**Abstract**—Clinical Best Practice Guidelines (BPGs) are systematically developed, evidence-based statements published by medical institutions and associations that standardize diagnosis and treatment for various clinical scenarios. When expressed in an executable medium, BPGs can be utilized to build systems that assist healthcare professionals (HCPs) with situation-specific advice. Such systems, known as Guideline-based Clinical Decision Support Systems (CDSSs), have been shown to improve patient outcomes.

Several Domain-Specific Languages (DSLs) have been proposed to facilitate expressing BPGs in a computer-interpretable format that is easily comprehensible to HCPs. Given the safety-critical nature of CDSSs, the need for such languages to have complete formal semantics and an ecosystem of formal analysis tools has been recognized. Moreover, since these languages evolve over time to accommodate complexities in modeling BPGs, tools for them must also be adaptable to changes. But, existing languages lack complete formal semantics, or analysis tools derived from them.

This work introduces MediK: a new DSL for expressing BPGs with a complete executable formal semantics, and formal analysis tools, including a model checker, symbolic execution engine, and deductive verifier. As MediK's tools are derived from its semantics, any update to the language is automatically reflected across all tools. To evaluate our approach, we collaborated with a major pediatric hospital to develop a MediK-based CDSS for the screening and management of Pediatric Sepsis and validated that it satisfies desired safety properties. Our CDSS is Institutional Review Board (IRB) approved and is slated to undergo clinical simulations.

**Index Terms**—Semantics, Model checking

I. 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]–[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].

A guidelines-based CDSS usually consists of: (a) a translation of the guideline to an executable medium, called the knowledge-base, (b) an interface for user-interaction, and, (c) additional infrastructure that integrates with external data sources such as sensors, health records [22]. Typically, to develop a CDSS, domain experts in medicine collaborate with computer scientists to develop requirements documentation that presents the BPGs’s semantics in a manner amenable to software development [23]. This documentation is then utilized to develop the knowledge-base, which is subsequently integrated with data sources (such as patient-parameter sensors.
and health records), and a User Interface (UI) to obtain a complete system. Thus, the BPG serves as a functional specification for the CDSS’s knowledge-base. But, the aforementioned process has several limitations. First, the implementation, i.e., the knowledge-base may not concur with its specification, i.e., the text-based BPG. BPGs are specified as long, complex textual documents, where the exact meaning of terms may not be explicitly stated, and recommendations may be ambiguous [24]. Capturing and communicating these complexities via requirements documentation is challenging, and incorrect or incomplete documentation has resulted in failed implementations [25]. Second, as BPGs evolve to reflect new evidence or local adaptions, corresponding updates must be made to the CDSS as well. However, due to the gap between the BPG and the knowledge-base, effort must be expended into bringing the knowledge-base to reflect said updates.

To address above mentioned limitations, several Domain Specific Languages (DSLs) for directly expressing knowledge-base as Computer Interpretable Guidelines (CIGs) have been introduced. By providing mechanisms to facilitate representation of medical knowledge, such DSLs allow the CIG to serve as both the system specification, i.e. the BPG, and implementation, i.e. the knowledge-base. This ensures that there is no gap between the BPG and its executable counterpart. Given the safety-critical nature of CDSSs, the need for formally verified execution engines and analysis tools has been recognized. To this end, some existing DSLs have partially-defined semantics and support for verification via model-checking. However, as identified by the authors of [26], [27], existing languages lack a complete formal and executable semantics, interpreters or compilers with correctness guarantees, and a comprehensive suite of accompanying tools such as model-checkers, symbolic-execution engines, and deductive verifiers. The difficulties of formal analysis are further compounded by the fact that CDSSs are concurrent systems involving interactions with heterogeneous external agents such as sensors and HCPs, making their analysis challenging. We address these by introducing MediK (pronounced Medi-kay), a DSL for expressing a CDSS’s knowledge-base as concurrently-executing state machines. MediK provides:

1) A complete executable formal semantics specified in the semantics framework.
2) A correct-by-construction interpreter, and analysis tools such as a model-checker and deductive verifier.
3) A uniform way of modeling heterogeneous agents for both execution and analysis.

To evaluate our approach, we worked with the Children’s Hospital of Illinois at OSF St. Francis Medical Center (referred to as OSF in the remainder of this work) to develop a CDSS for their pediatric sepsis management guidelines. The MediK-based system expresses the guideline succinctly, and allows establishing desired safety properties. To the best of our knowledge, ours is the first system for sepsis management with a set of safety guarantees.

We briefly describe the organization of this paper. In section II, we present a real-world BPG for management of sepsis, and use it to illustrate requirements that a DSL for encoding clinical guidelines must satisfy. In section III, we describe the MediK DSL, and illustrate how it addresses aforementioned requirements. To evaluate our approach, we utilized MediK to implement a real-world CDSS for pediatric sepsis management, which we describe in Section IV. In section V, we discuss how MediK builds on existing work, mention directions for future work in VI, and conclude in section VII.

II. Motivating Example

In this section, we introduce a real world BPG for management of sepsis in pediatric cases to motivate the need for Guidelines-based Clinical Decision Support Systems, and to illustrate characteristics that are desired of a DSL for such systems.

Sepsis is life-threatening condition caused by the body’s extreme response to an infection [28], and is a major cause of morbidity and mortality in children [29]. Adverse outcomes can, however, be mitigated through timely identification and prompt treatment with antibiotics and intravenous (IV) fluids [30], [31]. BPGs for screening and management of sepsis in pediatric Emergency Departments (EDs) have shown effectiveness in screening and management of sepsis [29], leading to their adoption in many pediatric EDs [32], [33].

In Fig. 1, we present a simplified version of the screening section of OSF’s sepsis management guideline. In essence, when a patient arrives at the ED with a fever or an infection, the 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 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 I, 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, fluid resuscitation, and administering antibiotics. In Fig. 2, we provide a version of the fluid resuscitation guideline used at OSF. Briefly, if the patient is flagged as potentially septic, the guideline suggests (i) obtaining any fluid-overload risks, (ii) administering normal saline (typically over a period of 15 minutes), where the dosage is dictated by risks determined in previous step, (iii) assessing signs of fluid-overload, (iv) evaluating patient responsiveness to normal saline upon completion of the administering process, and, (v) determining whether another fluid bolus should be administered based on information from previous steps.

This real-world BPG exhibits characteristics common across

<table>
<thead>
<tr>
<th>Age</th>
<th>Heart Rate</th>
<th>Systolic BP</th>
<th>Temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0d – 1m</td>
<td>≥ 130</td>
<td>&lt; 60</td>
<td>&lt; 36  or &gt; 38</td>
</tr>
<tr>
<td>≥ 1m – 3y</td>
<td>≥ 130</td>
<td>&lt; 70</td>
<td>&lt; 36  or &gt; 38</td>
</tr>
<tr>
<td>≥ 3m – 1y</td>
<td>&gt; 190</td>
<td>&lt; 70</td>
<td>&lt; 36  or &gt; 38.5</td>
</tr>
<tr>
<td>≥ 13y</td>
<td>≥ 100</td>
<td>&lt; 90</td>
<td>&lt; 36  or &gt; 38.5</td>
</tr>
</tbody>
</table>

TABLE I: Vital Signs Chart

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 [34] and [35] 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.

Note that the aforementioned characteristics are not specific to one guideline. According to a review paper on CIGs [24], such DSLs should additionally (a) be formally defined, i.e, have a formal syntax and semantics, and (b) have an execution engine to provide decision support.

In the following sections, we describe how these characteristics dictate the design philosophy behind MediK. We argue that this philosophy makes MediK both intuitive to HCPs, and suitable for expressing complex guidelines.

III. MediK

In this section, we introduce the MediK DSL for expressing CIGs. MediK has designed to describe knowledge-base used in safety-critical systems. Thus, it is vital that:

- The interpreter is correct w.r.t. the formal semantics.
- The language has a comprehensive suite of formal program analysis tools.
- New features based on HCP feedback can be implemented quickly, conveniently, and correctly.

We achieve this by defining MediK (i.e., its syntax and semantics) in K. K is a rewriting-based framework for defining executable semantics of languages, type systems and formal analysis tools. It has been successfully used to define executable semantics of many real world languages such as C [36], Java [37], Javascript [38], and the Ethereum Virtual Machine [39]. We will introduce K by need while discussing
Medi\(\mathcal{K}\). For more details on \(\mathcal{K}\), we refer the reader to [40] [41].

The \(\mathcal{K}\) ecosystem provides a suite of tools, such as an interpreter, model-checker, and deductive verifier that are parametric over the language’s semantics, as shown in Fig. 3. Thus, by defining the semantics of Medi\(\mathcal{K}\) in \(\mathcal{K}\), we obtain aforementioned tools for it without any extra effort. Additionally:

- The \(\mathcal{K}\)-based interpreter for Medi\(\mathcal{K}\) essentially executes the language’s semantics rules, it is correct-by-construction.
- Incorporating changes to Medi\(\mathcal{K}\) only requires updating the semantics. Since the tools are derived from the semantics, they’re automatically updated.

The remainder of this section introduces Medi\(\mathcal{K}\) and describes how it’s designed around characteristics of BPGs from Section II. Recall that BPGs typically involve concurrent workflows, often expressed using a flowchart-like notation that may involve inter-workflow interactions. To ensure Medi\(\mathcal{K}\) programs are comprehensible to HCPs, they must be representable in a flowchart-like notation that HCPs are already comfortable with, and be capable of expressing inter-workflow interactions succinctly. To address these diverse requirements, we borrow from existing state-of-art languages for modeling large concurrent systems, like P [42], but make adaptions to make expressing and validating BPGs easier. We explore the differences to existing techniques in section V. In Medi\(\mathcal{K}\), 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\(\mathcal{K}\). 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.

Next, we describe Medi\(\mathcal{K}\) using its \(\mathcal{K}\)-framework definition. The \(\mathcal{K}\) definition of a language has two components. The first is the language’s syntax, which is defined using a BNF-like notation. \(\mathcal{K}\) utilizes this grammar to generate a parser for programs in the language. We describe Medi\(\mathcal{K}\)’s syntax in depth in Section III-A. The second is the semantics, which is defined using a \(\mathcal{K}\)-configuration and rewrite rules. The \(\mathcal{K}\)-configuration organizes the program’s execution state. Rewrite rules that operate over said configuration dictate the evolution of program state during execution. We describe the semantics in greater depth in Section III-B\(^1\)

### A. Syntax

We use the skeleton of a Medi\(\mathcal{K}\) machine, and use it to describe the syntax. Note that we use [...] to denote optional constructs, <...> for mandatory constructs, lowercase for terminals, and uppercase for non-terminals.

```plaintext
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       on <IDENTIFIER> ([IDENTIFIER_LIST]) do { 
9         <STMT> // event handler 
10       } 
11     } 
12 } 
13
```

A Medi\(\mathcal{K}\) 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 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.

Within the entry and event handler code blocks, there may be statements. Below, we give a simplified version of the \(\mathcal{K}\)-grammar for statements. In \(\mathcal{K}\), productions are defined using the keyword syntax (lines 1, 7). Terminals are enclosed in quotes (""), and non-terminals begin with an uppercase character.

```plaintext
1 syntax Exp ::= Id | Val | "this" 
2   | Exp "." Exp 
3   | "obtainFrom" "(" Exp "," Exp ")" 
4   | "interval" "(" Exp "," Exp ")" 
5   | Exp "." Exp 
6   | Exp "+" Exp 
7 syntax Stmt ::= Exp "=" Exp 
8   | "if" "(" Exp ")" Block else Block 
9   | "new" Id "(" Exp ")" 
10   | "createFromInterfaces" "(" Id "," String ")" 
11   | "sleep" "(" Exp ")" 
12   | "send" Exp 
13   | "broadcast" Id 
14   | "goto" Id "(" Exp ")" 
15   | Exp "+" 
```

Lines 1-5 define the syntax of Medi\(\mathcal{K}\) expressions. Line 1 defines basic expressions such as identifiers (denoted by the

\(^1\)The complete executable semantics is available at [43].
_builtin \texttt{K} production \texttt{Id}), values such as booleans, or rationals, or \texttt{"this"}, which enables an instance to refer to itself. Line 2 defines the usual dot operator (.), which can be used to access members of an instance. obtainFrom (line 3), interval (line 4) and \texttt{in} (line 5) are useful in context of defining BPGs, and are described through an example in section IV. Apart from these, Medi\texttt{K} also supports common expressions such as +, -, >, >= over rationals and &&, || over booleans.

In lines 7-15, we define syntax for Medi\texttt{K} statements. Some of these, such as variable assignment (line 7), if-else (line 8) and \texttt{new Id(..);} (line 9) are commonly found in other languages, and have expected meanings. The remaining statements (lines 10-15) have nuanced meanings in context of state machines. We shall go over these while discussing Medi\texttt{K}’s semantics in section III-B.

B. Semantics

Semantics of a language defined in \texttt{K} has two components: (1) description of program state via \texttt{K}-configurations, and (2) \texttt{K} rules that dictate state evolution. Next we describe these components in detail.

1) \texttt{K}-Configuration: \texttt{K} represents program execution state using \texttt{K}-configurations. A \texttt{K}-configuration is an unordered list of (potentially nested) cells, specified using an XML-like notation. When declaring rules (as rewrites) over this state, any subset of the cells present in the configuration can be mentioned. This allows specifying only necessary parts of the state for a given rule, letting \texttt{K} assume that the rest of the configuration remains unchanged. The following configuration defines the initial state for any Medi\texttt{K} program:

```plaintext
configuration
  <instance multiplicity="*" type="Map"> ...
  <k> createMachineDefs($SPGM)
  <!-- createInitInstances </k>
  <env> .Map </env>
  <inBuffer> .List </inBuffer>
  <activeState> . </activeState>
  </instance>
<machine multiplicity="*" type="Map"> ...
  <machineName> </machineName>
  <states> ...
  <state multiplicity="*" type="Map">
    <stateName> </stateName>
    <entryBlock> </entryBlock>
    <eventHandlers> </eventHandlers>
  </state>
  </states>
</machine>
```

The keyword \texttt{configuration} (line 1) defines a \texttt{K}-configuration, followed by xml-like notation for the \texttt{K}-cells. For example \texttt{<foo> ... </foo>} corresponds to a \texttt{K}-cell with the name \texttt{foo}. The \texttt{<instance>} cell (lines 2-9) contains state of each Medi\texttt{K} machine instance during execution. Each instance manages its instance variables using a map in the \texttt{<genv>} cell (line 5), a buffer of incoming events in the \texttt{<inBuffer>} cell (line 7) and the currently executing code in the \texttt{<k>} cell (lines 3-4).

When a Medi\texttt{K} program is executed, \texttt{K} replaces \texttt{$SPGM} (line 3) with the Abstract Syntax Tree (AST) of the program, obtained by parsing the program using the syntax from section III-A. The \texttt{createMachineDefs} construct is defined (using rewrite rules) to traverse the program AST and populate the configuration with information related to each machine. The \texttt{createInitInstances} creates an instance for the machine with the \texttt{init} keyword, leading to execution of the initial machine’s entry block. Note that \texttt{->} symbol (line 3) is interpreted by \texttt{K} as “followed-by”, i.e., execution of \texttt{createMachineDefs} is followed by execution of \texttt{createInitInstances}. The attribute \texttt{multiplicity="*"} on lines 2 and 10 signifies that multiple copies of the corresponding cells, in this case \texttt{<machine>} and \texttt{<instance>} cells, can exist in the configuration during execution. This allows, during execution, for multiple machine definitions, each with multiple instances, to exist. The \texttt{<machine>} cell (lines 10-19) holds information relevant to a machine definition, such as the name in the \texttt{<machineName>} cell (line 11) and states in the \texttt{<states>} cell (lines 12-18). The \texttt{<state>} (lines 13-17) holds information relevant to a state, such as the entry block in cell \texttt{<entryBlock>} (line 15) and event handlers in cell \texttt{<eventHandlers>} (line 16).

2) \texttt{K}-Rules: \texttt{K}-rules operate over the configuration and define the evolution of program state during execution. A \texttt{K}-rule begins with the keyword \texttt{rule}, and is a statement of the form \texttt{\varphi \Rightarrow \psi}, where \texttt{\varphi} and \texttt{\psi} are patterns over configuration terms and \texttt{K}-variables. We say \texttt{\varphi} is the LHS and \texttt{\psi} is the RHS of the rule. Let substitution \texttt{\theta} be a map from \texttt{K}-variables to terms. Say, for given pattern \texttt{\varphi} and substitution \texttt{\theta}, \texttt{\theta \varphi} be the term obtained by replacing each variable \texttt{v} in \texttt{\varphi} with \texttt{\theta(v)}. During execution, if the current configuration \texttt{C}, i.e., program execution state, matches \texttt{\varphi} with substitution \texttt{\theta}, then it is rewritten to \texttt{\psi \theta}. We say pattern \texttt{\varphi} matches configuration \texttt{C} iff there exists a substitution \texttt{\theta} s.t. \texttt{C = \varphi \theta}. For example, consider the following rule for updating the value of a local program variable:

```plaintext
rule <k> I:Id = V:Val \Rightarrow V ... </k>
  <env> (I \rarr \theta Loc) ... </env>
  <store> Store \Rightarrow Store[Loc \rarr V] </store>
```

Here, \texttt{i}, \texttt{v}, \texttt{Loc}, and \texttt{Store} are \texttt{K}-variables. Note the distinction between program variables and \texttt{K}-variables: while program variables are simply identifiers, \texttt{K}-variables have logical meaning. The \texttt{...} is used to denote parts of the configuration not relevant to the rule. Typically, the top of the \texttt{k} cell contains the statement currently being executed. Suppose we’re executing the statement \texttt{i = 2};. In this case, the current configuration will have a \texttt{k} cell of the form \texttt{<k> i = 2 ... </k>}, an environment cell \texttt{env} where variable \texttt{i} maps to some pointer \texttt{p}, and a store cell \texttt{store} containing a map \texttt{M} with some value pointed-to by \texttt{p}. The LHS matches with substitution \texttt{\theta = (I \rarr i, V \rarr 2, Loc \rarr p, Store \rarr M)}, resulting in the top of the \texttt{k} cell to be rewritten to the value 2, and pointer \texttt{p} updated to point to 2 in \texttt{M}. Note if there exist multiple rules that can match the current configuration, then one rule is non-deterministically chosen and applied. An execution is a sequence of rule applications that continues until no rule matches the configuration.
In the following sections we present several MediK constructs relevant to defining BPGs using their K-rules. We first present the rule for sending and receiving messages.

```
rule
<instance>
  <k> send instance(RecvId) , EventName:id , { Args }
  => done ... <k>
</instance>
<instance>
  eventArgsPair(EventName | Args | Epoch + 1)
  => ListItem
</instance>
<instance>
  <id> RecvId </id> ....
  <inBuffer> ... (List
  => ListItem
</inBuffer> ...
</instance>
<instance>
  <epoch> Epoch </epoch>
```

When the top of the k cell has send, the rule above (i) obtains the id of the receiver instance, the event name and the event arguments by matching the variables RecvId, EventName and Args against the current configuration (line 3), (ii) rewrites the top of the k cell (line 4) to done, marking the completion of execution for the construct, (iii) adds the event and associated arguments to the buffer of incoming events (lines 8-12) of the instance with id RecvId (line 7). The epoch decides when the machine can run, and is discussed in Section III-B2a.

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 following code shows the process of obtaining external measurements in MediK.

```
interface HeartRateSensor {}

machine TreatmentMachine { ...
  var hrSensor = createFromInterface(HeartRateSensor, *heartRateSensor*);
  var heartRate = obtainFrom(hrSensor, *heartRate*);
  ...
```

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 interface 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 [44] 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.

Next, we describe the rule for supporting tables in MediK. Once a measurement, such as the heart rate has been obtained from a sensor, we need to use a table, such as TABLE I to check if the measurement is within a normal range. In MediK, we can write a function that does the required check, as shown in Fig. 4. In the code, if the age lies in any of the intervals (closed on the left, open on the right) on lines 3-5, the corresponding statement to the right of the colon (:) is run. Otherwise line 6 is run. In MediK, the following rules are responsible for assigning semantical to the in-interval construct:

```
rule E in interval(L, U) => (E >= L) && (E < U)
[macro]
rule E in ( interval(L, U) : S : Stmt C : CaseDecl ) => if (E in interval(L, U)) {S} else (E in { Cs })
[macro-rec]
```

Note the rules above are marked with the attributes macro (line 2) or macro-rec (line 5). This specifies that these constructs are not part of the language’s semantics, but merely syntactic sugar. On line 1, we specify that E in interval(L, R) desugars to checking the expression e is between the lower and upper bound L and U respectively. Similary we desugur each case statement to an if-else statement. In lines 3-5, we say that if the expression E is in interval with lower and upper bounds L and U respectively, then execute S, otherwise check E against the remaining cases Cs. Note the postfix -rec after macro specifies that the rule applies recursively, to desugur the remaining case statements.

a) MediK Scheduling Semantics: Since the K-generated interpreter is single-threaded, MediK employs interleaving-semantics for concurrency, using a single executor thread shared between machine instances. A machine instance that is either at the start of an entry block, or has an event in the input buffer that it can handle is said to be enabled, i.e. one that can run once the executor becomes available. But, a naive strategy that non-deterministically chooses one enabled machine instance may lead to unfairness. Specifically, there may be situations where a machine instance is enabled but is never chosen for execution. Therefore, to ensure fairness, we use a scheduling strategy based on a monotonically increasing global counter called the epoch. We show this execution strategy in Fig. 5.

Recall from Section III-A that a MediK program consists of a set of machines, of which one, prefixed with the keyword init, is the initial machine. Each machine has one state prefixed with init, referred to as the initial state. Let $P = \{M_0, M_1, \ldots, M_{n-1}\}$ be a program with $n$ machines, where $M_0$ is prefixed with init. MediK allows instances of a machine to be created dynamically at runtime. For machine $M_i \in P$, let $I_{M_i,j-1}$ be its $j$-th instance.
1  epoch ← 0
2  scheduled ← \{I_{M_i,0}\}
3  while scheduled ≠ \emptyset do
4    I_{M_i,j} ← choose (scheduled) s.t.
5     \(\tau \leq \text{epoch} \land \text{enabled}(I_{M_i,j})\)
6    scheduled ← scheduled \setminus I_{M_i,j}
7    execute(I_{M_i,j}, scheduled)
8    if \(\exists i', j', \tau'\) s.t. \(I_{M_{i'},j'} \in \text{scheduled}\)
9     \& \(\tau' \leq \text{epoch}\) \& \((\text{enabled}(I_{M_{i'},j'}))\) then
10       epoch ← epoch + 1
11  end if
12  end while

Fig. 5: MediK Scheduling Semantics

Execution begins in epoch zero with the implicit (first) instance of the initial machine, denoted by \(I_{M_i,0}\). We use \(I_{M_i,j-1}\) to say that the j-th instance of machine \(M_i\) is scheduled for execution in epoch \(\tau\). Recall that a state definition may have an entry block, containing code that is executed when the state is entered, or event handlers containing code that is executed when an event is dequeued from the input buffer. When execution begins, the entry block of the initial state of the implicit instance of the initial machine \(I_{M_i,0}\) becomes scheduled (line 2) at epoch 0. On line 4, an instance \(I_{M_i,j}\) is non-deterministically chosen from all machines that are both scheduled to run when \(\tau \leq \text{epoch}\) and \text{enabled}. We use \text{execute}(I_{M_i,j}, scheduled) on line 5 to denote this execution process. Execution of the entry or event handler block is atomic, i.e., a context-switch can only occur at the end of the block. Note that when a new instance of a machine is created using the keyword new, the entry block of the initial state of the target machine is executed synchronously before control returns to the source machine, and the instance is added to the multiset of scheduled machines. A context switch only occurs in three cases: goto, sleep, and obtainFrom, which we describe later.

During execution, if an instance \(I_{M_i,j}\) sends an event to another instance \(I_{M_{i'},j'}\), then the event is scheduled to be handled by \(I_{M_{i'},j'}\) in or after the next epoch, i.e., scheduled ← scheduled \cup \{I_{epoch+1, M_{i'},j'}\}. Similary, if a goto statement is encountered, the entry block of the target state is scheduled for execution at epoch + 1. If no other machine is both scheduled to run in the current epoch, and enabled, then the epoch advances by one (line 8).

b) Timer Semantics: Next, we discuss how MediK handles temporal aspects of BPGs. For instance, consider the Fluid resuscitation guideline BPG from Section 2. After administering fluids, the BPG recommends waiting for 15 minutes before evaluating their effectiveness. This waiting behavior in MediK is implemented using a sleep(duration) statement. Formalizing the execution semantics of such a statement in K presents a challenge as K does not provide built-in support for timers. Therefore, in MediK, sleep(duration) is described by the following rule:

```plaintext
1  rule <k> sleep(Duration:Int) ;
2  => jsonWrite( { "action" : "sleep" , "duration" : Duration , "tid" : Tid } )
3  <inBuffer>
4  (<ListItem(event($SleepDone | TId | Tau)) ) ...
5  </inBuffer>
6  <executorAvailable>
7  true => false
8  </executorAvailable>
9  <epoch> Epoch </epoch>
10  requires Tau <= Int Epoch
```

The \text{waitForSleepResponse}(TId) blocks execution until the external response indicating the expiration of the sleep timer is received in the input buffer (line 4). Once the response is received, the machine instance resumes execution when (a) the execution lock becomes available (indicated by true on line 8), and, (b) the epoch the instance was scheduled in (line 4) is less than or equal to the current epoch (lines 10-11).

An obtainFrom statement also results in a context switch. Just as in the case of sleep, a json message is sent to the remote endpoint, while the machine instance release the execution lock, and waits for a response. Once data for the requested field is available, it’s communicated as an event to the MediK process, and the machine resumes execution.

IV. EVALUATION

A. Sepsis Management CDSS

To evaluate our approach, we collaborated with the Children’s Hospital of Illinois at OSF St. Francis Medical Center to develop a MediK-based CDSS for screening and management of Pediatric Sepsis 3.

3the entire CDSS for sepsis management is available at [45].
Recall from Fig. 1 the guideline for sepsis screening. In Fig. 6, 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, 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. 4 to arrive upon a diagnosis.

Recall from Section II that once a sepsis diagnosis has been arrived upon, one of the guideline suggested actions include administering fluids as shown in Fig. 2. In Fig. 7, we show the corresponding MediK code for administering fluids. The process starts when an external StartFluidTherapy event, corresponding to a button press by the HCP is received (line 6). The next steps include (a) obtaining any risks associated with administering fluids (lines 11-13), (b) suggesting an appropriate dose to administer based on the risks, if any (lines 15-17), and, (c) waiting for the HCP to confirm that the suggested dose was administered (line 21). Once the dose is administered, the machine waits for the for 15 minutes as specified by the guideline (line 22), before prompting the HCP to evaluate the patient’s responsiveness to the administered fluid dose (lines 27-38), and check for any signs of fluid...
overload (lines 39-41). If the patient exhibits any signs of fluid overload, then a recommendation to handle the overload is made (line 47). Otherwise, the total dose of administered fluid is obtained from an external source (line 50). If the total dose is above the maximum allowed dose, then a recommendation based on the patient’s responsiveness to administered fluids is made to either (a) reduce the fluid flow to maintenance levels (line 53), or, (b) switch to inotropic support to address circulatory issues is made (lines 52-58). If the total dose of administered fluids is less than the maximum allowed limit, then a recommendation to administer one more fluid bolus is made (lines 58-60).

Note that both the SepsisScreening and FluidTherapy machines structurally resemble their paper based counterparts in Fig. 1 and Fig. 2 respectively, making it easier for Healthcare Providers to comprehend and validate the code.

B. Formal Analysis using MediK

During execution of a MediK program, a machine may be considered stuck if an event at the head of its input buffer does not have an associated handler, rendering said machine non-responsive. For this reason, languages for modeling large concurrent systems, such as P [42] raise an exception for unhandled events. To mitigate such exceptions, we can enforce every machine to define event handlers for all possible events in all states, and use static analysis to detect possible violations. But, for MediK programs, we found that for complex CDSSs, such as the one for screening and management of sepsis: (a) it’s tedious and error prone to define handlers for every event in every state, and, (b) it reduces the comprehensibility of the program, as many spurious event handlers that may never fire during execution have to be specified.

Thus, for MediK, we employ a weaker notion of responsiveness. We verify that every event that a state may possibly receive during execution must have a handler defined for it. This presents a challenge for reactive systems, or systems involve interactions with the external world, such as MediK-based CDSSs, as exploring the system’s state space requires modeling the external components. In MediK, we address this by specifying external components as ghost machines - a technique also used by other state machine formalisms such as P [42]. 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 follows:

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

When writing ghosts, values of measurements need to be abstract, to encompass all possible values that may be encountered during execution. For instance, when modeling entering a parameter such as the Heart Rate, we need to use an abstract value, representing all possible concrete values. To this end, we allow using an abstract value #nondet in ghost machines, with the following abstract semantics:

```java
  rule #nondet + _:Val => #nondet
  rule _:Val <= #nondet => #nondet
  rule #nondet &:< _ => #nondet
  rule if (#nondet) Block => Block
  rule if (#nondet) _ => .
```

The use of abstract encodings leads to a reduction of the state space. Recall from Section II that we needed to: (1) utilize patient’s basic information such as age and weight to calculate normal ranges for clinical measurements such as blood pressure and heart rate, and, (2) calculate abnormality in clinical measurements using aforementioned ranges. For example, determining whether the patient’s heart rate is abnormal is performed using the in-interval construct as show in Fig. 4. Recall that the in-interval construct is merely syntactic sugar for nested if-else statements. When using ghost machines for model checking, since the actual measurement is an abstract value, we know the final result of this abnormality checking operation is an abstract boolean value. Thus, instead of exploring each branch of if-else statements corresponding to in-interval constructs in Fig. 4, we replace the entire checking process with an final abstract boolean value. This reduces the state space but still allows us to explore all treatment options for both the normal and abnormal cases.

C. Model Checking the Sepsis CDSS

To verify responsiveness of the Sepsis CDSS, we implemented ghost machines for the external components using support for non-determinism and abstract values. We then added the following rule to the semantics, that takes a machine in an active state with an unhandled event at the head of the input buffer to a terminal stuck state.

```java
  rule
  <k> handleEvents -> _ => stuck </k>
  <activeState> ActiveState </activeState>
  <class> MachineName </class>
  <inBuffer>
    ListItem({event(InputEvent | _ | _ )} ...)
  </inBuffer>
  stuck
```

We utilize the semantics-generated bounded model checker to search the state space to a depth of 300,000 for a stuck pattern, i.e., a machine that’s no longer responsive. This depth we used was adequate for a complete run of the both the fluid and antibiotics machines simultaneously. The search command was executed on a machine with 64 GB of memory, and took roughly 90 minutes, and reported no such state was possible. To the best of our knowledge, this makes ours the first system for screening and management of sepsis with some formal safety guarantees.
V. Related Works

We broadly classify existing work into two categories: (a) languages/DSLs specifically developed to express BPGs in executable format, and, (b) languages/DSLs with a more general focus on expressing asynchronous event-driven systems. We first focus on category (a). The Arden Syntax [46] is a widely used medium for expressing CIGs. Guidelines are described using Medical Logic Modules that contains information related to guideline’s purpose, maintenance, and medical knowledge. 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) [47]. Arden Syntax’s limitation in modeling complexity is addressed by GLIF [48]: a language that uses flowcharts to express 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, i.e. the specification, and the CIG, i.e. 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. CIGs expressed in them are meant to be shared across hospitals, and are thus modular. However, neither formalism has complete formal semantics, or a comprehensive suite of formal analysis tools.

The need for formal analysis is identified by Asbru: a formalism with formally defined syntax and semantics [27]. 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 subplans. Apart from an execution engine, the Asbru ecosystem also contains other tools, such as a model checker for verification [49]. However, the formal semantics of Asbru have been only partially defined, and is insufficient to implement tools for the language [26]. The importance of a complete formal-semantics is identified and addressed by PROforma [26], 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) [26]. But, despite having complete formal semantics, PROforma’s semantics is not executable. Therefore, an interpreter and analysis tools have to be implemented in an ad-hoc manner. Our work builds on these existing languages, and addresses their shortcomings by utilizing a semantics-first approach to build a DSL for expressing CIGs. This provides MediKK, with a complete, executable semantics, and a suite of correct-by-construction tools derived from it, such as an interpreter, model checker and deductive verifier.

Next we look at existing work for defining large concurrent systems as State Machines. The closest project to this work, is probably the P language [42]. While P was considered for this project, it was given up for the lack of an executable semantics that would allow the language to quickly evolve to incorporate physician feedback. Moreover, until recently, P didn’t even have a symbolic execution, or executable semantics based tools that can be derived automatically from the executable semantics, features that we plan to use in future work.

VI. Future Work

In this work, we introduced MediKK, the first step towards building safe CDSSs. While MediKK has been used to implement and analyze a real system, we’re aware of many challenges that need addressing. Specifically (a) ghost machines may provide the ideal scenario for behavior of external agents, and may not take factors such as uncertainties into account, (b) the need to move beyond bounded model checking, and using deductive verification capabilities of K, (c) using symbolic execution to precisely trim unnecessary interleavings, and, (d) using semantics based compilation to extract inform, such as HCP-friendly diagrams from the code itself.

VII. Conclusion

Guideline-based Clinical Decision Support Systems (CDSSs) are now considered vital to the future of Medical Decision making in general, But, to find widespread adoption, guideline-based CDSSs must be held to the highest standards for safety-critical systems. While several advances have been made to CDSS over the years, several limitations have also been identified. This work fixed said limitations by introducing MediKK - a new DSL for expressing BPGs that uses a semantics-first approach to build CDSS. MediKK programs consist of concurrently executing instances of State Machine. MediKK models external agents as machines with transition systems external to the program called interfaces, allowing for a uniform way of dealing with heterogeneous external agents. For program analysis, MediKK allows modeling external agents via ghost machines that support non-determinism, enabling model-checking CDSSs for responsiveness. We collaborated with the Children’s Hospital of Illinois at OSF St. Francis Medical Center to develop a system for screening and management of pediatric sepsis using MediKK, and demonstrated it satisfies desired safety properties. To the best of our knowledge, our is the first system for sepsis management with any formal safety guarantees.

Acknowledgement

The authors would like to thank Dr. Johnathan A. Gehlbach and Dr. Paul M. Jeziorczak at the Children’s Hospital of Illinois at OSF St. Francis Medical Center for their assistance with the pediatric sepsis CDSS. The authors extend their gratitude to Dr. Gehlbach for his assistance with evaluating the strengths and limitations of the MediKK DSL.


