Formal Methods in Computer-Aided Design 2023

Medi$\mathbb{K}$: Towards Safe Guideline-based Clinical Decision Support

Manasvi Saxena
University of Illinois
Urbana, IL, United States
msaxena2@illinois.edu

Shuang Song
University of Illinois
Urbana, IL, United States
shuangs3@illinois.edu

Lui Sha
University of Illinois
Urbana, IL, United States
lrs@illinois.edu

***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 Medi$\mathbb{K}$: 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 Medi$\mathbb{K}$'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 Medi$\mathbb{K}$-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

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1545002

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_39

This article is licensed under a Creative Commons Attribution 4.0 International License

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 Medi$\mathbb{K}$ (pronounced Medi-kay), a DSL for expressing a CDSS's knowledge-base as concurrently-executing state machines. Medi$\mathbb{K}$ provides:

1) A *complete executable formal semantics* specified in the $\mathbb{K}$ 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 Medi$\mathbb{K}$-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-word 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 Medi$\mathbb{K}$ DSL, and illustrate how it addresses aforementioned requirements. To evaluate our approach, we utilized Medi$\mathbb{K}$ to implement a real-world CDSS for pediatric sepsis management, which we describe in Section IV. In section V, we discuss how Medi$\mathbb{K}$ 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, immunosuppresssion, etc, that increase likelihood of sepsis, and (c) the patient's vital signs,

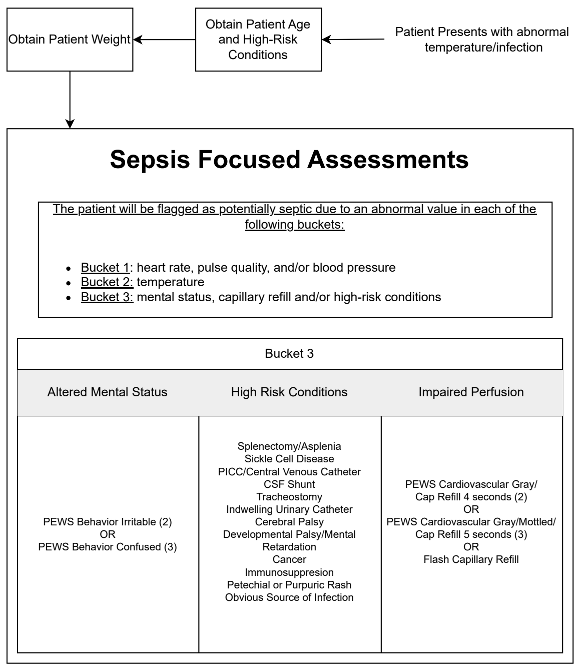
Obtain Patient Weight
Obtain Patient Age and High-Risk Conditions
Patient Presents with abnormal temperature/infection
Sepsis Focused Assessments
The patient will be flagged as potentially septic due to an abnormal value in each of the following buckets:
Bucket 1: heart rate, pulse quality, and/or blood pressure
Bucket 2: temperature
Bucket 3: mental status, capillary refill and/or high-risk conditions
Bucket 3
Altered Mental Status
High Risk Conditions
Impaired Perfusion
PEWS Behavior Irritable (2) OR PEWS Behavior Confused (3)
Splenectomy/Asplenia
Sickle Cell Disease
PICC/Central Venous Catheter
CSF Shunt
Tracheostomy
Indwelling Urinary Catheter
Cerebral Palsy
Developmental Palsy/Mental Retardation
Cancer
Immunosuppresion
Petechial or Purpuric Rash
Obvious Source of Infection
PEWS Cardiovascular Gray/ Cap Refill 4 seconds (2) OR PEWS Cardiovascular Gray/Mottled/ Cap Refill 5 seconds (3) OR Flash Capillary Refill

Fig. 1: 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
...	...	...	...
$\geq 13y$	> 100	< 90	< 36 or > 38.5

TABLE I: Vital Signs Chart

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 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.

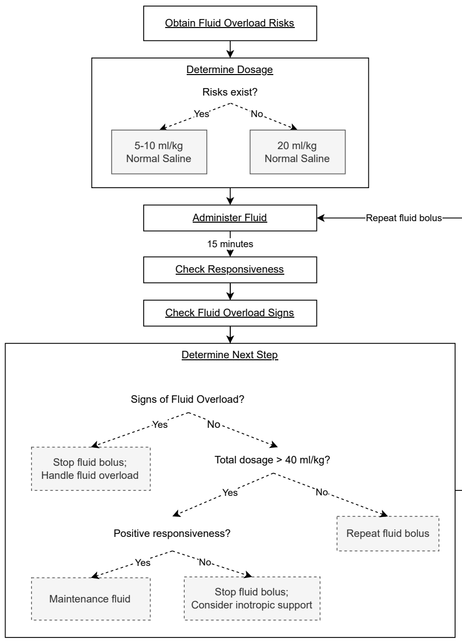

Fig. 2: Fluid Resuscitation Guideline

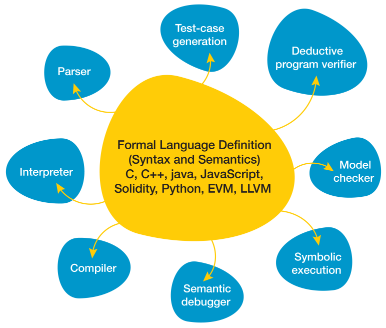
Test-case generation
Deductive program verifier
Parser
Formal Language Definition (Syntax and Semantics) C, C++, java, JavaScript, Solidity, Python, EVM, LLVM
Interpreter
Model checker
Compiler
Semantic debugger
Symbolic execution

Fig. 3: $\mathbb{K}$ Overview

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 Medi$\mathbb{K}$. We argue that this philosophy makes Medi$\mathbb{K}$ both intuitive to HCPs, and suitable for expressing complex guidelines.

III. Medi$\mathbb{K}$

In this section, we introduce the Medi$\mathbb{K}$ DSL for expressing CIGs. Medi$\mathbb{K}$ 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 Medi$\mathbb{K}$ (i.e., its syntax and semantics) in $\mathbb{K}$. $\mathbb{K}$ 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 $\mathbb{K}$ by need while discussing

Medi𝕂. For more details on 𝕂, we refer the reader to [40] [41].

The 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 Medi𝕂 in 𝕂, we obtain aforementioned tools for it without any extra effort. Additionally:

- The 𝕂-based interpreter for Medi𝕂 essentially executes the language's semantics rules, it is correct-by-construction.
- Incorporating changes to Medi𝕂 only requires updating the semantics. Since the tools are derived from the semantics, they're automatically updated.

The remainder of this section introduces Medi𝕂 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 Medi𝕂 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 Medi𝕂, 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 Medi𝕂. 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 Medi𝕂 using its 𝕂-framework definition. The 𝕂 definition of a language has two components. The first is the language's syntax, which is defined using a BNF-like notation. 𝕂 utilizes this grammar to generate a parser for programs in the language. We describe Medi𝕂's syntax in depth in Section III-A. The second is the semantics, which is defined using a 𝕂-configuration and rewrite rules. The 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 Medi𝕂 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.

```
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 }
```

A Medi𝕂 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 𝕂-grammar for statements. In 𝕂, productions are defined using the keyword `syntax` (lines 1, 7). Terminals are enclosed in quotes (`""`), and non-terminals begin with an uppercase character.

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

Lines 1-5 define the syntax of Medi𝕂 expressions. Line 1 defines basic expressions such as identifiers (denoted by the

[1]The complete executable semantics is available at [43].

builtin 𝕂 production `Id`), values such as booleans, or rationals, or "`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 `in` (line 5) are useful in context of defining BPGs, and are described through an example in section IV. Apart from these, Medi𝕂 also supports common expressions such as `+, -. >, >=` over rationals and `&&, ||` over booleans.

In lines 7-15, we define syntax for Medi𝕂 statements. Some of these, such as variable assignment (line 7), `if-else` (line 8) and `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 Medi𝕂's semantics in section III-B.

B. Semantics

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

1) 𝕂-Configuration: 𝕂 represents program execution state using 𝕂-configurations. A 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 𝕂 assume that the rest of the configuration remains unchanged. The following configuration defines the initial state for any Medi𝕂 program:

```
1  configuration
2    <instance multiplicity="*" type="Map"> ...
3      <k> createMachineDefs($PGM)
4       ~> createInitInstances </k>
5      <genv> .Map </genv>
6      <env> .Map </env>
7      <inBuffer> .List </inBuffer>
8      <activeState> . </activeState>
9    </instance>
10   <machine multiplicity="*" type="Map"> ...
11     <machineName> . </machineName>
12     <states>
13       <state multiplicity="*" type="Map">
14         <stateName> . </stateName>
15         <entryBlock> . </entryBlock>
16         <eventHandlers> ... </eventHandlers>
17       </state>
18     </states>
19   </machine>
```

The keyword `configuration` (line 1) defines a 𝕂-configuration, followed by xml-like notation for the 𝕂-cells. For example `<foo> ... </foo>` corresponds to a 𝕂-cell with the name `foo`. The `<instance>` cell (lines 2-9) contains state of each Medi𝕂 machine instance during execution. Each instance manages its instance variables using a map in the `<genv>` cell (line 5), a buffer of incoming events in the `<inBuffer>` cell (line 7) and the currently executing code in the `<k>` cell (lines 3-4).[2] When a Medi𝕂 program is executed, 𝕂 replaces `$PGM` (line 3) with the Abstract Syntax Tree (AST) of the program, obtained by parsing the program using the syntax from section III-A. The `createMachineDefs` constructs is defined (using rewrite rules) to traverse the program AST and populate the configuration with information related to each machine. The `createInitInstances` creates an instance for the machine with the `init` keyword, leading to execution of the initial machine's entry block. Note that `~>` symbol (line 3) is interpreted by 𝕂 as "followed-by", i.e., execution of `createMachineDefs` is followed by execution of `createInitInstances`. The attribute `multiplicity="*"` on lines 2 and 10 signifies that multiple copies of the corresponding cells, in this case `<machine>` and `<instance>` cells, can exist in the configuration during execution. This allows, during execution, for multiple machine definitions, each with multiple instances, to exist. The `<machine>` cell (lines 10-19) holds information relevant to a machine definition, such as the name in the `<machineName>` cell (line 11) and states in the `<states>` cell (lines 12-18). The `<state>` (lines 13-17) holds information relevant to a state, such as the entry block in cell `<entryBlock>` (line 15) and event handlers in cell `<eventHandlers>` (line 16).

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

```
rule <k> I:Id = V:Val => V ... </k>
     <env> (I |-> Loc) ... </env>
     <store> Store => Store[Loc <- V] </store>
```

Here, `I`, `V`, `Loc`, and `Store` are 𝕂-variables. Note the distinction between program variables and 𝕂-variables: while program variables are simply identifiers, 𝕂-variables have logical meaning. The `...` is used to denote parts of the configuration not relevant to the rule. Typically, the top of the `k` cell contains the statement currently being executed. Suppose we're executing the statment `i = 2;`. In this case, the current configuration will have a `k` cell of the form `<k> i = 2 ... </k>`, an environment cell `env` where variable `i` maps to some pointer p, and a store cell `store` containing a map M with some value pointed-to by p. The *LHS* matches with substitution $\theta = (I \mapsto i, V \mapsto 2, Loc \mapsto p, Store \mapsto M)$, resulting in the top of the `k` cell to be rewritten to the value 2, and pointer p updated to point to 2 in 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.

[2]For brevity, we present a simplified version of the configuration. See [43] for the entire configuration.

In the following sections we present several Medi$\mathbb{K}$ constructs relevant to defining BPGs using their $\mathbb{K}$-rules. We first present the rule for sending and receiving messages.

```
1  rule
2  <instance>
3   <k> send instance(RecvId) , EventName:Id , ( Args )
4       =>  done ... </k> ...
5  </instance>
6  <instance>
7   <id> RecvId </id> ....
8   <inBuffer> ... (.List
9    => ListItem(
10        eventArgsPair(EventName | Args | Epoch + 1
11        )))
12  </inBuffer> ...
13 </instance>
14 <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). (iv) The *epoch* decides when the machine can run, and is discussed in Section III-B2a.

To handle interaction with heterogeneous external sources, Medi$\mathbb{K}$ 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 Medi$\mathbb{K}$.

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

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 Medi$\mathbb{K}$ construct `createFromInterface`, which takes as arguments (a) the inteface name (lines 4), (b) a unique identifier string used to identify the instance outside the Medi$\mathbb{K}$ process. All other Medi$\mathbb{K}$ machines can interact with external sensor using variable `hrSensor`. There is no need to make any distinction between external, and Medi$\mathbb{K}$-based machines. To deal with external interactions, input and output pipes are provided to the Medi$\mathbb{K}$ process at launch. When the `send` construct is used on an external machine, Medi$\mathbb{K}$ 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 Medi$\mathbb{K}$, a JSON message in a pre-specified format needs to be written to the *write-end* of the input pipe.

Next, we desribe the rule for supporting tables in Medi$\mathbb{K}$. 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 Medi$\mathbb{K}$, 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

```
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 }
```

Fig. 4: Checking abnormality using tables

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 Medi$\mathbb{K}$, the following rules are responsible for assigning semantics to the `in-interval` construct:

```
1 rule E in interval(L, U) => (E >= L) && (E < U)
2   [macro]
3 rule E in { interval(L, U): S:Stmt Cs:CaseDecl }
4   => if (E in interval(L, U)) {S} else {E in { Cs }}
5   [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 desugar 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 desugar the remaining case statements.

a) Medi$\mathbb{K}$ Scheduling Semantics: Since the $\mathbb{K}$-generated interpreter is single-threaded, Medi$\mathbb{K}$ 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 Medi$\mathbb{K}$ 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 $P = \{\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_{n-1}\}$ be a program with n machines, where $\mathcal{M}_0$ is prefixed with `init`. Medi$\mathbb{K}$ allows instances of a machine to be created dynamically at runtime. For machine $\mathcal{M}_i \in P$, let $\mathcal{I}_{\mathcal{M}_i,j-1}$ be its j-th instance.

```
1  $epoch \leftarrow 0$
2  $scheduled \leftarrow \{\mathcal{I}^{0}_{\mathcal{M}_0,0}\}$
3  while $scheduled \neq \emptyset$ do
4      $\mathcal{I}^{\tau}_{\mathcal{M}_i,j} \leftarrow choose\,(scheduled)$ s.t.
               $\tau \leq epoch \wedge enabled(\mathcal{I}^{\tau}_{\mathcal{M}_i,j})$
5      $scheduled \leftarrow scheduled \setminus \mathcal{I}^{\tau}_{\mathcal{M}_i,j}$
6      $execute(\mathcal{I}^{\tau}_{\mathcal{M}_i,j}, scheduled)$
7      if $\nexists i', j', \tau'$ s.t. $(\mathcal{I}^{\tau'}_{\mathcal{M}_{i'},j'} \in scheduled)$
               $\wedge(\tau' \leq epoch) \wedge (enabled(\mathcal{I}^{\tau'}_{\mathcal{M}_{i'},j'}))$ then
8          $epoch \leftarrow epoch + 1$
9      end if
10 end while
```

Fig. 5: Medi𝕂 Scheduling Semantics

Execution begins in *epoch* zero with the implicit (first) instance of the initial machine, denoted by $\mathcal{I}^{0}_{\mathcal{M}_0,0}$. We use $\mathcal{I}^{\tau}_{\mathcal{M}_i,j-1}$ to say that the j-th instance of machine $\mathcal{M}_i$ is scheduled for execution in *epoch* τ. 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* $\mathcal{I}_{\mathcal{M}_0,0}$ becomes scheduled (line 2) at *epoch* 0. On line 4, an instance $\mathcal{I}^{\tau}_{\mathcal{M}_i,j}$ is non-deterministically chosen from all machines that are both scheduled to run when $\tau \leq$ *epoch* and *enabled*. We use $execute(\mathcal{I}^{\tau}_{\mathcal{M}_i,j}, scheduled)$ 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 $\mathcal{I}_{\mathcal{M}_i,j}$ sends an event to another instance $\mathcal{I}_{\mathcal{M}_{i'},j'}$, then the event is scheduled to be handled by $\mathcal{I}_{\mathcal{M}_{i'},j'}$ in or after the next epoch, i.e., $scheduled \leftarrow scheduled \cup \{\mathcal{I}^{epoch+1}_{\mathcal{M}_{i'},j'}\}$. Similary, if a `goto` statement is encoutered, 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 builtin support for timers. Therefore, in Medi𝕂, `sleep(duration)` is described by the following rule:

```
1  rule <k> sleep(Duration:Int) ;
2        =>    jsonWrite( { "action"   : "sleep"
3                         , "duration" : Duration
4                         , "tid"      : TId }
5                         , ... )
6          ~> releaseExecutor
7          ~> waitForSleepResponse(TId) ...
8       </k>
9       <tidCount> TId => TId +Int 1 </tidCount>
```

`sleep` results in a JSON message being sent to a remote endpoint (lines 1-5) specified when the Medi𝕂 process is launched. This mimics sending an event to an external *timer* machine, with the desired duration as the payload. At the remote endpoint, code must be provided (in any programming language) to parse the message, and respond with a JSON message indicating the expiration of the timer once the desired duration has passed. A unique transaction-id (lines 4, 7, 9), which the code at the endpoint is expected to provide in the response, uniquely identifies the machine instance being responded to. `sleep` causes a context-switch to occur on line 6, releasing the executor lock to process other *scheduled* machines.

When a message signifying the expiration of the timer is sent to the Medi𝕂 process, along with the transaction id of source instance, the corresponding event signalling the completion of the sleep statement is placed at the *beginning* of the source machine instance's input buffer, and the instance is scheduled to resume execution in the next epoch. The following rule handles the external response:

```
1  rule
2  <k> waitForSleepResponse(TId) => . ... </k>
3  <inBuffer>
4    (ListItem(event($SleepDone | TId | Tau ))
5     => .List)  ...
6  </inBuffer>
7  <executorAvailable>
8    true => false
9  </executorAvailable>
10 <epoch> Epoch </epoch>
11   requires Tau <=Int Epoch
```

The `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].

[3]the entire CDSS for sepsis management is available at [45].

```
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. 6: Sepsis Screening in Medi𝕂

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, in Fig. 1, 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. 6. 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. 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

```
1  machine FluidTherapy
2    receives StartFluidTherapy, ... {
3
4
5    init state Start {
6      on StartFluidTherapy do {
7        goto ObtainRisks;
8      }
9    }
10
11   state ObtainRisks {
12     // Obtain fluid overload related risks
13   }
14
15   state SuggestFluidDosage {
16     // Suggest a dosage based on risks
17   }
18
19   state WaitForAdministerFluidConfirmation {
20     // Handler for Normal Saline Administration
21     on ConfirmNormalSalineAdministered do {
22       sleep(900);
23       goto EvaluateResponsiveness;
24     }
25   }
26
27   state EvaluateResponsiveness {
28     entry {
29       send tablet
30         , Instruct
31         , ("get responsiveness to fluids");
32     }
33
34     on FluidResponsivenessEntered(responsiveness) do {
35       isResponsiveToFluids = responsiveness;
36       goto ObtainFluidOverloadSigns;
37     }
38   }
39   state ObtainFluidOverloadSigns {
40     // Obtain signs of fluid overload
41   }
42
43   state AskNextStep {
44     entry {
45       var recommendation;
46       if (this.fluidOverload) {
47         recommendation = "handle fluid overload";
48       } else {
49         // obtain total saline dose
50         if ((totalSalineDose >
51               measurementBounds.salineDosageUpperBound) {
52           if (isResponseiveToFluids) {
53             recommendation = "maintainence fluids"
54           } else {
55             recommendation = "consider inotropic support";
56             broadcast ConsiderInotropicSupport;
57           }
58         } else {
59           recommendation = "repeat fluid bolus";
60         }
61       }
62       // Send recommendation to tablet
63       // Wait for HCP response
64     }
65   }
66 }
```

Fig. 7: Fluid Resuscitation in Medi𝕂

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:

```
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:

```
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 syntatic 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.

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

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 Medi$\mathbb{K}$ 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 Medi$\mathbb{K}$, the first step towards building safe CDSSs. While Medi$\mathbb{K}$ 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 $\mathbb{K}$, (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 Medi$\mathbb{K}$ - a new DSL for expressing BPGs that uses a semantics-first approach to build CDSS. Medi$\mathbb{K}$ programs consist of *concurrently* executing *instances* of *State Machine*. Medi$\mathbb{K}$ 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, Medi$\mathbb{K}$ 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 Medi$\mathbb{K}$, 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 Medi$\mathbb{K}$ DSL.

References

[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] R. T. Sutton, D. Pincock, D. C. Baumgart, D. C. Sadowski, R. N. Fedorak, and K. I. Kroeker, "An overview of clinical decision support systems: benefits, risks, and strategies for success," *npj Digital Medicine*, vol. 3, no. 1, p. 17, 2020.

[23] M. Peleg, "Computer-interpretable clinical guidelines: A methodological review," *Journal of Biomedical Informatics (JBI)*, vol. 46, no. 4, pp. 744–763, 2013.

[24] P. A. De Clercq, J. A. Blom, H. H. Korsten, and A. Hasman, "Approaches for creating computer-interpretable guidelines that facilitate decision support," *Artificial Intelligence in Medicine (AIM)*, vol. 31, no. 1, pp. 1–27, 2004.

[25] P. Kubben, M. Dumontier, and A. Dekker, *Fundamentals of clinical data science*. Springer, 2019.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of c," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 336–345.

[37] D. Bogdănaş and G. Roşu, "K-Java: A complete semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, January 2015, pp. 445–456.

[38] D. Park, A. Ştefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 346–356.

[39] 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.

[40] T. F. Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu, "The K primer (version 3.3)," *Electronic Notes in Theoretical Computer Science*, vol. 304, pp. 57–80, 2014, proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[41] G. Roșu and T. F. Șerbănută, "An overview of the k semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[42] 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

[43] "The K semantics of MediK." [Online]. Available: https://github.com/fmcad-78/medik-semantics.git

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

[45] "Sepsis screening and management application." [Online]. Available: https://github.com/fmcad-78/psepsis.git

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

[47] 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.

[48] 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.

[49] 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.