", then he/she is flagged as potentially septic.

The BPG-recommended treatment for sepsis involves multiple concurrent workflows, such as screening for septic shock, administering fluids and inotropes, and administering antibiotics. While we refer the reader to [31] for more details, we briefly mention intended treatment to highlight the complexities in BPGs. Workflows may need to be performed concurrently and the status of one workflow may impact the other.

This real-world BPG exhibits characteristics common across many BPGs. Specifically BPGs typically:

- Involve *concurrent* workflows, such as administering drugs, monitoring vitals, performing treatment, etc. There may also be inter-workflow interactions. For instance, a diagnosis of sepsis during the screening may require modifications to an ongoing course antibiotics.
- Often specified in a *flowchart-like* notation. See [32] and [33] for other flowchart-based BPGs for management of *cardiac arrest*, and screening, risk-reduction, treatment and survivorship in cancer care respectively.
- Require communication between *heterogeneous agents* such as monitors and Electronic Health Records (EHRs).
- Often use *tables* indexed by parameters such as age, weight, etc to present normal/abnormal ranges for measurements, or recommended dosages for drugs.

2.2 Related Work

Existing research has been instrumental in both recognizing aforementioned challenges and addressing them to various degrees, leading to greater CDSS adoption. Recall from section 1 the following challenges in building CDSSs:

- **Specification-Implementation Gap:** The specification, i.e., the BPG, may differ from its translation in a computable medium, i.e. CIG.
- **Complexity:** BPGs encode complex medical knowledge, involving multiple *concurrent* workflows and *inter-workflow* interactions. Moreover, *CDSSs* interact with *diverse external agents* such as medical sensors and patient records.
- **Formal Analysis:** As CDSSs are *safety-critical*, it's desirable to have execution engines with *correctness guarantees* and a suite of formal analysis tools such as model checkers and deductive verifiers that can handle aforementioned complexity.

In Table 1, we provide an overview of how existing work addresses aforementioned challenges. We use ✓, ✓, and ✗ to depict that an approach fully-addresses, partly-addresses, or doesn't address a limitation respectively. We next go over notable approaches, and describe how they incrementally address said challenges. Next, we discuss each approach in detail.

	Specification-Implementation Gap	Complex Workflows	Diverse Agents	Formal Analysis
Arden Syntax	✓	✗	✗	✗
GLIF	✓	✓	✗	✗
Asbru	✓	✓	✗	✓
PROForma	✓	✓	✗	✗
GLARE	✓	✓	✗	✓
SAGE	✓	✓	✓	✗
Promela/SPIN	✗	✓	✗	✓
AMSs	✗	✓	✗	✓
P	✗	✓	✓	✓
Medi$\mathbb{K}$	✓	✓	✓	✓

Table 1: Comparison of Existing Approaches

Arden Syntax: The Arden Syntax [34] is a widely used medium for writing executable BPGs. First proposed in 1989, it was among the first languages to recognize and address the *specification-implementation gap* by emphasizing *comprehensibility* to non-experts in Computer Science [35]. Guidelines are described using Medical Logic Modules that contains information related to guideline's purpose , maintenance, and medical knowledge. The modules are modular to allow re-use and sharing across hospitals. But, Arden Syntax is focused on describing simple, modular, and independent guidelines (such as reminders), and not on guidelines with complex logic (such as treatment protocols) [36].

GLIF: Arden Syntax's limitation in modeling complexity is addressed by GLIF [37]: a language that uses flowcharts to expressed guidelines. A multi-level approach is employed to manage complexity: at the top is the conceptual level, where only high-level details relevant for human-comprehension are present. In the middle is a computable-level, where details of guideline execution flow and patient data elements are specified. At the bottom is the implementable level, where institution-specific details and mappings into patient data are specified. Both Arden Syntax and GLIF eliminate the gap between the BPG and its implementation as they're meant to be either directly used by clinicians (or in collaboration with computer scientists) to express BPGs in an executable medium. Such executable BPGs expressed in them are meant to be shared across hospitals, and are thus modular. However, neither formalism has complete formal semantics, or comprehensive support for rigorous formal analysis.

Asbru: The need for formal analysis is identified by Asbru: a formalism with formally defined syntax and semantics [23]. In Asbru, a guideline is modeled as a plan that contains: (i) intentions that define aims, (ii) conditions that specify when the plan is applicable, (iii) effects that define expected behavior during execution, and, (iv) a body containing other sub-plans. Apart from an execution engine, the Asbru ecosystem also contains other tools, such as a model checker for verification [38]. However, the formal semantics of Asbru have been only partially defined, and is insufficient to implement tools for the language [22].

PROForma: The importance of a complete formal-semantics is identified and addressed by PROforma [22], another formalism that uses plans to model guidelines. A PROforma plan is made of a sequence of tasks. The plan defines constraints on their enactment, and circumstances for termination (for example, exceptions) [22]. But, despite having complete formal semantics, it does not have a comprehensive suite of formal analysis tools such as model checkers, deductive verifiers.

SAGE: The SAGE guideline model [39] uses the Protégé knowledge representation framework [40] to model guidelines, and improves on aforementioned approaches by enabling seamless integration into hospitals' existing Clinical Information Systems (CISs). But, it lacks complete formal semantics, and analysis tools such as deductive verifiers and model checkers. The GLARE formalism [41] uses an actions based approach to represent guidelines, and addresses clinician-comprehensibility and modularity. For formal analysis, GLARE guidelines can be translated to Promela: the SPIN model checker's specification language [42]. The approach partly addresses holistic safety as external agents (such as clinicians) can be modelled and analyzed. But, the scenario where the external agent's behavior deviates from the model during system execution isn't addressed.

Languages outside the medical domain can also be used to reason about medical systems. For example, in [43], Abstract State Machines (ASMs) are used to validate and verify a system for measuring patients' stereoacuity in the diagnosis of amyblyopia. Along similar lines, the P language [44] and analysis of large concurrent systems provides a convenient medium for expressing multiple workflows and interactions as concurrently executing Finite State Machines (FSMs). External agents can be modeled using *ghost* that are used for formal analysis, but discarded at runtime. But such formalisms, while suitable for formal verification, may not be easily comprehensible to clinicians for validation.

This proposal aims to build on progress made by existing work to address aforementioned challenges using the *semantics-first* approach, which we describe in the following section.

2.3 Semantics-First Approach and the $\mathbb{K}$ Framework

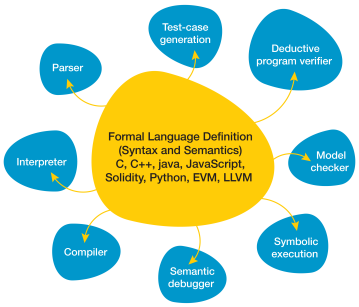


Fig. 2: *Semantics-first* Approach

Recall from section 1 that at the core of our approach is a new DSL for writing executable BPGs called Medi$\mathbb{K}$. The *semantics-first* approach dictates that instead of implementing tools such as interpreters, model checkers and deductive verifiers from scratch in an ad-hoc way, the semantics of the language should be formally defined and utilized to derive said tools in a *correct-by-construction* fashion, as shown in Fig. 2.

For a language like Medi$\mathbb{K}$, the semantics first approach has many benefits. First, since Medi$\mathbb{K}$ is meant to be used in safety-critical settings, it's vital that its interpreter has correctness guarantees and formal analysis tools such as model checkers and deductive verifiers, which the approach ensures. Second, Medi$\mathbb{K}$ has to evolve quickly to incorporate feedback that it receives from domain experts. Had we implemented all tools from scratch, any update to the language would require updates to the tools as well. With our approach, only the formal semantics need updating. Since the tools are derived from the semantics, they reflect all changes automatically.

Medi$\mathbb{K}$'s semantics have been formalized in the $\mathbb{K}$-framework, which we describe next. $\mathbb{K}$ is a rewrite-based executable semantics framework in which programming languages can be defined using configurations and rules [45]. Semantics defined in $\mathbb{K}$ typically consist of the following three components:

- *Syntax:* Defined using BNF-like notation, and utilized by $\mathbb{K}$ to generate a parser for the language.

- *Configuration:* Organizes the program execution state into units called *cells* that be nested.
- *Rules:* Operate over configuration segments and define program evolution via rewrites.

```
1  module IMP-SYNTAX
2    imports DOMAINS-SYNTAX
3    syntax Exp ::=
4        Int
5      | Id
6      | Exp "+" Exp     [left, strict]
7      | Exp "-" Exp     [left, strict]
8      | "(" Exp ")"     [bracket]
9
10   syntax Stmt ::=
11       Id "=" Exp ";" [strict(2)]
12     | "if" "(" Exp ")"
13         Stmt Stmt     [strict(1)]
14     | "while" "(" Exp ")" Stmt
15     | "{" Stmt "}"    [bracket]
16     | "{" "}"
17     > Stmt Stmt       [left, strict(1)]
18   syntax Pgm ::= "int" Ids ";" Stmt
19   syntax Ids ::= List{Id,","}
20 endmodule
```

```
21 module IMP imports IMP-SYNTAX
22   imports DOMAINS
23   syntax KResult ::= Int
24   configuration
25    <T> <k> $PGM:Pgm </k>
26        <state> .Map </state> </T>
27   rule <k> X:Id => I ...</k>
28        <state>... X |-> I ...</state>
29   rule I1 + I2 => I1 +Int I2
30   rule I1 - I2 => I1 -Int I2
31   rule <k> X = I:Int ; => . ...</k>
32        <state>... X |-> (_ => I) ...</state>
33   rule {} S:Stmt => S
34   rule if(I) S _ => S requires I =/=Int 0
35   rule if(0) _ S => S
36   rule while(B) S => if(B) {S while(B) S} {}
37   rule <k> int (X, Xs => Xs) ; S </k>
38        <state>... (. => X |-> 0) </state>
39   rule int .Ids ; S => S
40 endmodule
```

Fig. 3: IMP Semantics in $\mathbb{K}$

In Fig. 3, we provide the $\mathbb{K}$ semantics for Imp, a simple imperative language that supports arithmetic expressions and statements such as variable declaration and assignment, branching (`if`) and looping (`while`). In $\mathbb{K}$ code must be placed inside a `module`. The Imp definition consists of two such modules: `IMP-SYNTAX` (lines 1-20) defining the language's syntax, and `IMP-SEMANTICS` containing the `configuration` (lines 24-26) and rewrite rules (lines 27-39). Note that `module IMP` imports `module IMP-SYNTAX`, allowing rules to utilize the language's syntax.

Syntax in $\mathbb{K}$ is defined using BNF-like notation; terminals are enclosed in quotes, and non-terminals begin with an uppercase. For example, consider the declaration of Imp expressions on lines 3-8. On lines 4-5, $\mathbb{K}$'s builtin support for integers (`Int`) and program identifiers (`Id`) is used to support arithmetic expressions over program variables. On lines 6 and 7, the attribute `left` is used to declare the corresponding operator as left associative. The `strict` attribute is used to assign *evaluation strategies*. On lines 6 and 7, the attribute signifies that both operand sub-expressions must be completely evaluted before the before the operator (`+`, `-`) can be evaluted. Similarly, for an assignment statement on line 12, the `strict(2)` indicates that the second argument, i.e., the expression to the right of the `=` sign must be evaluated before the identifier on the left of the `=` is updated. On line 17, the `>` signifies that block of `Stmt` productions preceding the sign (lines 11-16) have a higher precedence, i.e. bind tighter, than the production on line 17. On line 18, an Imp program is defined to start with a list of program variable declarations (`"int" Ids ";"`) followed by other statements, where `List{Id, ","}` is used on line 19 to define `Ids`, the comma-separated list of program identifiers.

Once the *syntax* has been defined, $\mathbb{K}$ can utilize it to generate a parser for programs in the language. To impart semantics, we must describe how the program's *execution state* evolves during execution. In $\mathbb{K}$ *state* is organized into a *configuration*. The configuration is an unordered list of (potentially nested) *cells*, specified using an XML-like notation. When writing *rules* (as rewrites) over this state, any subset of the *cells* present in the configuration can be mentioned. Thus, only parts of the state *relevant* to the rule need to be mentioned; the rest of the state is assumed to remain unchanged. The keyword `configuration` (line 24) defines a *configuration*, followed by xml-like notation for the $\mathbb{K}$ *cells*. For instance, `<foo> <bar> ... </bar> </foo>` corresponds to $\mathbb{K}$ *cells* with the names `foo` and `bar` respectively, where `bar` is nested under `foo`.

The configuration for Imp (lines 24-26) consists of a top cell `<T>` that contains a `<k>` cell (line 25) and a `<state>` cell (line 26). Typically, the program to be executed is placed in the `<k>` cell. At runtime, `PGM` is

replaced by the Abstract Syntax Tree (AST) of the program. On line 25, `PGM:Pgm` signifies that the program must parse as the sort `Pgm` defined on line 18. The dot (.) in $\mathbb{K}$ indicates *empty*. For instance, `.Map` on line 26 defines the contents of the `<state>` cell to be an *empty* map. Said map will be utilized to keep track of values assigned to program identifiers during execution.

$\mathbb{K}$ *rules* operate over configuration segments and define evolution of program state. A rule is begins with the keyword `rule` and is a statement of the form $\varphi \Rightarrow \psi$, where φ, ψ are patterns over configuration terms and $\mathbb{K}$ variables. We say φ is the LHS and ψ is the RHS of the rule. We define a *substitution* θ be a map from $\mathbb{K}$-variables to terms. Say, for given pattern φ and *substitution* θ, $\varphi\theta$ is the pattern obtained by replacing every variable v in φ with $\theta(v)$. We pattern φ matches term τ iff there exists a substitution θ s.t. $\tau = \varphi\theta$. During execution, if the current configuration term τ *matches* the LHS φ of rule $\varphi \Rightarrow \psi$ with substitution θ, then C is rewritten to $\psi\theta$. Consider the rule for program variable assignment on lines 31-32. $\mathbb{K}$-variables always begin with an uppercase, and may be suffixed with `:S`, where `S` is the variable's sort. The rule in considerations uses two variables `X` and `I` having sorts `Id` and `Int` respectively. Suppose we're trying to evaluate the statement `x = 2 + 3;`. Recall that the `<k>` cell typically contains the program. Thus, we have would have `x = 2 + 3;` at the start of the cell. Note that in $\mathbb{K}$, ... is used to represent parts of the configuration *not relevant* to the rule. Said parts remain *unchanged* during the rewrite. Since variable *I* is suffixed with the sort `Int`, the variable assignment rule will *not* match the configuration. Since the assignment statement is `strict` in the second argument (line 11), $\mathbb{K}$ will *heat*, or pull-out the second argument for evaluation. This results in a configuration term of the form `<k> 2 + 3` $\curvearrowright$ `x = □; ...</k>`, where $\curvearrowright$ to mean *followed-by*, i.e. the evaluation of `2 + 3` must occur *before* the evaluation of `x = □;`. □ denotes a *hole* left in place of the argument that was *heated*. `<k> 2 + 3` $\curvearrowright$ `x = □; ...</k>` is re-written to `<k> 5` $\curvearrowright$ `x = □; ...</k>` by an application of the arithmetic addition rule on line 29. On line 23, we specify any term of sort `Int` to be a `KResult`. This signifies that the term can no longer be evaluated, and is *cooled* or plugged-back into the original, resulting in `<k> x = 5; ...</k>` Now the LHS of the assignment rule can *match* with substitution $\theta = (\texttt{X} \mapsto \texttt{x}, \texttt{I} \mapsto \texttt{5})$, resulting in the value that identifier `x` is bound to in the `<state>` cell to be rewritten to `5`. Note the difference between program identifiers and $\mathbb{K}$ variables. While program variables are simply terms belonging to sort `Id`, $\mathbb{K}$ variables have logical meaning. If multiple rules can match the configuration term, then one rule is non-deterministically chosen. Execution is a sequence of rule applications that continues until no rule can match the configuration.

2.4 Matching Logic

The semantics of a $\mathbb{K}$ definition are given in Matching Logic (ML), which we briefly describe next. We only present concepts that are relevant to this proposal, and direct the reader to [46, 47] for a more thorough introduction to ML.

An *ML* Σ*-theory* is the pair (Σ, Γ), comprising of Σ, a *signature*, and Γ a set of Σ*-patterns* called *axioms*.

The *syntax* of matching logic is parametric in two sets of variables *EV* and *SV*, called *element variables* and *set variables*. We use a lowercase letter x (uppercase letter X) to denote $x \in EV$ $(X \in SV)$. A *(matching logic) signature* Σ is a set of *(constant) symbols*. A *(matching logic) pattern* φ is inductively defined as:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \rightarrow \varphi_2 \mid \exists x.\, \varphi \mid \mu X.\, \varphi$$

where in $\mu X.\, \varphi$ it's required that φ has no negative occurrences of X.

Next, we give an intuive description of the semantics of ML. Given signature Σ, a Σ*-model* contains, among other things, a carrier set M of elements, and for every $\sigma \in \Sigma$, an *interpretation* $\sigma_M \subseteq M$ for every $\sigma \in \Sigma$. In *models*, patterns evaluate to sets of elements that *match* them. For instance, variable $x \in EV$ evaluates to

a *singleton* containing *element*, while $X \in SV$ evalutes to some subset of M. $\bot$ evaluates to the $\emptyset$. $\varphi_1 \rightarrow \varphi_2$ is matched by all elements that match φ_2 if they also match φ_1. $\exists x.\, \varphi$ allows abstracting away irrelevant parts (i.e. x) of the pattern φ. $\varphi_1\ \varphi_2$ allows for application of a function/operation/constructor, i.e. φ_1, to its arguments, i.e. φ_2. $\mu X.\, \varphi$ evaluates to the *smallest* set X s.t. φ evaluates to X. Other patterns such as $\neg\varphi$, $\varphi_1 \vee \varphi_2$ can all be derived from the basic ones mentioned above, and have expected semantics. For instance, $\neg\varphi$ matches everything *not* matched by φ. $\varphi_1 \vee \varphi_2$ matches everything that either φ_1 or φ_2 matches. We refer the reader to [48] for a detailed discussion on the semantics of matching logic.

2.5 $\mathbb{K}$ in Matching Logic

Given a language L defined in $\mathbb{K}$, the $\mathbb{K}$ tool compiles the $\mathbb{K}$-definition of L into a matching logic theory Γ^L. Recall from section 2.3 that a $\mathbb{K}$-defintion consists of rewrite rules of the form $\varphi \Rightarrow \psi$. These rules define a *transition system* over computation configurations. This is captured in ML using a symbol $\bullet \in \Sigma$, called *one-path-next*. Intuitively, for a given configuration γ, $\bullet\gamma$ is *matched* by all states that can transition to γ in one step [48]. We can then define $\diamond\varphi \equiv \mu X.\, \varphi \vee \bullet X$, which equals $\varphi \vee \bullet\varphi \vee \bullet\bullet\varphi \ldots$ or $\bigvee_{i\in\mathbb{N}} \left(\bullet^i \varphi\right)$. Intuitively, the $\diamond$ operator behaves exactly as its modal-logic equivalent. It matches all states that can *reach* φ in zero or more rewrite steps. Thus, a rewrite $\varphi \Rightarrow \psi$ is expressed in ML as $\varphi \rightarrow \diamond\psi$, or the set of all configurations that if match φ, can also reach ψ in zero or more steps.

3 Current Results

In this section, we describe progress made towards goals listed in section 1. In section 3.1, we describe our experience with the semantics-first approach in $\mathbb{K}$ that provided confidence in using $\mathbb{K}$ as the basis of our solution. We then briefly describe our experience of using a $\mathbb{K}$ definition directly to encode medical guidelines in section 3.2, followed by a discussion on the Medi$\mathbb{K}$ DSL in section 3.3

3.1 Evaluating $\mathbb{K}$ for our approach

To derive maximum benefit from the $\mathbb{K}$ semantics, it is important that: (i) the $\mathbb{K}$ semantics is descriptive enough to implement language-specific tools such as interpreters, model checkers, etc, and, (ii) $\mathbb{K}$-derived tools compare favorably against their language-specific counterparts. In Medi$\mathbb{K}$'s case, we needed to ensure that the interpreter, in particular, is appropriate for use in a CDSS. The concern for performance stems from the fact that being language-agnostic, $\mathbb{K}$ derived tools cannot accommodate language-specific optimizations.

To evaluate the feasibility of $\mathbb{K}$ semantics and derived tools, we drew upon our work of defining the semantics of a real-world language (unrelated to Medi$\mathbb{K}$), where the $\mathbb{K}$ semantics and derived tools offer are viable alternatives to their language-specific counterparts.

In [49], we completely defined the semantics of the Ethereum Virtual Machine (EVM) in $\mathbb{K}$. The EVM is a low-level stack-based language that forms backbone of the Ethereum Blockchain. Unlike Medi$\mathbb{K}$, EVM had an existing informal paper-based semantics called the Yellow Paper [50], and a reference implementation in C++ [51]. This allowed us to compare the $\mathbb{K}$ based semantics and interpreter against mature language-specific ones.

We based our $\mathbb{K}$ semantics on the Yellow Paper, and found several inconsistencies that were confirmed by its developers [49]. This stems from the fact that unlike the $\mathbb{K}$ semantics, the Yellow Paper is an informal

document in natural language. Often, the Yellow Paper was found to be unclear, underspecified or inconsistent with the behavior of actual implementations. As the $\mathbb{K}$ semantics are executable, all details have to considered for our interpreter to pass all tests in the reference test-suite. We utilized the developer documentation for our semantics to generate an alternative to the Yellow Paper itself, called the Jello Paper [52]. The Jello Paper has been well-received by the community, with discussions to designate it as the *canonical document* instead of the Yellow Paper. For more details, we refer the reader to section VII of [49].

Test Set (no. tests)	**Lem EVM**	**KEVM**	**cpp-ethereum**
Lem (40665)	288:39	34:23	3:06
VMStress (18)	-	72:31	2:25
VMNormal (40665)	-	27:10	2:17
VMAll (40683)	-	99:41	4:42
GSNormal(22705)	-	35:00	1:30
GSQuad (250)	-	855:24	0:21
GSAll (22955)	-	889:00	1:51

Fig. 4: Lem EVM vs KEVM vs cpp-ethereum

Fig. 4 compares the $\mathbb{K}$-generated interpreter against the reference implementation (**cpp-ethereum**) and another formalization (**Lem EVM**) of the EVM semantics [53] in Lem [54] on the official test that implementations are expected to pass. All execution times are given as the full sequential CPU time (in MM:SS format) on an Intel i5-3330 processor (3GHz on 4 hardware threads) and 24 GB of RAM. By comparing to the C++ reference implementation, we show the feasibility of using the KEVM formal semantics for prototyping, development, and test-suite evaluation.

The row Lem indicates a run of all the tests that the Lem semantics can run (a subset of the VMTests). The row VMStress indicates a run of all 18 stress tests in the test-suite, to compare the performance of KEVM with the C++ implementation. The row VMNormal is a run of all the non-stress tests in the test-suite (*not* the same set of tests as Lem). VMAll is the addition of the second and third rows and is included for completeness. The last three rows indicate a runs of the GeneralStateTests; GSNormal are the non-stress tests, GSQuad are the stress tests, and GSAll is the addition of the two. Under the GeneralStateTests, our tools performs well except in the case of QuadraticStateTests (250 out of 22955).

As shown in the comparison, not only did the $\mathbb{K}$ generated interpreter significantly outperformed the Lem-based formal executable EVM semantics, it performs favorably when compared to the language-specific reference implementation. It was under 30 times slower on the stress tests, roughly 20 times slower on all tests, and only 11 times slower on VMNormal tests. Note that since the publication of the paper, newer versions of $\mathbb{K}$ have been introduced with significant performance improvements.

In [49, 55], we used the $\mathbb{K}$ semantics of the EVM to build a tool for verification EVM smart contracts. As EVM bytecode is a low-level stack based language lacking constructs such as functions, verification of smart contracts proved tedious. To address this, we introduced a DSL for simulating function calls modeled on the Ethereum ABI [56] and several EVM-specific abstractions to make verification scalable. As a result, we could verify several real world smart contracts satisfied their correctness specifications. Thus, our work allowed to us to conclude that (i) the performance of a $\mathbb{K}$ generated interpreter compares favorably to its language-specific counterpart, (ii) the $\mathbb{K}$ can be used to implement a language from scratch, i.e. without an informal paper based semantics, as in the case of our DSL for calling functions at the bytecode level, and, (iii) the verifier can be used on large programs, but may require additional work, such as simplification abstractions and lemmas.

3.2 Executable Best-Practice Guidelines as $\mathbb{K}$ Definitions

In [57], we presented a CDSS for management of cardiac arrest based on a well known BPG published by the American Heart Association [58]. Recall from section 2.1 that BPGs are often expressed using flowchart-like notation. The cardiac arrest management BPG also follows the same convention. In our work, we encoded transitions in BPG flowcharts as $\mathbb{K}$-rules. As $\mathbb{K}$'s configuration abstraction mechanism enables only relevant parts of the configuration to be mentioned, we argue that our methodology permits *succinct* representation of medical knowledge. This is supported by the fact that our $\mathbb{K}$ definition encoding the BPG was only a few hundred lines of code, and could be used to implement an entire CDSS. But, the $\mathbb{K}$ based BPG proved hard to comprehend, both to computer scientists not familiar with $\mathbb{K}$, and healthcare professionals. Although this work *failed* to address the objectives from section 1, it emphasized the importance of comprehensibility to HCPs for future endeavors.

3.3 Medi$\mathbb{K}$

In [31] we presented the Medi$\mathbb{K}$ DSL for expressing BPGs that are both executable and comprehensible to HCPs. This enables HCPs to validate the correctness of medical knowledge encoded in Medi$\mathbb{K}$ code. Recall from section 2.1 the following characteristics of BPGs: (a) involve *concurrent* workflows with inter-workflow dependencies, (b) are often specified using a *flowchart-like* notation, and, (c) require communication between heterogeneous *external* agents such as sensors, monitors and Electronic Health Records.

```
1  [init] machine <IDENTIFIER>
2    receives <IDENTIFIER_LIST> {
3    vars <IDENTIFIER_LIST>;
4
5    [init] state <IDENTIFIER> {
6      entry [(IDENTIFIER_LIST)] {
7        <STMT> // entry block
8      }
9      on <IDENTIFIER> [(IDENTIFIER_LIST)] do {
10       <STMT> // event handler
11     }
12   }
13 }
```

Fig. 5: Medi$\mathbb{K}$ FSM Skeleton

Medi$\mathbb{K}$ has been designed to account for aforementioned characteristics. In Medi$\mathbb{K}$, like in P, programs are expressed as concurrently executing instances of state machines that communicate via passing messages. Given a BPG where each workflow is expressed as a flowchart, we express said flowcharts as State Machines in Medi$\mathbb{K}$. Each flowchart node in the BPG is represented as a state in a state machine, and edges are represented as state transitions. During execution, instances of these machines are created, which interact with each other by passing events. Note the distinction between machine and its instance. A machine is analogous to an Object Oriented Programming (OOP) class, whereas its instance is analogous to an OOP object.

As FSMs are central to our approach, we provide a skeleton of a Medi$\mathbb{K}$ machine in Fig. 5 before explaining how BPGs are encoded in Medi$\mathbb{K}$ code. Note that we use `[...]` to denote optional constructs, `<...>` for mandatory constructs, lowercase for terminals, and uppercase for non-terminals.

A Medi$\mathbb{K}$ program consists of a set of machine definitions. A machine definition starts with the keyword `machine`, followed by its name (line 1). On line 2, following the `receives` keyword, is a comma-separated list of identifiers signifying the events that the machine can receive from other machines. One machine in a program can be prefixed with the `init` keyword. This machine is referred to as the initial machine. On line 3, following the keyword `vars`, another comma-separated list of identifiers signifies the instance-variables. During execution, each instance maintains a mapping from these variables to values. Each machine defines a set of states, such as the one in lines 5-11. A state has a name, an optional entry block (lines 6-8), and a set of event handlers (lines 9-11). The entry block begins with the keyword `entry`, and may contain a list of variables that are bound to values when the state is entered during execution. One state in the machine may

```
1  interface HeartRateSensor { }
2
3  machine TreatmentMachine { ...
4    var hrSensor = createFromInterface(HeartRateSensor,
5                                       "heartRateSensor");
6    var heartRate = obtainFrom(hrSensor, "heartRate");
7  }
```

(a) External Sources

```
1  fun isHeartRateNormal() {
2    days(age) in {
3      interval(days(0)   , months(1)): return hr > 205;
4      interval(months(1), months(3)): return hr > 205;
5      // omitting other cases
6      default                        : return hr > 100;
7    }
8  }
```

(b) Checking abnormality using tables

Fig. 6: Executable BPGs in MediK

```
1  machine SepsisScreening receives .. {
2    init state Start {
3      on StartScreening do {
4        goto ObtainAge;
5      }
6    }
7    state ObtainAge {
8      entry {
9        send tablet, Instruct, ("get age");
10     } on ConfirmAgeEntered do {
11       goto ObtainWeight;
12     }
13   }
14   state ObtainWeight { ... }
15   state ObtainHighRiskConditions { ... }
16   state CalculateScore {
17       var hrAbnormal = !isInNormalRange("HR", ...);
18       var bucket1    = hrAbnormal ||  ...
19       var bucket3    = mentalStatusAbnormal || ...
20
21       var sepsisSuspected
22         = bucket1 && bucket2 && bucket3;
23
24       send tablet, SepsisDiagnosis
25         , (sepsisSuspected);
26   }
27 }
```

Fig. 7: Sepsis Screening in MediK

be prefixed with `init`, specifying the initial state. When execution begins, an implicit instance of the initial machine is created, and the `entry` block of its initial state is executed. When an instance of a machine is dynamically created during runtime, the `entry` block of its initial state is executed. Event handlers within a state begin with `on` followed by the event name and an optional list of variables. When the event handler is executed, data from the received event's payload is bound to aforementioned variables which can be used in the code block that follows the `do` keyword.

To handle interaction with heterogeneous external sources, MediK models them as interfaces. An interface is a FSM that has its transition system defined externally. For example, certain measurements such as the heart rate are often obtained from sensors. The code in Fig. 6a shows the process of obtaining external measurements in MediK.

Since we don't have the transition system for the heart rate sensor, we declare it as an interface (line 1). Next, instead of using `new` to create an instance, we use a builtin MediK construct `createFromInterface`, which takes as arguments (a) the inteface name (lines 4), (b) a unique identifier string used to identify the instance outside the MediK process. All other MediK machines can interact with external sensor using variable `hrSensor`. There is no need to make any distinction between external, and MediK-based machines. To deal with external interactions, input and output pipes are provided to the MediK process at launch. When the `send` construct is used on an external machine, MediK will write a JSON [59] message with the event data, the identifier from line 5, and a unique transaction id to the *write-end* of the output pipe. At the *read-end*, we need to write external code (in any programming language) to handle the JSON message. In the example above, this involves reading from the external heart rate sensor. To send data to MediK, a JSON message in a pre-specified format needs to be written to the *write-end* of the input pipe.

Recall from 2.1 the guideline for sepsis screening shown in Fig. 1a. In Fig. 7, we show MediK code corresponding to the sepsis screening guideline. When modeled in MediK, a flowchart in the guideline is represented using a MediK machine. *Nodes* in the flowchart are represented as *states* in a MediK machine, while flowchart *edges* as *state-transitions*. Note that we use *node* to refer to constructs in the flowchart, and *state* to refer to counterparts in MediK. Also, while it's desirable to represent each flowchart *node* as a state machine *state*, the task in the flowchart *node* may warrant using multiple state-machine *states*. For

example, in Fig. 1a, the step "Obtain Patient Age, Weight, and High Risk Conditions" is translated to states `ObtainAge` (lines 7-13), `ObtainWeight` (line 14), and `ObtainHighRiskConditions` (line 15) in Fig. 7. Within each of these states, the code permits communication with heterogeneous external agents for obtaining required parameters. For instance, on line 9, an `Instruct` event is sent to an external `tablet` machine with the payload `"get age"`. The recipient process runs on a tablet held by the Healthcare Provider, and handles the event by prompting the provider to enter the patient's age. A `ConfirmAgeEntered` event, emitted once the age is obtained, enables the screening machine to proceed to the next step (lines 11-13). Once all appropriate measurements have been obtained, they are checked for abnormality (lines 18-26) using tables shown in Fig. 6b to arrive upon a diagnosis.

We draw attention to the fact that the program in Medi$\mathbb{K}$ structurally resembles the paper-based BPG from Fig. 1a. As demonstrated in [31], our Medi$\mathbb{K}$-based approach allowed us to conveniently model all screening and treatment workflows from section 2.1 to build a functioning CDSS that is being considered for clinical simulations.

4 Research Plan

In this section, we discuss remaining challenges we plan to address through this work. In section 5 we summarize both completed and pending work, and discuss how we address research questions laid out in section 1

4.1 Semantics Based Visual Representation Generation

Extremely vital to the success of Medi$\mathbb{K}$, the ability to generate visual representation of Medi$\mathbb{K}$ programs to aid comprehensibility to medical domain experts, enabling them to verify accuracy of encoded medical knowledge. Even though Medi$\mathbb{K}$ code structurally resembles paper-based guidelines, said representations aid comprehensibility for medical domain experts as the representations resemble their paper-based counterparts, imparting a sense of familiarity.

Control Flow Graphs (CFGs) form the basis of not only such visual representations, but other important analyses. For example, flow-sensitive analyzers can be used to verify that a medical procedure is performed on both treatment branches. CFGs find use in applications from compilers to model checkers. But, despite their importance, they are not well understood. To illustrate the lack of a *canonical* definition of a CFG, in [60], the authors used three different CFG generators on the same program, and obtaining three different CFGs, each tailored to the objectives of the generator.

Thus, generation of CFGs that present *all relevant* information, with *minimal* intervention during generation, in a *correct-by-construction* fashion, is indeed challenging. In [60] that authors argue that operational semantics are not suitable to be used directly for generation of CFGs by demonstrating that many control flow edges correspond to different halves of the same rule, while some correspond to none. To address this, the authors utilize an algorithm to construct an Abstract Machines (AM) style semantics from the SOS.

Consider for example, the following SOS rules for assignment:

$$\frac{e, \mu \rightsquigarrow e' \mu'}{x := e, \mu \rightsquigarrow x := e', \mu'}\ \textit{AssnCong} \qquad \frac{}{x := v, \mu \rightsquigarrow (\text{skip}, \mu\,[x \to v])}\ \textit{AssnEval}$$

The authors then automatically generate an AM, written $\langle (t, \mu) \mid K \rangle$ where K is the *context* or *continuation* composed of frames of the form $[x := (\square_t, \square_\mu)]$ indicating holes for an evaluated value and environment.

Thus, the AM rules are:

$$\langle (x := e, \mu) \mid k \rangle \rightarrow \langle (e, \mu) \mid k \circ [x := (\square_t, \square_\mu)] \rangle$$
$$\langle (v, \mu) \mid k \rangle \rightarrow \langle (\text{skip}, \mu\,[x \rightarrow v]) \mid k \rangle$$

Next, as it's possible for the AM to have an infinitely-many states, an abstraction is required to collapse the states and make the transition system finite. To this end, the authors utilize a *value irrelevant* abstraction that replaces all values with a single one $\star$ that representing any of them. With this, the abstract states can be treated as nodes in the CFG. Thus, program execution with this abstraction will produce a CFG. For example, consider evaluation of statement $x := y$, under an infinite number of environments, can lead to an infinite number of states of the AM. But, converting all values to $\star$ results in one-possible execution: $\langle (x := y, [y \mapsto \star]) \mid k \rangle \rightarrow \langle (y, [y \mapsto \star]) \mid k \circ [x := (\square_t, \square_u)] \rangle \rightarrow \langle (\star, [y \mapsto \star]) \mid k \circ [x := (\square_t, \square_u)] \rangle \rightarrow \langle (\text{skip}, [x \mapsto \star, y \mapsto \star]) \mid k \rangle$

Note that expressions built over $\star$-values can still pose a challenge. For instance, a statement of the form `while e do s` when evaluated under an environment $[\star \mapsto \star]$ requires handling $\langle (e, [\star \mapsto \star]) \mid k \rangle$, which can be matched by any expression rule. To handle this, $\langle (e, [\star \mapsto \star]) \mid k \rangle$ is over-approximated to $\langle (\star, [\star \mapsto \star]) \mid k \rangle$ as evaluating any such e would eventually result in $\star$.

In this work, we plan to utilize ideas from [60] in conjunction with the $\mathbb{K}$-Summarizer toolchain. The $\mathbb{K}$-Summarizer is an effort to improve performance of semantics-based tools by modifying the $\mathbb{K}$-definition itself. One of the primary objectives of the summarizer is to enable semantics-based compilation. Semantics-based compilation can be considered a generalization of established techniques for established techniques for program optimization such as partial evaluation of programs [61].

For a given language L, let $[\![\_]\!]$ be the interpreter for L. Say, for given $p \in L$–Program, and inputs in_1, in_2, output $= [\![p]\!]\,[\text{in}_1, \text{in}_2]$ if p terminates. We can now express partial evaluation of program p, using another program mix, called so since performs a mix of both execution and code generation. Now, we get a two stage execution process:

$$p_{\text{in}_1} = [\![\text{mix}]\!]\,[p, \text{in}_1]$$
$$\text{output} = [\![p_{\text{in}_1}]\!]\text{in}_2$$

Now, for given program p, if in_1 is statically known (at stage 1), and in_2 is dynamically known (at stage 2), then, $[\![p_{\text{in}_1}]\!]$ can be computed *once*, and re-used for any new in_2. Semantics-based compilation can be seen as a generalization of this idea. Say L is a $\mathbb{K}$ definition. We can generate a new definition $L^{p_{\text{in}_1}}$ that directly implements $[\![p_{\text{in}_1}]\!]$. At runtime, only the dynamically determined input in_2 to complete execution.

Conceptually, the summarization process results in a CFG, where each basic block is summarized as a $\mathbb{K}$ rule. But, in Medi$\mathbb{K}$'s case, the inherent concurrency presents additional challenges that we intend to tackle.

While SBC is one of the primary goals of the $\mathbb{K}$-Summarizer project, it has also been used to optimize $\mathbb{K}$ definitions by *merging* multiple rules that take smaller steps into one rule that *summarizes* all of them. This is similar to the translation from SOS to abstract machine presented in [60]. In our work, we plan to generate such *summaries* soundly, and utilize them to generate visualizations of Medi$\mathbb{K}$ programs. Note, that both $\mathbb{K}$ Summarizer and the work in [60] have been tried in *deterministic* settings. We plan to address additional challenges from *non-determinism* in our work.

4.2 Formal Analysis of Best Practice Guidelines

In [31], we focused on designing a language that would be conducive to expressing BPGs. Our efforts were primarily focused on implementing a complex CDSSs that could be used to evaluate our approach in real-

world hospital setting. As a result, we (a) *caught issues* with the medical logic during the design phases, and, (b) *implemented* a system with positive HCP feedback during a *dry-run*. However, we identified limitations with the formal analysis component of our approach. Next, we idenitfy challenges and propose solutions.

Responsiveness Verification: In [31], we implemented a real-world CDSS for management of sepsis in pediatric cases, and verified a desired safety property holds. Recall that a MediK program consists of a concurrently executing instances of FSMs. During execution, an instance may be considered *stuck* if the event at the head of its input buffer does not have an associated handler, rendering it *unresponsive*. For this reason, languages such as P [44] raise an exception in such cases. One way to ensure *responsiveness* is to define event handlers for all events in all states. But, for MediK we found that: (a) it is tedious and error prone to define handlers for all events in all states, and, (b) comprehensibility of programs is reduced as many handlers may never fire during execution.

Thus, we employed a weaker notion of responsiveness. We verified that all possible events that a state can potentially receive have associated handlers. The property we desire our system does not eventually end up in a state φ_{stuck}, or, a state where the head of the input buffer has an event without an associated handler. The ML-pattern of interest is $\neg(\varphi_{init} \Rightarrow \varphi_{stuck})$, where φ_{init} is the initial pattern. We can then use K's search command to search for φ_{stuck} starting from φ_{init}. If K returns $\top$, then the desired property doesn't hold.

```
either {
  broadcast StartFluidTherapy;
  broadcast StartAntibioticTherapy;
} or {
  broadcast StartAntibioticTherapy;
  broadcast StartFluidTherapy;
}
```

Fig. 8: Non-determinism in MediK Ghosts

K has multiple backends, including a fast execution LLVM backend that can only handle *ground* patterns, i.e. patterns with logical variables, and a haskell backend that can handle ML patterns in their full generality. Since MediK programs involve interactions with the external world, performing a search requires modeling the external components. We address this by specifying external components as *ghost* machines - a technique also used by other state machine formalisms such as P [44]. For program analysis, *ghost* machines substitute external agents, permitting exploration of the state space. During execution, *ghosts* are discarded and replaced by actual external agents. Due to this, ghosts machines may have statements to express non-determinism in processes. Consider, for instance, on a positive sepsis diagnosis, a HCP may chose to either administer fluids first, followed by antibiotics, or vice-versa. MediK supports such non-determinism using `either-or` statements as shown in Fig. 8

When writing ghosts, values of measurements need to be abstract, to encompass all possible values that may be encountered during execution. Ideally, any such value can simply be represented by a logical variable that ranges over

```
1  rule #nondet + _:Val    => #nondet
2  rule _:Val   <= #nondet => #nondet
3  rule #nondet && _       => #nondet
4  rule if (#nondet) Block => Block
5  rule if (#nondet) _     => .
```

Fig. 9: Abstract Semantics

We then added a rule that takes a machine instance in an active state with an unhandled event at the head of the input buffer to a terminal *stuck* state. This permits us to K's search capability to check if the *stuck* state is reachable along any path. While in the ideal case, we would have symbolically executed the program while search for the *stuck* pattern, we ran into performance issues with K's haskell backend that at the time of writing was the only backend that supported symbolic execution. Thus, as a fallback, we use an abstraction to encompass all possible values that may be encountered

during execution. For instance, when modeling entering a parameter such as the Heart Rate, we use an abstract value to represent all possible concrete values. This also means we needed to add the following rules to the semantics to handle the abstraction.

This abstraction mechanism enables us to use $\mathbb{K}$'s LLVM backend that only supports concrete execution to search for the *stuck* state.

However, we recognize that the inability to perform symbolic reasoning is a limitation. In this proposal, we plan to address performance issues to enable symbolic execution. We also plan to prove that our abstraction-based methodology that allows us to use concrete execution is indeed sound, i.e., to show that if the abstract program satisfies the responsiveness property, then the corresponding concrete program will also satisfy it.

4.3 Systematic Modeling of Time

In [31], we were primarily focused on designing a language that could serve as a convenient medium for expressing BPGs. in this proposal, we plan to develop the requisite formal underpinings that enable reasoning about such systems.

4.4 Applicability to other CDSSs

While Medi$\mathbb{K}$ has been used to implement one real-world, complex CDSSs for management of sepsis in pediatric cases, it remains to be seen if it is a convenient medium for building CDSSs in general. To answer this, we plan to implement at least one more real-world CDSS using Medi$\mathbb{K}$. To this end, we plan to express the American Heart Institute's (AHA) BPG for management of cardiac arrest [58]. Our choice is motivated by the following reasons: (i) an effective CDSS for cardiac arrest can significantly improve outcomes, (ii) enables us to evaluate the Medi$\mathbb{K}$ code against directly encoding the guideline in $\mathbb{K}$, as we had used the same BPG for a $\mathbb{K}$-based CDSSs (see section 3.2).

5 Summary and Conclusion

Preventable Medical Errors remain a challenge in medicine, but can be mitigated through the use of Guidelines-based Clinical Decision Support Systems (CDSSs). Such systems codify Best Practice Guidelines (BPGs) and support Healthcare Providers (HCPs) with situation-specific advice. While such systems have shown to improve clinical outcomes, developing them is challenging, which limits their adoption.

Our proposal intends to address these challenges using a *semantics-first* approach. At the core of our approach is a new language for expressing BPGs in an executable manner called Medi$\mathbb{K}$. Medi$\mathbb{K}$ has a formal semantics defined in the $\mathbb{K}$-framework, and thus has a *correct-by-construction* interpreter and analysis tools.

In section 1, we identified research questions (RQs) that we plan to address through this work. We now summarize how our current progress and proposed work addresses them.

(RQ1) Language Design: How do we design a language to accomodate expressing complex BPGs that is also comprehensible to Healthcare Providers (HCPs)?

To answer this, we identified common characteristics of BPGs that are desirable of a language for expressing them in an executable manner. In [31], we explain how our language is built around these characteristics.

Our work build on the state-of-art in modeling large concurrent systems, but adapts it to specific needs of BPGs.

(RQ2) Applications: Can we build real-world CDSSs using our approach?

To answer this, we collaborated with the OSF Healthcare Children's Hospital of Illinois in Peoria to build a CDSS for management of sepsis in pediatric cases based on their BPG. Our tool has been approved by the hospital Institutional Review Board, and is slated to under clinical simulations. Thus, we argue our approach does indeed enable building real-world systems. To further answer this question, in section 4.4 we identify another BPG that we intend to encode in Medi$\mathbb{K}$, and demonstrate that our approach works for another BPG.

(RQ3) Ecosystem: Do the semantics-derived tools work well for our purposes? Can we use these tools for execution and formal analysis?

The interpreter generated from the semantics is able to satisfy performance needs of the sepsis management system. This CDSS is based on a complex BPG, involving multiple workflows that are encoded as Medi$\mathbb{K}$ machines. However, as discussed in section 4.2, we ran into performance issues while utilizing $\mathbb{K}$'s symbolic execution capabilities due to the size of the program. To get around this, we utilized an abstraction to check that the sepsis management CDSS satisfies desired responsiveness properties. We propose to prove the soundness of our abstraction-based technique and investigate using symbolic execution to further make analyzing programs easier.

In section 4.1, we propose to implement a semantics-derived solution to generate visual representations of Medi$\mathbb{K}$ programs. Such representations are vital to ensure that HCPs can comprehend Medi$\mathbb{K}$ code to validate accuracy of medical knowledge.

Thus, with these questions answered, we demonstrate an advancement over the existing state-of-art in building CDSSs discussed in section 2.2.

References Cited

[1] T. L. Rodziewicz, B. Houseman, and J. E. Hipskind, *Medical Error Reduction and Prevention*. StatPearls Publishing, Treasure Island (FL), 2022. [Online]. Available: http://europepmc.org/books/NBK499956

[2] M. S. Donaldson, J. M. Corrigan, L. T. Kohn *et al.*, *To err is human: building a safer health system*. National Academies Press, 2000.

[3] M. A. Makary and M. Daniel, "Medical error—the third leading cause of death in the us," *The BMJ*, vol. 353, 2016. [Online]. Available: https://www.bmj.com/content/353/bmj.i2139

[4] C. Andel, S. L. Davidow, M. Hollander, and D. A. Moreno, "The economics of health care quality and medical errors," *Journal of health care finance*, vol. 39, no. 1, p. 39, 2012.

[5] M. J. Field, K. N. Lohr *et al.*, *Clinical practice guidelines*. National Academies Press (US) Washington, DC, USA, 1990.

[6] E. Steinberg, S. Greenfield, D. M. Wolman, M. Mancher, R. Graham *et al.*, *Clinical practice guidelines we can trust*. National Academies Press, 2011.

[7] A. R. Panchal, J. A. Bartos, J. G. Cabañas, M. W. Donnino, I. R. Drennan, K. G. Hirsch, P. J. Kudenchuk, M. C. Kurz, E. J. Lavonas, P. T. Morley *et al.*, "Part 3: adult basic and advanced life support: 2020 american heart association guidelines for cardiopulmonary resuscitation and emergency cardiovascular care," *Circulation*, vol. 142, no. 16_Suppl_2, pp. S366–S468, 2020.

[8] A. A. Topjian, T. T. Raymond, D. Atkins, M. Chan, J. P. Duff, B. L. Joyner Jr, J. J. Lasa, E. J. Lavonas, A. Levy, M. Mahgoub *et al.*, "Part 4: Pediatric basic and advanced life support: 2020 american heart association guidelines for cardiopulmonary resuscitation and emergency cardiovascular care," *Circulation*, vol. 142, no. 16_Suppl_2, pp. S469–S523, 2020.

[9] J. P. Ornato, M. A. Peberdy, R. D. Reid, V. R. Feeser, H. S. Dhindsa, N. investigators *et al.*, "Impact of resuscitation system errors on survival from in-hospital cardiac arrest," *Resuscitation*, vol. 83, no. 1, pp. 63–69, 2012.

[10] H. A. Wolfe, R. W. Morgan, B. Zhang, A. A. Topjian, E. L. Fink, R. A. Berg, V. M. Nadkarni, A. Nishisaki, J. Mensinger, R. M. Sutton *et al.*, "Deviations from aha guidelines during pediatric cardiopulmonary resuscitation are associated with decreased event survival," *Resuscitation*, vol. 149, pp. 89–99, 2020.

[11] C. P. Crowley, J. D. Salciccioli, and E. Y. Kim, "The association between acls guideline deviations and outcomes from in-hospital cardiac arrest," *Resuscitation*, vol. 153, pp. 65–70, 2020.

[12] K. Honarmand, C. Mepham, C. Ainsworth, and Z. Khalid, "Adherence to advanced cardiovascular life support (acls) guidelines during in-hospital cardiac arrest is associated with improved outcomes," *Resuscitation*, vol. 129, pp. 76–81, 2018.

[13] M. D. McEvoy, L. C. Field, H. E. Moore, J. C. Smalley, P. J. Nietert, and S. H. Scarbrough, "The effect of adherence to acls protocols on survival of event in the setting of in-hospital cardiac arrest," *Resuscitation*, vol. 85, no. 1, pp. 82–87, 2014.

[14] C. Rand, N. Powe, A. Wu, and M. Wilson, "Why don't physicians follow clinical practice guidelines," *Journal of the American Medical Association (JAMA)*, vol. 282, p. 14581465, 1999.

[15] D. A. Davis and A. Taylor-Vaisey, "Translating guidelines into practice: a systematic review of theoretic concepts, practical experience and research evidence in the adoption of clinical practice guidelines," *Canadian Medical Association Journal (CMAJ)*, vol. 157, no. 4, pp. 408–416, 1997.

[16] S. H. Woolf, R. Grol, A. Hutchinson, M. Eccles, and J. Grimshaw, "Potential benefits, limitations, and harms of clinical guidelines," *The BMJ*, vol. 318, no. 7182, pp. 527–530, 1999.

[17] A. X. Garg, N. K. Adhikari, H. McDonald, M. P. Rosas-Arellano, P. J. Devereaux, J. Beyene, J. Sam, and R. B. Haynes, "Effects of computerized clinical decision support systems on practitioner performance and patient outcomes: a systematic review," *Journal of the American Medical Association (JAMA)*, vol. 293, no. 10, pp. 1223–1238, 2005.

[18] K. Kawamoto, C. A. Houlihan, E. A. Balas, and D. F. Lobach, "Improving clinical practice using clinical decision support systems: a systematic review of trials to identify features critical to success," *The BMJ*, vol. 330, no. 7494, p. 765, 2005.

[19] P. Bennett and N. R. Hardiker, "The use of computerized clinical decision support systems in emergency care: a substantive review of the literature," *Journal of the American Medical Informatics Association (JAMIA)*, vol. 24, no. 3, pp. 655–668, 12 2016.

[20] N. Sahota, R. Lloyd, A. Ramakrishna, J. Mackay, J. Prorok, L. Weise-Kelly, T. Navarro-Ruan, N. Wilczynski, and b. Haynes, "Computerized clinical decision support systems for acute care management: A decision-maker-researcher partnership systematic review of effects on process of care and patient outcomes," *Implementation Science (IS)*, vol. 6, p. 91, 08 2011.

[21] B. C. James, "Making it easy to do it right," *New England Journal of Medicine (NEJM)*, vol. 345, no. 13, pp. 991–993, 2001.

[22] D. R. Sutton and J. Fox, "The syntax and semantics of the pro forma guideline modeling language," *Journal of the American Medical Informatics Association (AMIA)*, vol. 10, no. 5, pp. 433–443, 2003.

[23] Y. Shahar, S. Miksch, and P. Johnson, "An intention-based language for representing clinical guidelines." in *Proceedings of the AMIA Annual Fall Symposium*. American Medical Informatics Association, 1996, p. 592.

[24] "Advanced cardiac life support algorithms." [Online]. Available: https://cpr.heart.org/en/resuscitation-science/cpr-and-ecc-guidelines/algorithms

[25] A. Rhodes, L. E. Evans, W. Alhazzani, M. M. Levy, M. Antonelli, R. Ferrer, A. Kumar, J. E. Sevransky, C. L. Sprung, M. E. Nunnally *et al.*, "Surviving sepsis campaign: international guidelines for management of sepsis and septic shock: 2016," *Intensive Care Medicine*, vol. 43, pp. 304–377, 2017.

[26] M. Eisenberg, E. Freiman, A. Capraro, K. Madden, M. C. Monuteaux, J. Hudgins, and M. Harper, "Comparison of manual and automated sepsis screening tools in a pediatric emergency department," *Pediatrics*, vol. 147, no. 2, 2021.

[27] S. L. Weiss, J. C. Fitzgerald, F. Balamuth, E. R. Alpern, J. Lavelle, M. Chilutti, R. Grundmeier, V. M. Nadkarni, and N. J. Thomas, "Delayed antimicrobial therapy increases mortality and organ dysfunction duration in pediatric sepsis," *Critical Care Medicine*, vol. 42, no. 11, p. 2409, 2014.

[28] I. V. Evans, G. S. Phillips, E. R. Alpern, D. C. Angus, M. E. Friedrich, N. Kissoon, S. Lemeshow, M. M. Levy, M. M. Parker, K. M. Terry *et al.*, "Association between the new york sepsis care mandate and in-hospital mortality for pediatric sepsis," *Journal of American Medicine (JAMA)*, vol. 320, no. 4, pp. 358–367, 2018.

[29] F. Balamuth, E. R. Alpern, M. K. Abbadessa, K. Hayes, A. Schast, J. Lavelle, J. C. Fitzgerald, S. L. Weiss, and J. J. Zorc, "Improving recognition of pediatric severe sepsis in the emergency department: contributions of a vital sign–based electronic alert and bedside clinician identification," *Annals of Emergency Medicine*, vol. 70, no. 6, pp. 759–768, 2017.

[30] R. J. Sepanski, S. A. Godambe, C. D. Mangum, C. S. Bovat, A. L. Zaritsky, and S. H. Shah, "Designing a pediatric severe sepsis screening tool," *Frontiers in Pediatrics*, vol. 2, p. 56, 2014.

[31] M. Saxena, S. Song, and L. Sha, "Medik: Towards safe guideline-based clinical decision support," in *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2023*, p. 306.

[32] "Cpr and emergency cardiovascular care algorithms." [Online]. Available: https://cpr.heart.org/en/resuscitation-science/cpr-and-ecc-guidelines/algorithms

[33] "Clinical practice algorithms." [Online]. Available: https://www.mdanderson.org/for-physicians/clinical-tools-resources/clinical-practice-algorithms.html

[34] G. Hripcsak, "Writing arden syntax medical logic modules," *Computers in biology and medicine*, vol. 24, no. 5, pp. 331–363, 1994.

[35] M. Samwald, K. Fehre, J. De Bruin, and K.-P. Adlassnig, "The arden syntax standard for clinical decision support: experiences and directions," *Journal of Biomedical Informatics*, vol. 45, no. 4, pp. 711–718, 2012.

[36] M. Peleg, A. A. Boxwala, E. Bernstam, S. Tu, R. A. Greenes, and E. H. Shortliffe, "Sharable representation of clinical guidelines in glif: relationship to the arden syntax," *Journal of biomedical informatics*, vol. 34, no. 3, pp. 170–181, 2001.

[37] A. A. Boxwala, M. Peleg, S. Tu, O. Ogunyemi, Q. T. Zeng, D. Wang, V. L. Patel, R. A. Greenes, and E. H. Shortliffe, "Glif3: a representation format for sharable computer-interpretable clinical practice guidelines," *Journal of Biomedical Informatics (JBI)*, vol. 37, no. 3, pp. 147–161, 2004.

[38] S. Bäumler, M. Balser, A. Dunets, W. Reif, and J. Schmitt, "Verification of medical guidelines by model checking – a case study," in *Model Checking Software*, A. Valmari, Ed. Springer Berlin Heidelberg, 2006, pp. 219–233.

[39] S. W. Tu, J. Campbell, and M. A. Musen, "The sage guideline modeling: motivation and methodology," in *Computer-based Support for Clinical Guidelines and Protocols*. IOS Press, 2004, pp. 167–171.

[40] N. F. Noy, M. Crubézy, R. W. Fergerson, H. Knublauch, S. W. Tu, J. Vendetti, and M. A. Musen, "Protégé-2000: an open-source ontology-development and knowledge-acquisition environment." in *AMIA... annual symposium proceedings. AMIA Symposium*, 2003, pp. 953–953.

[41] P. Terenziani, S. Montani, A. Bottrighi, M. Torchio, G. Molino, and G. Correndo, "The glare approach to clinical guidelines: main features," in *Computer-based Support for Clinical Guidelines and Protocols*. IOS Press, 2004, pp. 162–166.

[42] L. Giordano, P. Terenziani, A. Bottrighi, S. Montani, and L. Donzella, "Model checking for clinical guidelines: an agent-based approach," in *Amia annual symposium proceedings*, vol. 2006. American Medical Informatics Association, 2006, p. 289.

[43] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene, "Formal validation and verification of a medical software critical component," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, pp. 80–89.

[44] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 321–332. [Online]. Available: https://doi.org/10.1145/2491956.2462184

[45] "The k framework." [Online]. Available: https://kframework.org

[46] G. Rosu, "Matching logic," *Logical Methods in Computer Science*, vol. 13, 2017.

[47] X. Chen and G. Rosu, "Matching mu-logic," in *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2019, pp. 1–13.

[48] ——, "Applicative matching logic," 2020, https://hdl.handle.net/2142/104616.

[49] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the Ethereum Virtual Machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.

[50] "Ethereum yellow paper." [Online]. Available: https://yellowpaper.org

[51] "Aleth - ethereum c++ client, tools and libraries." [Online]. Available: https://github.com/ethereum/cpp-ethereum

[52] "Ethereum jello paper." [Online]. Available: https://yellowpaper.org

[53] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," WSTC17, International Conference on Financial Cryptography and Data Security, 2017.

[54] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: reusable engineering of real-world semantics," in *ACM SIGPLAN Notices*, vol. 49, no. 9. ACM, 2014, pp. 175–188.

[55] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A Formal Verification Tool for Ethereum VM Bytecode," in *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, November 2018, pp. 912–915.

[56] Ethereum, "Ethereum application binary support," 2017, https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI.

[57] M. Saxena, X. Chen, S. Song, S. Meng, L. Sha, and G. Rosu, "Rewriting-based computer interpretable best practice guidelines," 2022, https://hdl.handle.net/2142/116016.

[58] "American heart association emergency cardiovascular care." [Online]. Available: https://cpr.heart.org/en/resuscitation-science/cpr-and-ecc-guidelines/algorithms

[59] "Ecma-404 the json data interchange standard." [Online]. Available: https://www.json.org/json-en.html

[60] J. Koppel, J. Kearl, and A. Solar-Lezama, "Automatically deriving control-flow graph generators from operational semantics," vol. 6, no. ICFP. ACM New York, NY, USA, 2022, pp. 742–771.

[61] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.