A SEMANTICS-FIRST APPROACH TO SAFE CLINICAL DECISION SUPPORT

BY

MANASVI SAXENA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Grigore Roşu, Chair & Director of Research
Professor Lui Sha
TBD

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.

Table of contents

List of Abbreviations

AAP	American Academic of Pediatrics
AAP	American Academy of Pediatrics
ACLS	Advanced Cardiovascular Life Support
AED	Automatic Emergency Defibrillator
AHA	American Heart Association
ALS	Advanced Life Support
BLS	Basic Life Support
BNF	Backus-Naur Form
BPG	Best Practice Guideline
CAST	Commercial Aviation Safety Team
CDSS	Clinical Decision Support System
CPR	Cardiopulmonary Resuscitation
CSM	Communicating State Machines
CTL	Computation Tree Logic
DeGeL	Digital Electronic Guideline Library
EHR	Electronic Health Record
EVM	Ethereum Virtual Machine
GEE	Guideline Execution Engine
GEODE-CM	Guided Entry of Data Elements for Clinical Management
GLARE	Guideline Acquisition, Representation, and Execution
GLIF	Guideline Interchange Format
GOSpeL	Guideline prOcess Specification Language
GPROVE	Guideline PRocess cOnformance VErification Framework
GP	General Practitioner
HCP	Healthcare Practitioner

ITL	Interval Temporal Logic
LTL	Linear Temporal Logic
MBA	Model-based Architecture
MLM	Medical Logic Module
NAM	National Academy of Medicine
NHS	National Health Service
ONC	Office of the National Coordinator for Health Information Technology
P-CAPE	Partners Computerized Algorithm Processor and Editor
PCAST	President's Council of Advisors on Science and Technology
PME	Preventable Medical Error
ROSC	Return of Spontaneous Circulation
SAGE	Standards-based Guidelines Environment
SOS	Structural Operational Semantics

Chapter 1

Introduction

Preventable medical errors (PMEs) characterized by misdiagnosis or mistreatment present a significant challenge in healthcare, both in the United States, and abroad [1]. According to a seminal report on the subject by the Institute of Medicine, in 1997, between 44,000 and 98,000 deaths were estimated to have been caused by PMEs in the United States alone [2]. 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 prevalence of PMEs in healthcare now finds increasing recognition. In September 2023, the President's Council of Advisors on Science and Technology (PCAST) published a report stating patient safety to be an "urgent national public health issue", and calling for a "transformational effort" to address the problem [4]. Approximately a quarter of medicare patients, the report mentioned, experienced adverse events during their hospitalization, with many catastrophic outcomes. Worryingly, 40% of these adverse events were due to PMEs. Thus, while recognizing the commitment that healthcare practitioners and their organizations towards ensuring quality care, the report deemed rates of medical errors and injuries as "alarmingly high".

The adverse effects of PMEs extend beyond patient outcomes. PMEs have a negative effect on healthcare providers (HCPs). According to the authors of [1], PMEs caused psychological effects such as anger and guilt in healthcare providers (HCPs), adversely impacting their mental health. Moreover, there are costs associated with such errors. In 2008, the financial burden of PMEs to the United States was estimated to be $19.5 billion, of which $17 billion could be directly attributed to additional healthcare expenditure such as ancillary and prescription drug services, and inpatient and outpatient care. Additionally, $1.4 billion was attributed to increased mortality and loss of productivity from missed work due to short-term disability claims [5].

In the United States, several initiatives to improve patient safety by reducing preventable errors have been undertaken at the federal government level. The aforementioned September 2023 PCAST report itself included several suggestions to address patient safety, such as:

(i) ensuring patients receive evidence-based practice for preventing harm and assessing risk, where as many measures as possible are generated in real-time from electronic health data.

(ii) incentivizing hospitals to utilize evidence-based practices and associated novel computing tools.

(iii) promoting research and development in areas such as computer science and health technology, including tools in prototype form, or limited deployment that hold promise to guide decision making to improve quality and safety.

(iv) empowering hospitals with safety-critical methods, transparency, and a instilling a culture of safety.

(v) harnessing advances in technology, such as in Artificial Intelligence, while employing best practices and engineering principles to mitigate safety and reliability of such systems.

The report highlights that technology is expected to play a major role in addressing the problem. The emphasis on technology to reduce human errors isn't unprecedented. The report recommends the establishment of a multidisciplinary National Patient Safety Team, along the lines of the Commercial Aviation Safety Team (CAST) [6], comprising of experts from clinical, safety and computer sciences, to address challenges in patient safety. In the decade following its inception in 1997, CAST was responsible for reducing the fatality risk for commercial aviation in the United States by 83%, and aims to reduce the risk further by 50% by 2025. To put CAST's achievements into perspective, between 2008 and 2020, U.S. carriers transported more than 7.8 billion passengers. But, in the same time-frame, there were only 2 fatalities [7]. Several solutions were instrumental in achieving such remarkable results, including:

- fly-by-wire control that optimizes pilot inputs for safety, reliability, and passenger comfort [8].
- flight management and autopilot systems that automate certain navigation and flying tasks to reduce pilot workloads and make bandwidth available for monitoring and managing emergencies [9].
- diagnostic and assurance systems that support enhanced pilots' and maintenance staff's understanding of aircraft systems, faults, and mitigation strategies [9].
- traffic management systems that enhance situational awareness of pilots, air traffic controllers, and flight planners [10], [11].

CAST was instrumental in enabling the development of, and enacting policy to encourage adoption of aforementioned solutions. While solutions to address safety issues in aviation were often met with hurdles and skepticism, sustained efforts that involved stakeholders such airline pilots, government agencies and safety-centric aviation companies eventually led to widespread adoption and deployment [12].

Unfortunately, despite similarities, improvements in aviation safety haven't been adapted in healthcare [13]. But, as evidenced by the September 2023 PCAST report, initiatives are been undertaken to address this. Addressing patient safety is a complex problem requiring multifaceted solutions, of which comprehensive use of technology is a part. This work in particular focuses on improving patient by enabling the development of *safe* and *cost effective* guidelines-based clinical decision support systems (CDSSs). These systems assist healthcare providers (HCPs) follow evidence-based best practice guidelines (BPGs) – evidence-based statements published by medical institutions that codify recommended treatment for various clinical scenarios [14]. 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 [15].

Evidence suggests that BPGs can reduce medical errors, but, their effectiveness hinges on healthcare providers adhering to them in practice. 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) [16], [17]. 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 [18]–[22]. This can partly be attributed to the fact that BPG-adherence is difficult to achieve in practice [23], [24]. However, integrating BPGs with existing patient care-flow, and making them readily-accessible when required can improve adherence [25]. To this end, CDSSs codify BPGs and support HCPs with situation-specific

advice. Such systems have been shown to improve BPG-adherence [26], [27], and evidence from multi-center clinical trials suggests that they reduce PMEs [28], [29]. Thus, guideline-based CDSSs are now considered imperative to the future of medical decision making in general [30].

Typically, a CDSS usually consists of the following components [31]:

(a) a translation of the guideline to an executable medium, called the knowledge-base.

(b) an user interface (UI) that healthcare providers use to interact with the system.

(c) additional infrastructure that integrates with external data sources such as sensors, health records.

The system utilizes available data to assess patient state, and supports the HCP by providing guidelines-prescribed advice for treating assessed patient condition. As is evident, CDSSs align well with recommendations for improving patient safety put forth by the September 2023 PCAST report. They enable the use of best practice that codify evidence-based advice to optimize treatment outcomes and typically utilize patient data available via patient sensors and electronic health records to ascertain patient state. Well implemented CDSSs can aid decision-making and improve overall quality and safety at healthcare facilities that use them.

The PCAST report also stresses on adapting lessons from aviation in healthcare. In aviation, systems for automation, monitoring and error mitigation that reduce worklkoads and enhance situational awareness of pilots have improved safety as pilots became accustomed to using and relying on such systems through training and operating procedures. CDSSs can play a similar role in healthcare, by ensuring HCPs, who often work in stressful environments, have the relevant information available to them at appropriate times to reduce workloads and improve situation awareness. CDSSs present a promising approach to addressing issues of patient safety in healthcare, and are thus the focal point of this work.

1.1 Hurdles to Wider CDSS Adoption

The Office of the National Coordinator for Health Information Technology (ONC) [32] has also recognized the potential that CDSS have to improve safety [33]. As part of its efforts to enable wider CDSS adoption, the ONC organized meetings that involved key stakeholders such as CDSS experts, policy makers and Electronic Health Record (EHR) vendors to study the state of CDSS in healthcare and barriers limiting adoption. Their findings were summarized in a National Association Medicine (NAM) special report [34]. Drawing from several publications on CDSSs, including a literature review commissioned by the Agency for Healthcare Research Quality [35], the report highlighted that despite significant evidence that demonstrates CDSSs reduce both errors and cost while improve quality of care, implementation and actualization of such systems remains "nascent" due to the following challenges (Cs):

C1. Absence of systematic ways of *validating content* in a *reliable*, *accessible* and *updateable* manner.

C2. Lack of *reliable*, *shareable* CDSS content that can be easily adopted across healthcare organizations and their (Information Technology) IT systems.

C3. Technical difficulties of sharing due to *need for adaptation* to diverse Electronic Health Records (EHR) systems.

C4. *Suboptimal* User Interfaces (UIs), implementation choices and workflows.

1.2 Solution

Typically, CDSS are implemented as cumbersome standalone systems. To develop a CDSS, practitioners work with developers to come up with a requirements documentation that presents the BPGs's medical knowledge in a manner amenable to software development [36]. This documentation is then utilized to develop the CDSS. Thus, the BPG serves as a functional specification for the medical knowledge encoded in the CDSS. But, the aforementioned process has several limitations. First, the implementation, i.e., the medical knowledge within the CDSS, may not concur with its specification, i.e., the textual BPG. BPGs are long, complex textual documents, where the exact meaning of terms may not be explicitly stated, and recommendations may be ambiguous [37]. Capturing and communicating these complexities via requirements documentation is challenging, and incorrect or incomplete documentation has resulted in failed implementations [38]. Second, as BPGs evolve to reflect new evidence or adaptions, corresponding updates must be made to the CDSS as well. Thus, effort (and associated cost) must expended to ensure that the medical knowledge is accurately updated in the CDSS as well.

To address these limitations, instead of a monolithic system, our approach decomposes the CDSSs into:

(a) the medical knowledge encoded in the CDSS, which we refer to as the knowledge-base.

(b) an interface for user-interaction.

(c) additional infrastructure that integrates with external data sources such as sensors, health records.

To ensure that the knowledge-base accurately reflects medical knowledge, domain specific languages (DSL) that allow medical guidelines to be described in a manner that is both *computer interpretable* and *comprehensible* to healthcare practitioners are often utilized. A computer interpretable guideline (CIG) can be directly used as knowledge-base in a CDSS. But, by being comprehensible to HCPs, such a guideline can also replace its textual counterpart. Thus, such DSLs enable production of medical guidelines that can serve as both the system's functional specification, i.e. the BPG, and implementation, i.e. the knowledge-base. This ensures that there is no gap between the BPG and the CDSS' knowledge-base.

Given the safety-critical nature of CDSSs, the need for CIG DSLs to have a complete formal semantics that can be used to implement tools such as execution engines, model checkers, etc. has been recognized. It's also desirable for such languages to have execution engines with formal *correctness guarantees*, and accompanying tools for formal analysis. To this end, some existing DSLs have partially-defined semantics and support for verification via model-checking. However, as identified by the authors of [39], [40], existing languages lack a complete formal semantics, execution engines with correctness guarantees and a comprehensive suite of accompanying tools such, 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.

To address limitations of existing approaches, we developed a new language for computer interpretable guidelines called Medi$\mathbb{K}$. Medi$\mathbb{K}$ follows the *semantics-first* philosophy. As shown in **??**, this philosophy dictates that instead of implementing tools such as an interpreter, compiler, program verifier for a language from scratch in an ad-hoc way, the execution semantics of the language should be formally defined, from which said tools are derived, making them *correct-by-construction*. This also enables Medi$\mathbb{K}$ to quickly and systematically incorporate healthcare practitioner feedback, as only the semantics need to be updated. As the tools are derived from the semantics, changes to the semantics are automatically reflected across all tools.

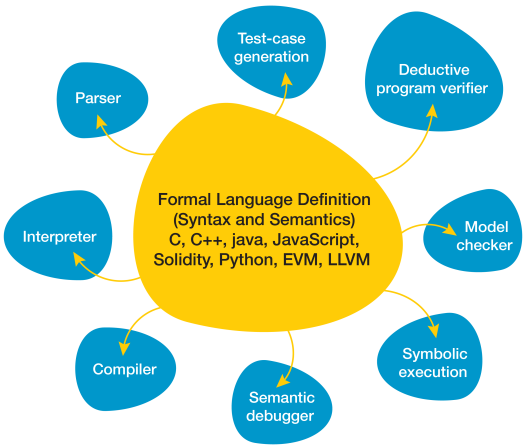


Figure 1.1: Semantics-first philosophy

MediK's semantics have been formalized in K – a rewriting-based framework for defining programming languages [41]. Our choice for using K was motivated by (a) experience and expertise in developing K, and defining languages in it, and, (b) the fact that K has been used to define other real-world languages such as C [42], Java [43] and the Ethereum Virtual Machine [44], and verify programs in them [45], [46].

CDSSs are complex systems. The complexity stems from:

(i) *concurrency* to account for multiple diagnoses and treatments occur simultaneously, and,

(ii) interaction with heterogeneous *external agents* such as sensor, health records, and practitioners.

To support formal analysis of such systems, a DSL for computer interpretable guidelines must take into account said *complexity* while ensuring *comprehensibility* to healthcare practitioners.

To ensure *comprehensibility*, we ensure MediK-based guidelines *resemble* their existing textual counterparts that practitioners are already *familiar* with. We started by examining existing BPGs and the state-of-art DSLs for expressing them in a *computer-interpretable* way, and observed that BPGs are often expressed in informal flowchart-like notation. For instance, see [47] and [48] for flowchart-based BPGs for management of *cardiac arrest*, and screening, risk-reduction, treatment and survivorship in cancer care. To tackle *complexity*, we utilized concepts from existing state-of-art in modeling and verification of large concurrent systems like P [49] and maude [50]. In MediK, like in P, a program is a collection of concurrently executing state machines that interact by passing messages. Just as in object-oriented maude, MediK treats external agents, such as the UI, sensors, etc., as state machines. Since BPGs are often expressed using flowcharts, we represent each flowchart in MediK as a state machine. For formal analysis, we need to model heterogeneous external components such as the UI, sensors, etc. We model such agents as *ghost machines*. These machines are only used for formal analysis, but are discarded during execution. During execution, the interface that glues all components is responsible for forwarding messages to and from external agents to MediK. This allows MediK to support a uniform way of modeling heterogeneous external agents for both execution and analysis.

In collaboration with the Children's Hospital of Illinois at OSF St. Francis Medical Center (referred to as OSF in the remainder of this work), we used our MediK-based approach to implement a CDSS for management pediatric sepsis based on OSF's guidelines. Sepsis is life-threatening condition caused by the body's extreme response to an infection [51], and is a major cause of morbidity and mortality in children [52]. Adverse outcomes can, however, be mitigated through timely identification and prompt treatment with

antibiotics and intravenous (IV) fluids [53], [54]. BPGs for screening and management of sepsis in pediatric Emergency Departments (EDs) have shown effectiveness in screening and management of sepsis [52], leading to their adoption in many pediatric EDs [55], [56].

The MediK-based CDSS is a complex system, involving multiple concurrent workflows that interact with each other. As outlined earlier, the CDSS consists of three independent components:

- A UI tailored to the preferences of practitioners at OSF.
- A MediK based computer interpretable guideline based on OSF's existing textual BPG.
- Additional infrastructure that glues aforementioned components together.

The knowledge-base of the CDSS is implemented as a MediK program. This computer interpretable guideline in MediK expresses medical knowledge *succinctly*, and allows establishing desired *safety* properties. For the sepsis management CDSS, we used MediK's semantics-based model checker to establish *responsiveness*. We say the system is *responsive* if, under certain reasonable assumptions about the UI and interactions with external agents such as sensors, health records, all possible inputs to the system lead to a response. This ensures that the system does not "freeze" or "hang" when being used. But, beyond *responsiveness*, we need to ensure that the MediK program captures medical knowledge *accurately*. To this end, we try to ensure that MediK programs resemble existing flowcharts-based BPGs that practitioners are familiar with, enabling them to validate the semantics of the MediK-based computer-interpretable guideline. Once verified and validated, the MediK program can be combined with a customizable UI to obtain a complete CDSS with safety guarantees. To the best of our knowledge, our is the first comprehensive CDSSs for sepsis management to provide any *safety guarantees*.

Recall from **??**, the four challenges from the NAM special report on CDSSs. Our approach's modular architecture and use of MediK for computer-interpretable guidelines tackles all challenges. MediK based guidelines provide a *systematic* way of *validating* medical knowledge that can be easily updated, thus addressing C1. Comprehensibility of MediK to practitioners ensures transparency. Practitioners can rely on the system to perform as expected, partly addressing C2. Our approach is agnostic to IT being utilized at the hospital. As mentioned earlier, our approach requires the implementation of a hospital-specific interface that glues the MediK-based computer interpretable guideline and the UI with the hospital's IT infrastructure, addressing C2, C3 and partly C4. Moreover, our approach allows the UI to be safely customized, as any changes remain contained to the UI itself, and safety-critical components such as the knowledge-base remain untouched. This addresses C4, as the UI can be optimized to a hospital's preferences. Moreover, as both the MediK-based guideline and UI are shareable, the cost for deploying a CDSS is lower than building a standalone, non-modular system from scratch.

Chapter 2

Background

?? conceptually introduced best practice guidelines (BPGs) and clinical decision support systems (CDSSs). This chapter introduces relevant background details on BPGs and CDSSs. In **??**, we utilize a real-world BPG to explain the motivation behind codifying treatment in the form of clinical guidelines. We also briefly discuss common characteristics of such guidelines that enable medical knowledge to be represented efficiently and accurately.

BPGs are usually published by hospitals, research institutions and medical associations with the aim to improve quality of care by (a) reducing medical errors due to preventable causes, (b) standardizing knowledge from latest evidence-based research, and, (c) enabling access to aforementioned knowledge at medical establishments that lack resources to conduct research.

While in theory, following BPGs should improve clinical outcomes, their effectiveness in practice is dictated by whether healthcare practitioners follow them or not. In **??**, we present challenges that practitioners encounter in following BPGs. We then argue that non-conformance results in worse patient outcomes. Next, we show how computerized systems that utilize data from available heterogeneous sources such as electronic health records and sensors for patient parameters can improve patient outcomes by addressing challenges to following BPGs encountered by practitioners.

2.1 Clinical Best Practice Guidelines

Clinical best practice guidelines are evidence-based statements published by hospital and medical associations that codify recommended interventions for various clinical scenarios [57]. It has been recognized that, if implemented correctly, BPGs have the potential to:

1. Reduce unwarranted variations in clinical practice.
2. Improve healthcare safety and quality.
3. Enhance translation of research into practice.
4. Reduce healthcare costs.
5. Enable development of performance measures for diseases [58], [59].

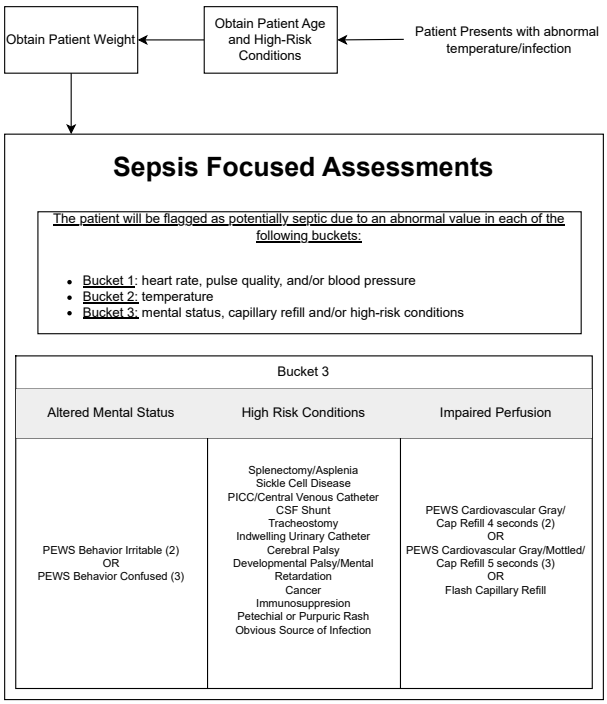


Figure 2.1: Pediatric sepsis screening BPG

Guidelines in the form of statements and recommendations were first introduced in the 1970s, and were mostly based on expert opinion In the 1980s and the early part of 1990, a significant increase in evidence from proliferation of randomized controlled trails occured. This co-incided with the introduction of computers at medical instutions that enabled available evidence to be quickly accessed to aid decision making [60], [61]. These developments facilitated a shift towards more rigorous development of BPGs, where evidence was prioritized over expert opinion [58].

To be able to administer optimum care, practitioners must take into account evidence from ongoing research to guide treatment. This can be particularly challenging, as available evidence can evolve rapidly. BPGs enable findings from latest research to be quickly translate in practice. Well designed BPGs are usually developed and published by trusted medical establishments (such as medical associations and research institutions) that base recommendations on best available evidence [58]. As the body of evidence increases over time through research, BPGs must be updated accordingly to reflect latest findings.

To illustrate characteristics of BPGs, we briefly go over a BPG for managing sepsis in pediatric cases used at OSF St. Francis Medical Center in Peoria, Illinois – a major pediatric hospital in the United States. Note that for brevity, we refer to said hospital simply as OSF in the remainder of this section. Sepsis is life-threatening condition caused by the body's extreme response to an infection [51], and is a major cause of morbidity and mortality in children [52]. Adverse outcomes can, however, be mitigated through timely identification and prompt treatment with antibiotics and intravenous (IV) fluids [53], [54]. BPGs for screening and management of sepsis in pediatric Emergency Departments (EDs) have shown effectiveness in screening and management of sepsis [52], leading to their adoption in many pediatric EDs [55], [56].

In **??**, we present a simplified version of the screening section of OSF's sepsis mangement 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,

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 2.1: Vital Signs Chart

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 **??** that contains 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 **??**, 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.

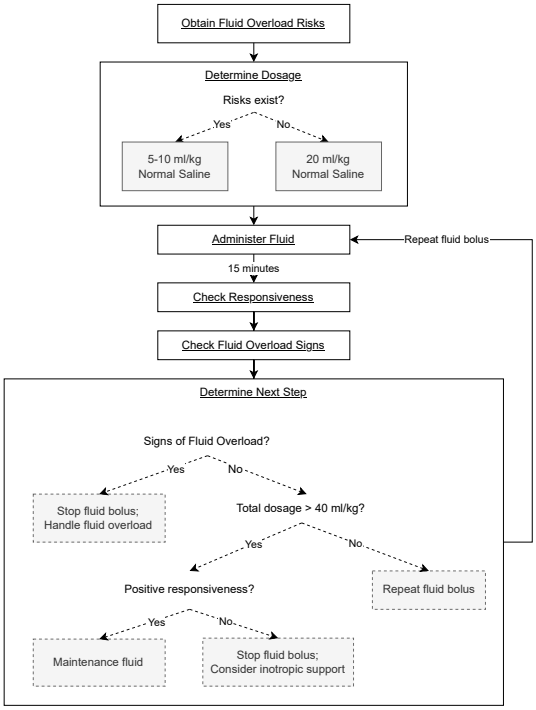


Figure 2.2: Fluid Resuscitation Guideline

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 antibitiotics.
- Often specified in a *flowchart-like* notation. See [47] and [48] for other flowchart-based BPGs for management of *cardiac arrest*, and screening, risk-reduction, treatment and survivorship in cancer care respectively.
- Require communication between *heterogeneous agents* such as monitors and Electronic Health Records (EHRs).
- Often use *tables* indexed by parameters such as age, weight, etc to present normal/abnormal ranges for measurements, or recommended dosages for drugs.

2.2 Clinical Decision Support Systems

BPGs are developed with the intention of improving patient outcomes by reducing preventable medical errors, and their have resulted in them being widely adopted in daily practice [25]. However, BPG adoption in clinical settings has several challenges. BPGs are complex documents, which may also contain some vagueness. Exact meaning of terms is not always defined and recommended actions may not be clearly articulated. The long and cumbersome nature of BPGs may make them difficult to effectively apply in regular practice. Additionally, such BPGs need to be periodically updated to take into account latest evidence, and adapted according to local needs of the setting they're utilized in [62].

To mitigate aforementioned issues, systems that utilize available patient data to assess patient state and issue guideline-based decision support to practiioners can be utilized [62]. Such computerized clinical decision support systems (CDSSs) codify medical knowledge in BPGs and provide practitioners with situation-specific advice that "guides" them towards adherence to the underlying BPG. Well implemented CDSSs have been shown to improve BPG-adherence [26], [27], and reduce PMEs based on evidence from multi-center clinical trials [28], [29].

2.2.1 CDSS Example

In **??**, we presented a real-world BPG for screening and management of sepsis in pediatric cases. To illustrate how a CDSSs can enable better BPG adherence, we utilize an example CDSS that codifies the sepsis management BPG to administer decision support.

?? and **??** show an overview of a CDSS for sepsis management. The CDSS is intended to work as tablet at the patient's beside in a hospital's emergency department (ED). **??** depicts main screen that the practitioner sees on the tablet. The main screen consists of four panes (labeled and color coded for illustration purposes). Pane 1 shows a grid with the patient's vitals pulled from a combination of the patient's records, sensor measurements and assessments made by practitioner through prompts that appear on the screen. Red and green depict abnormality and normality respectively, while gray indicates missing measurements, which are assumed to be normal. As more measurements become available, corresponding boxes in the grid are updated to reflect their status. The two boxes in the top left corner depict whether the patient is suspected to have sepsis, and be in septic shock. The CDSS continually monitors these vitals to determine whether they satisfy the criteria for sepsis or septic shock specified in the hospitals guideline, where the guideline itself is a

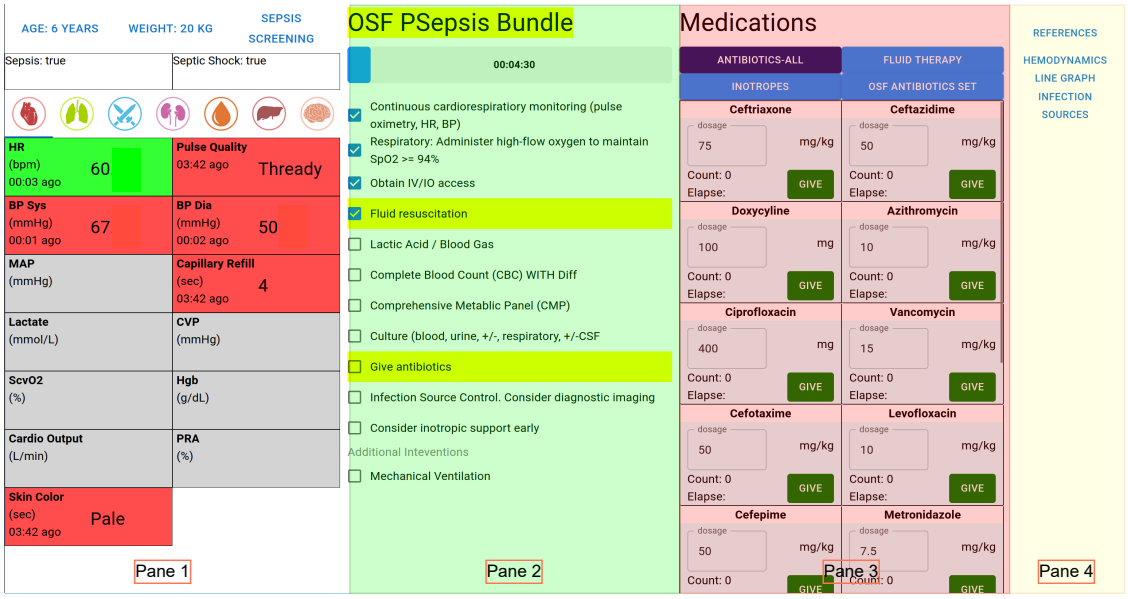


Figure 2.3: Sepsis Tool Main Screen

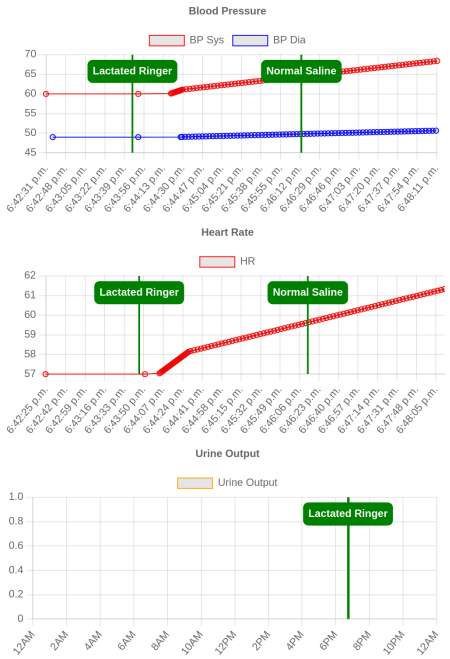
MediK program that the practitioners can view and comprehend. Pane 2 shows the tasks the practitioner is expected to perform once sepsis is detected. As these tasks are supposed to be performed within an hour (referred to as golden hour in sepsis treatment), the CDSS tracks time since the first task is started by clicking the corresponding check-mark. For some complex tasks, such as fluid resuscitation and antibiotics administration, an auxiliary window guides the practitioner through the task. The example scenario shows a patient with concerning vitals despite having given multiple fluid boluses, indicating septic shock – a condition that demands immediate attention. Thus, as specified in the guidelines, the system recommends urgently administering antibiotics alongside fluids, by highlighting the relevant tasks in yellow. Pane 3 contains medications grouped by type, where clicking the "GIVE" button corresponds to administering the drug. Pane 4 leads to toggleable windows with important information. For instance, clicking on "HAEMODYNAMICS LINE GRAPHS" toggles the window in **??** depicting the patient's vitals over time punctuated by points at which drugs were administered.

2.2.2 CDSS Components

CDSSs were first introduced in the 1970s. Initial systems didn't integrate well with existing patient care workflows, and didn't find greater adoption outside academia. However, wider adoption of electronic health records and digital systems for patient parameters enabled better integration of CDSSs with workflows in everyday practice, enabling wider adoption. Modern CDSS can deliver recommendations through a diverse spectrum of devices such as systems at the patient bedsides, desktops or tablets carried by practitioners and smartphones. This versatility has further enhanced CDSSs adoption [31].

In **??**, we provide an overview of the components of a typical CDSS, and the underlying architecture. A typical guidelines-based CDSS conceptually consists of the following components [31]:

Knowledge Base: The knowledge base is the encoding of the underlying guideline in a computable medium. Recall from **??** that BPGs are typically developed experts in medicine and published as textual documents.



(a) Patient Vitals vs Time

To develop a CDSS, experts in medicine collaborate with computer scientists or software developers to develop requirements documentations that present the BPG's semantics in a manner amenable to software development. This is subsequently utilized by software developers to encode knowledge from the textual BPG into some programming language [36]. For instance, the CDSS for sepsis management from **??** utilizes the textual BPG shown in **??** and **??** to diagnose and manage sepsis. Thus, medical knowledge in the textual BPG must translated into some programming language to enable its use in a functioning CDSS.

Clinical Data: CDSSs utilize clinical to assess patient state and generate recommendations as specified in the underlying BPG. Typically this involves utilizing available data from various sources such as electronic health records, devices that monitor various patient vitals, and inputs from HCPs.

Consider the sepsis screening BPG in **??**. In order to screen a patient for sepsis, data and measurements such as the patient's age, associated high-risk conditions, mental status, heart rate, etc. is required. The origins of the data is generally very diverse. Information such as the patient's age, weight and associated high risk conditions remain static over the duration of sepsis management, and can be obtained from the patient's health records. On the other hand, information such as the patient's mental state or the condition of the patient's skin necessitates an assessment from an HCP at the patient's bedside before it's manually entered into the system. Finally, for patient parameters such blood pressure or mean arterial pressure that change dynamically during the course of treatment, sensors or monitors that provide them in real-time should ideally be utilized. Typically, CDSS must take said diversity in data sources into account to be effective.

User Interface (UI): The UI is the interface the HCPs utilize to interact with the system. The UI typically:

- Delivers guideline-specified information (recommendations, alerts, etc.).
- Facilitates input of necessary data from the practitioner.

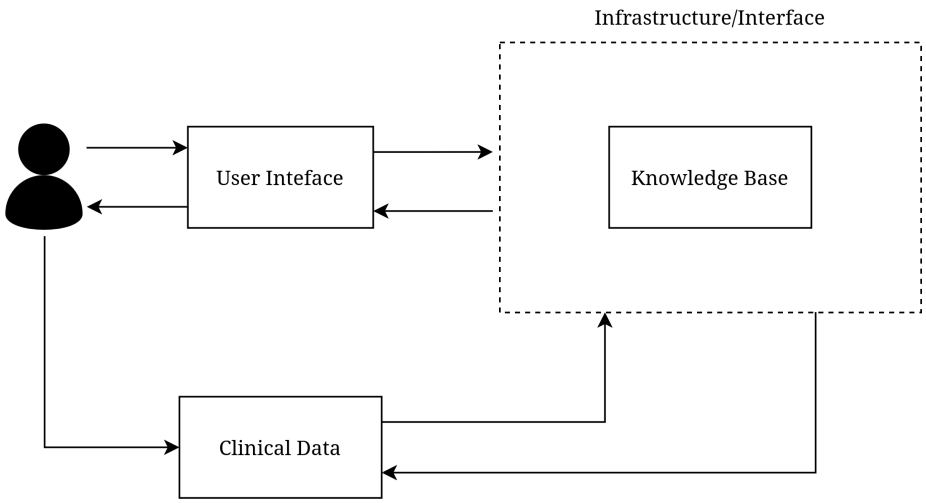


Figure 2.5: CDSS Components

- Displays relevant patient information (vital signs, treatment progress).

The UIs need to account for the diversity in devices used in modern healthcare. Interaction can occur through desktops, tablets, or even cell phones. The UI of the our running example CDSS for managing CDSS is meant to be displayed on tablet at the patient's bedside, and is shown in **??**. The displayed information is organized into distinct "panes": Pane 1 for patient state, Pane 2 for treatment progress, Pane 3 for medication details, and Pane 4 for textual guideline references. Time-sensitive and critical information appears as overlaid popups/prompts.

Additional Infrastructure Finally, additional infrastructure is required to glue all aforementioned components into a functioning CDSS. Said infrastructure typically consists of software to:

- Integrate with a healthcare establishment's existing Information Technology stack for access to electronic health records, and systems for services such as drug order and delivery, HCP communication, etc.
- Interface with various devices, including sensors and monitors, that track patient parameters like heart rate, blood pressure, etc.
- Control treatment-related devices such as infusion pumps, ventilators, etc.

For instance, the example CDSS for sepsis management shown in **??** integrates requires additional software for integration with electronic health, several monitors for patient parameters such as heart rate, blood pressure, mean arterial pressure, etc. and the hospital's drug order and delivery systems to be effective.

Chapter 3

Hurdles to CDSS Adoption

There is now increasing evidence to suggest that well implemented CDSSs can significantly improve quality of care [63], [64]. However, despite several advantages, several challenges continue to inhibit wider CDSS adoption [34]. Some challenges are non-technical, i.e., require changes to legislation, incentive mechanisms and practitioner education and training, and are beyond the scope of this work. But, several limitations in existing CDSS technology have also inhibited further adoption. This chapter discusses said challenges, and the progress made by existing state of art towards addressing them.

Recall, from **??**, that in 2017, the National Academy of Medicine published a report on CDSSs that laid out a roadmap to optmize CDSS uptake in medicine. According to the report, several challenges need to be tackled for wider adoption, such as:

C1. Absence of systematic ways of *validating content* in a *reliable*, *accessible* and *updateable* manner.

C2. Lack of *reliable*, *shareable* CDSS content that can be easily adopted across healthcare organizations and their (Information Technology) IT systems.

C3. Technical difficulties of sharing due to *need for adaptation* to diverse Electronic Health Records (EHR) systems.

C4. *Suboptimal* User Interfaces (UIs), implementation choices and workflows.

3.1 Addressing Adoption Hurdles

CDSS first appeared in the 1960s, and have evolved over time to address aforementioned challenges. The following sections describe progress made towards addressing aforementioned challnges.

3.1.1 Monolithic CDSSs

Early CDSSs were developed as monolithic standalone systems that were self-contained, requiring direct user input for clinical data [65]. Such systems co-existed with primitive electronic health records (EHR) systems, and thus had to rely on manual data entry before administering support.

Several successful CDSSs implementations utilized a standalone architecture. Early CDSS implementations such as MYCIN [66] required the HCP to answer a set of questions to provide advice regarding microbial therapy. Other early CDSSs such as DXplain [67] utilized a wide range of findings (history, data, etc.) to come up with a diagnosis, and is still in active development.

The reliance on manual entry made using such systems time-consuming. As support for EHR matured, CDSSs implementations became better integrated with EHR systems for automated clinical data retrieval. However, with monolithic systems, the integration was usually EHR-specific [65]. Migrating or sharing CDSS content across medical establishments presented significant challenges as a system designed for an establishment's EHR system couldn't be used with a different establishment's EHR system [68].

3.1.2 Modular CDSS Architectures

In **??**, we presented the components that every guidelines-based clinical decision support can conceptually be decomposed into. Early CDSSs from **??** were not designed with modularity that enabled sharing components between different implementations. As the need for scaling CDSSs across institutions grew, component-based architectures that enabled EHR agnostic systems to be developed became prevalent [68].

EHR-agnostic architectures represent a significant step towards addressing several challenges. Such architectures address C3 as the knowledge-base can be shared across institutions with different EHR systems. C2 is partially addressed as the knowledge-base can be independently developed, maintained and distributed. C4 is also partially addressed as decoupled components, such as the UI, are easier to adapt to HCP preferences.

Over the years, several CDSS implementations have utilized a components-based architecture. For instance, in [68], the authors utilize the service-oriented architecture [69] to build a CDSS web service that can be utilized in a completely EHR-agnostic way. Recent efforts include CDSSs platforms such as EvidencePoint that enable CDSS to be integrated closely with the hospital's EHR without being tightly coupled [70]. EHR-agnostic architectures allow decision support to be administered using the EHR's UI. Given their prevalence in modern medical establishments, EHR systems have become integrated into workflows, and HCPs are accustomed to using them. Dispensing clinical decision support through the EHR's UI is vital for adoption, as better workflow integration can lead to higher adoption [71], [72].

Approaches that utilize a component-based architecture enable medical knowledge to be shared more efficiently. But, it's possible for medical knowledge itself to be incorrectly encoded. BPGs are generally expressed as long textual documents meant to be understood by HCPs [73]. To build a CDSS, the BPG has to be systematically expressed in a computable medium. This translation process, referred to as knowledge formalization, is generally ad-hoc, and can be the source of inconsistencies in encoded medical knowledge resulting in CDSSs that render wrong advice [74].

Typically, in order to translate textual guidelines to a computable medium, experts in medicine collaborate with computer scientists and software developers to come up with a requirements document. This is subsequently utilized to develop the knowledge base, i.e., the computer interpretable encoding of the BPG [36] in a traditional programming language. Thus, the BPG as a functional specification for the knowledge base. For instance, in the case of the aforementioned EvidencePoint, the knowledge base is expressed in Javascript [70].

However, since computer scientists/software developers don't understand medicine, and experts in medicine typically don't understand programming languages, it's possible for the functional specification, i.e., the BPG, to diverge from its implementation, i.e., the knowledge base. Thus, C1 from **??** remains unaddressed.

3.1.3 Computer-Interpretable BPGs

CDSSs are safety-critical systems, where bugs can have serious consequences. Thus, ensuring correctness of CDSSs is vital to widespread adoption. To ensure bugs arising out of divergences between the textual

BPG and its computable translation, i.e., the knowledge base, the gap between the BPG and the knowledge base must be eliminated. To this end, instead of encoding BPGs in a conventional programming language, domain-specific languages (DSLs) for directly expressing BPGs in a computer-interpretable manner can be utilized. By emphasizing comprehensibility to HCPs, these DSLs enable HCPs to validate the accuracy of encoded medical knowledge. A computer-interpretable BPGs can both as the specification, i.e. the textual BPG, and the implementation, i.e., the knowledge base, thereby eliminating the specification-implementation gap.

In 1989, the Arden Syntax was developed as a result of incompatibility of medical knowledge between medical institutions [75]. While it was designed to represent simple guidelines, such as those related to reminders, more complex treatment protocols couldn't be represented. Arden syntax had a formal Backus-Naur Form (BNF) syntax definition, but no formal semantics, or a standardized execution engine. Instead, ad-hoc execution engines for the language have been developed over time [37].

Shortcomings regarding ability of the Arden Syntax to represent complex treatment protocols were addressed by formalisms such as GLIF [76] that permit encoding complex guidelines through the use of a multi-layer approach to represent both high-level medical knowledge and low-level implementation details. However, just like the Arden Syntax, GLIF lacks a formal semantics or formal analysis tools.

The need for formal analysis is identified by Asbru: a formalism with formally defined syntax and semantics [40]. 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 [77]. However, the formal semantics of Asbru have been only partially defined, and is insufficient to implement tools for the language [39]. The importance of a complete formal-semantics is identified and addressed by PROforma [39], 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) [39]. 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.

Several endeavors, mainly academic, but also involving the industry towards making CDSSs safer, easier, effective and cheaper. We briefly talk about some related work in this chapter to provide an overview of existing progress and remaining questions. But, **??** provides an in-depth discussion of related work. In **??**, the earliest CDSSs attempted to increased adherence to evidence-based best practices at medical establishments. To ensure CDSSs could be scaled better, components-based architectures, discussed in **??** enabled medical knowledge to be developed and maintained independently of other components (such as system UI). To ensure that medical knowledge is indeed encoded correctly, several DSLs that eliminate the gap between a textual BPG and the CDSS' knowledge base have been developed. However, the following issues still remain:

- Interpreters and compilers for said DSLs are developed in an ad-hoc way, and can be prone to bugs that manifest during execution.
- There is a lack of formal analysis tools such as model checkers and program verifiers that can be utilized to establish that the computer-interpretable BPGs satisfy desired safety and liveness properties.
- DSLs lack complete formal semantics that can be utilized as a language model for a suite of formal analysis and execution tools.

Addressing aforementioned issues is vital to improving CDSS adoption. In order to have trustworthy and useable computer interpretable BPGs, it's important to buttress HCP-friendly DSLs with a suite of formal analysis tools that can establish desired safety properties, thus comprehensively addressing C1.

Chapter 4

Related Work

In **??**, we provided a brief overview of existing approaches and their limitations. This chapter provides a comprehensive discussion of related approaches. Recall from **??** that implementing a guidelines-based clinical decision support system requires collaboration between experts in medicine and software engineers for knowledge formalization, i.e., the process of encoding medical knowledge in textual BPGs in some programming language. Using a conventional programming language for knowledge formalization can lead to an inaccurate encoding medical knowledge, as experts in medicine, being unaccustomed to computer programming, cannot validate the accuracy of the encoding. To address this, DSLs designed specifically for expressing BPGs in a computer-interpretable format are utilized. BPGs expressed in such languages can serve as both textual guideline documents and knowledge bases in CDSSs. But, medical knowledge in BPGs has also been expressed and formalized using other non domain-specific approaches. The discussion in this chapter has also been split along the same lines. In **??**, we discuss approaches involving DSLs for computer interpretable guidelines. In **??**, we discuss other approaches that aren't specific to BPGs.

4.1 DSL-based Approaches

4.1.1 Arden Syntax

The Arden Syntax is among the earliest and most widely-used standards for expressing medical logic, with the first draft of the standard appearing in 1989. It was also among the earliest attempts to create a domain specific language specifically for use in CDSSs [78].

In Arden Syntax, code is organized into self-contained medical logic modules (MLMs) that have a well-defined structure to separate higher-level medical logic from low-level implementation details such as variable declarations. The language continues to evolve to accommodate diverse uses, and is supported by multiple execution engines [79], [80].

The maturity of the Arden Syntax platform is reflected in its use to implement a large and diverse set of CDSSs. These include:

- Systems for monitoring and infections surveillance [81], [82].
- Treatment planning and decision support [83], [84].

Being one of the earliest DSLs specifically designed for CDSSs, the Arden Syntax has found widespread adoption. But, the language also has several limitations. Notably:

- Support on simple, independent guidelines instead of complex treatment workflows [37].
- Lack of formal semantics and clarity in the language specifications [78].

4.1.2 Dilemma

The Dilemma project was among the first attempts at standardizing BPGs to originate out of Europe. In Dilemma, a protocol describes the set of activities that must be carried out in order to reach on objective based on clinical indications [85]. Protocols are recursively broken into smaller elementary protocols. Each protocol has one or more associated indication that determine when it becomes applicable, and a set of objectives that define the expected outcome.

Applying a protocol to a patient state is defined as a procedure, where leaf-level procedures are specifically referred to as activities. Activities are carried out by agents (usually HCPs such as doctors, nurses, etc.), and typically involve:

- Maintaining or changing patient state through treatment steps.
- Obtaining information or confirming procedures.
- Communication or interaction with other agents.

Dilemma was utilized to implement guidelines from several European countries, such as primary care guidelines from the Netherlands and the UK and cancer treatment guidelines from France [85].

4.1.3 GEODE-CM

Guided Entry of Data Elements for Clinical Management (GEODE-CM) was an early DSL developed by the Decision Systems Group at Harvard University Medical School. The GEODE-CM DSL utilized a state machine-based architecture to represent medical knowledge. Clinical problems in GEODE-CM were broken down into clinical management states, where entry, exit and transitions between states were determined by data collected during execution [86]. Each clinical management state was represented as state machine nodes, and edges between nodes represented transitions between the various clinical mangement states [87].

4.1.4 EON

The EON langauge, developed at Stanford University, also utilizes a state machine-based architecture for representing medical guidelines. Treatment protocols are first recursively composed into smaller granular elementary protocols that cannot be decomposed any further. The protocols are then represented via directed multigraphs, where nodes represent both patient and treatment state, and transitions represent actions or changes in patient conditions [88].

EON has many similarities to GEODE-CM. Medical knowledge in both DSLs is expressed using a finite-state machine notation, where nodes are patient or treatment states and edges represent actions or changes in state. However, EON, unlike GEODE-CM:

- Has an informal, yet comprehensive operational semantics.
- Support for sequencing, looping and synchronization constructs to support guidelines with multiple concurrent actions [88].

While Arden Syntax emphasizes expressing medical logic using independent, modular medical logical modules, EON state machines are more tightly coupled, enabling representation of complex protocols and guidelines [88]. EON has been used to implement complex CDSSs for management of conditions such as AIDS [89] and Breast Cancer [88].

4.1.5 GLIF

The Guideline Interchange Format (GLIF) was introduced in 1998 as a result of a collaboration between researchers from Columbia University, McGill University, Harvard University and Stanford [37]. Good medical guidelines improve healthcare quality, but require substantial work to develop and maintain. At the time, guidelines were published through unstructured text documents that were not easily shareable. The motivation behind the GLIF project was to enable BPG between institutions sharing through the development of:

- A standardized electronic format for rapid dissemination.
- A repository of guidelines to prevent duplicated effort.
- Tools that HCPs could utilize to retrieve and execute relevant guidelines.
- Analysis tools that enabled authors to develop and publish high-quality, unambiguous guidelines.

The GLIF team comprised of research groups from various universities where other languages, including Arden Syntax (**??**), GEODE-CM (**??**), were developed. Thus, perspectives and learnings from existing efforts guided the development of GLIF, including borrowing certain useful constructs directly from EON [87]

In GLIF, guidelines are expressed using the GLIF class containing relevant attributes corresponding to treatment information. One such attribute is an unordered list of all guideline steps. The guideline specification itself is a directed graph defined over collection of said steps. A step can be one of the following:

- Action steps: Specify clinical-care actions during treatment.
- Conditional steps: Determine control flow based on logical statements.
- Branch steps: Enable flow to multiple guideline steps concurrently.
- Synchronization steps: Converge concurrent flow back to a single guideline step [87].

GLIF was used to implement several CDSSs, such as a web-based application for support in clinical consultations [90] and the Partners Computerized Algorithm Processor and Editor (P-CAPE) system for creating and deploying BPGs [91].

The experience and lessons from implementing aforementioned CDSSs in GLIF also revealed certain limitations of the language, such as:

- Integration of heterogeneous clinical systems was challenging.
- Guidelines were cumbersome to develop and readability was suboptimal.

To address these shortcomings, an updated version of the language called GLIF3 was introduced [76]. GLIF3 utilized different abstraction levels to present relevant information based on the user's role. Guidelines are modeled using three hierarchical "abstraction levels":

i. The conceptual level, where the guideline is represented as a flowchart for consumption by HCPs.

ii. The computable level, where information regarding execution exists to enable use in a CDSS.

iii. The implementable level, where additional information regarding integration with institution-specific EHRs resides.

Like GLIF, GLIF3 has also been used to implement several CDSSs, including systems for depression screening and management [92] and hyperkalemia patient screening [93]. But, despite its use in several systems, the semantics of the GLIF framework remain only informally defined.

4.1.6 PROforma

PROforma was initially developed at the Cancer Research UK Advanced Computation Laboratory, and has been utilized to develop several CDSSs [37]. In PROforma, guidelines are modeled as tasks and data items. Tasks are hierarchically organized into plan, and may be further divided into:

- Actions: Procedures that must be performed in an external environment.
- Enquiries: Guideline points at which data must be obtained.
- Decisions: Points at which a choice determines further control flow [39].

PROforma guidelines can be visualized as directed graphs, with nodes representing actions and edges constraints on control flow. Pictorially, actions are represented as squares, enquiries as diamonds and decisions as circles. This enables PROforma guidelines' visual representations to resemble flowcharts and medical algorithms that HCPs are already familiar with.

A PROforma task has associated properties that determine how it's interpreted. These include:

- Captions and descriptions that improve guideline comprehensibility.
- Preconditions that specify conditions for a task's execution to begin.
- Scheduling constraints that enable synchronization between concurrent tasks.

Additionally, plans have termination and abort conditions that specify successful end of the current plan to facilitate execution of successor plans.

Using PROforma requires a suite of tools called Tallis [94], written in Java, that include a composer for creating and viewing guidelines, a tester for debugging them, and an execution engine for running them [39].

PROforma has been used to implement several CDSSs that have demonstrated improvements in care quality in clinical settings. Among the first applications of PROforma was the CAPSULE system that assisted general practitioners prescribe medicines [95]. Similarly, the LISA system was designed to HCPs with dosages for pediatric patients with acute lymphoblastic leukemia [96]. Other applications include support with image interpretation and diagnosis [97], and support for breast cancer diagnosis and treatment [98].

PROforma is also among the first DSLs to identify the for a formally defined syntax and semantics, and expend significant effort in doing so. In [39], the authors enunciate that PROforma "must have" open source syntax and semantics definitions to ensure that medical logic in PROforma is unambiguous, and to ensure tools "correctly" read and process PROforma.

4.1.7 Asbru

Asbru, developed by collaborators from Stanford, Vienna University of Technology, and Ben Gurion University, focuses on time-oriented clinical guidelines [37]. The language attempted to address limitations from existing approaches such Dilemma (**??**) such as:

- Lack of human-readability.
- Inability to integrate well with electronic patient records.
- Inability to model flexibility in guideline-prescribed steps.

To address these shortcomings, especially readability and modeling flexibility, Asbru utilizes a skeletal plans-based approach. Skeletal plans are a sequence of generalized steps that are instantiated in a specific problem context to solve a given problem [99]. Thus, skeletal plans provide a broad framework for solving a class of problems, instead of fine-grained details required to solve a specific problem. This flexibility enables Asbru guidelines to have intended objectives that can be achieved in different ways. For instance, to manage diabetes, an HCP can either administer additional insulin, or instruct the patient to consume less carbohydrates during a meal. Both actions have the intended outcome of moderating blood sugar levels. Specifically, an Asbru skeletal plan based guideline has:

- Actions to be performed by the HCP.
- An intended plan made of action described as pattern of actions.
- The intended outcome described through patterns over patient states [100].

When executed, the HCP applies the guideline by following actions, which are observed, recorded and abstrated over time into an absracted plan. both patient state, and the inferred intention is also recorded [100]. Asbru's execution model also provides the ability to critique execution along these axes:

- Guideline-prescribed actions vs HCP followed actions.
- Guideline-intended plan vs HCP followed plan.
- Guideline-intended state vs the HCP's intended state.
- Guideline-intended state vs actual patient state.
- HCPs's intended state vs actual patient state.

These axes provide 32 different scenarios of guideline execution [100].

Asbru provides comprehensive support for temporal aspects of guidelines through the use of time reference annotations. These represent uncertainty in starting, ending and duration of time intervals. An annotation can be an absolute reference point, a reference point with uncertainty (defined by a region), or a function of some previous plan. Deviation from the reference point represent uncertainty in starting. Through the use of these annotations, intervals have an earliest start shift, latest start shift, earliest finishing shift, latest finishing shift and minimum duration and maximum durations [100].

A guideline in an Asbru library consists of a set of plans and corresponding time annotations. During execution, the interpreter attempts to decompose the plan into subplans from the library. If a plan is

non-decomposable, it is considered to be atomic, usually corresponding to actions to be performed by an external agent such as obtaining information, administering treatment, etc.

Library plans have four states: considered, possible, rejected and ready that determine whether a plan is applicable. During execution, a ready plan is instantiated. Apart from the name, arguments and time annotation, a plan also has:

- Preferences: Factors that influence the selection of a plan to achieve a goal. Examples include start conditions for the plan, heuristics (such as exact-fit) to specify a selection criteria, etc.
- Intentions: High level goals of the plan.
- Effects: Functional relations between arguments and measurable clinical parameters.
- Body: Other plans (that also have the same structure) that may be executed in parallel, in sequence, or some combination thereof [100].

The Asbru language has been used to implement several guidelines in Endocrinology, Pulmonology, Gynaecology, such as diabetes mellitus, jaundice in infants, and mechanical ventilation in premature babies [101]. The language has a formal syntax in BNF, and the earlier versions of the language also have a formal structural operational semantics [102].

During the development of Asbru, it was realized that CDSSs are limited by a lack of standardized EHR formats, and in many settings, a complete lack of EHR support. To address these issues, a new Asbru-based system that enables implementing CDSSs in settings with complete, partial, or no EHR, called Spock, was developed. Spock utilizes Hybrid-Asbru, a semi-formal variant of Asbru developed for the Digital Electronic Guideline Library (DeGeL) [101], [103]. Unlike Asbru, Hybrid-Asbru lacks a complete executable semantics and an interpreter with any correctness guarantees.

KIV-based Verification

In [104], a formal-methods approach to improve Asbru-based guidelines is presented. The authors identify that in order to use formal methods for systematic verification of guidelines and protocols requires an appropriate representation language with a clear and well-defined semantics. Two real-world guidelines were chosen, and analyzed using a by gradually transforming them into more formal representations. First, the protocols are transformed from plain text into Asbru plans. Then, the Asbru plans are transformed into KIV programs encoding said plans, for analysis using the KIV theorem prover [105]. A crucial step in using KIV is the definition of the formal semantics of the main parts of Asbru, which can also be utilized for verification of guidelines beyond those presented in [104]. As part of formalizing the guidelines, several properties relevant to clinical guidelines were identified, formulated over Asbru code, formalized in KIV, and established against the KIV code.

To evaluate the effectiveness of their approach, the authors chose the following guidelines for the management of:

(a) Hyperbilirubinemia (Jaundice) in normal term infants by the American Academy of Pediatrics [106].

(b) Diabetes mellitus type 2 by the Dutch Association of General Practitioners [107].

The formalization process required significant effort, where the following problems were observed:

- **Ambiguity**: Several terms presented interpretation related challenges, were details were left to the HCP. While missing details may be overlooked in paper versions, they needed accurate descriptions in the computer-interpretable counterparts.
- **Incompleteness**: Expected guidance or information was found to be missing in some cases. For instance, conditions such as "rapid increases" in certain patient parameters lacked accurate descriptions, and interpretations were left to the HCPs.
- **Inconsistencies**: Certain guidelines were found to lead to conflicting advice for the same patient data. Such scenarios were deemed to be very serious, as following the guideline could potentially result in non optimal treatment.
- **Redundancies**: Instances of repeated recommendation were often discovered during the process of formalizing the guidelines in Asbru.

Asbru Semantics The semantics of Asbru are defined for two purposes: (a) to formally document the language specification, and, (b) to derive a calculus for verification. To this end, a structural operational semantics (SOS) is defined [108]. Recall from **??** that Asbru programs consist of hierarchical plans that execute concurrently. The semantics consist of an execution engine that encodes concurrent execution of plans, and a data abstraction unit that abstract signals to the said plans.

The semantics of plan execution are described by expressing the transition system as a statecharts [109]. Statecharts are directed graphs representing state machines [110]. Plan states represents statechart nodes, and edges transitions. The transitions can then be conveniently formalized as SOS rules. For instance, the rule:

$$\frac{[\![\text{P.State} = \text{considered}]\!]_\sigma \qquad \text{da}\,(\Omega, \sigma) \rightarrow *\text{da}\,(\text{true}, \sigma')}{\text{psm}_P\,(\sigma) \rightarrow \text{psm}_P\,(\sigma'\,[\text{P.State} \leftarrow \text{ready}])}$$

expresses the semantics of a plan P (expressed as a statechart psm) to transition from state "consider" to state "ready". The SOS rule expresses that if, under the environment σ, (a) the filter condition Ω evalutes to true by the data abstraction unit da, and, (b) the plan's current state is consider, then the plan P advances to state ready.

Once the semantics have been formally captured, Asbru plans are translated to programs into KIV. In KIV, program behavior is specified using standard dynamic logic program constructs, such as variable assignments $v := \tau$, conditional if(Ψ) S_1 else S_2, iteration while(ψ) do S, parellel execution $S_1 \parallel S_2$. The translation from Asbru to KIV is performed in a structure-preserving way, as most Asbru constructs can directly be mapped into a corresponding KIV one. For certain features of Asbru, additional encoding is required. The KIV encoding makes KIV's theorem proving capabilities available for formal verification, where properties are stated as Interval Temporal Logic (ITL) formulas [111]. Notably, plan intentions can be encoded as logical formulas, and verified against the plan's KIV translation. However, the verification may require experience with theorem proving when dealing with properties of interest.

The outlined based approach demonstrates that formal methods can successfully be utilized for improving real BPGs. The analysis revealed several issues with both protocols, some of which were deemed very serious. Moreover, the corresponding Asbru plans were verified to satisfy their stated intentions, improving system safety. But, the approach has the following limitations:

- While the formal semantics of Asbru intends to serve as a manual for implementing execution engines for Asbru, existing engines are not derived from it. This leaves it possible for the execution engine to

implement semantics that diverge from the SOS semantics used for verification, leaving it possible for the verification results to not hold on As newer versions of the language are introduced, effort has to be expended into keeping the semantics and all tools in sync.

- Only a subset of the language has a direct mapping into KIV constructs. Manual effort and expertise is required for encoding constructs without direct mappings. Expressing intentions as logical formulas also requires expertise, and, may be error prone as the property isn't directly written over the Asbru program, but its translation in KIV.

4.1.8 Guideline Acquisition, Representation, and Execution (GLARE)

The GLARE system attempts to decouple the representation of knowledge in BPGs to their execution [112]. The GLARE representation language represents guidelines using actions, that may be either atomic or composite. Atomic actions involve elementary steps that cannot be broken down further, and can be one of the following:

- Work actions: Steps or procedures that must be performed at a particular point during guideline execution.
- Query actions: Obtaining information from the outside world, for instance, from doctors, or databases.
- Decision actions: Selecting alternate actions in the guideline based on guideline-specified criteria.
- Conclusion actions: Finalizing the outcome of executing the guideline .

Atomic actions provide the elementary primitives for modeling medical knowledge, but, to specify a guideline, additional mechanisms for expressing relations over said actions is needed. Relations in GLARE can either be structure-related or control-related. Structural relations enable building trees from actions, where nodes represent composite actions, and leaves atomic ones.

Guideline execution requires specifying temporal ordering on actions. This is supported through control-related relations, which can be one of:

- Sequence relations that enforce a strict ordering on actions, where, if any one of the actions in the sequence fails, then the entire sequence is considered to have failed.
- Concurrent relations that allow actions to run in any order, including in parallel.
- Decision relations that allow execution of alternative actions based on the conclusion of a decision action.
- Iteration relations that allow looping behavior, where an action may be repeated for a given duration, or, until a particular condition is met.

To enhance usability, actions in GLARE must further follow a particular structure. Work actions are further refined into clinical procedures and pharmacological prescriptions. Additionally, work actions may additionally have properties such as:

- Preconditions that specify the context under which the action may apply.
- Cycles that specify temporal behavior [112].

Guideline Acquisition and Consistency Checking

The GLARE framework provides a graphical tool for authoring guidelines that enables HCPs to specify GLARE-based guidelines. The interface has distinct "windows" that deal with different guideline aspects.

The structure window is used to represent relations between different actions through a tree. Each node represents an action, and its children represent sub-actions that must be performed as a part of executing the main-action. A separate control window is utilized to enforce execution ordering on actions. For a given action, its sub-actions are represented as nodes in a control graph, where edges specify control relations between said nodes.

GLARE framework provides mechanisms for ensuring modeled guidelines do not have inconsistencies [112]. GLARE supports the following two types of checks:

Standardization Inconsistencies: Writing guidelines in GLARE occurs in one of two modes: safe and advanced. In safe mode, when a new term is introduced as a part of an action's description, GLARE checks if the term has already been defined in its existing database, and disallows the introduction if the term is not found. This ensures standard terminology and range of values is used throughout the system. In advanced mode, new terms can be introduced, and a warning is issued, under the assumption that the reason for ignoring existing checks is intentional.

Logical Inconsistencies: The checks for logical inconsistencies ensures that certain "logical design criteria" are followed. The following inconsistencies are disallowed:

- Cycles: Introduction of cyclic behavior through control arcs is disallowed, as repetitiveness can be specified through attributes.
- Alternatives without decisions: An action that models available choices must follow a decision action. This ensures that if a guideline provides multiple paths to reaching a goal, then there exists an explicit way of discriminating between them.
- Queries before decisions: When a decision action is reached, all data must have already been obtained. This ensures that all data necessary to make a decision is already available when control reaches a decision action.

Temporal Inconsistencies: These checks try to ensure semantic correctness from a temporal standpoint [112]. Recall that sequences of actions and sub-actions form graph structures. Minimum and maximum durations of actions and minimum and maximum delays between actions form logical constraints. These constraints are utilized in the control graph to infer new constraints, that can be checked for consistency using the LaTeR temporal manager [112], [113] .

Execution of GLARE Guidelines

Execution in GLARE corresponds to an instantiation of the guideline with a specific patient's data. GLARE's execution module is responsible for:

1) Interacting with the patient's EHR to generate required SQL [114] queries to obtain relevant data.
2) Maintaining the set of applicable guidelines for each patient, and their status (such as active, suspended, etc.).

3) Maintaining the list of patients undergoing treatment.

4) Storing logs for each guideline execution to append patient history.

The execution module also supports different "modalities" to enable use in diverse settings, such as clinical practice, or training. For example, in actual practice, time delays between actions are modeled using wall-clock time, while in training, a simulated time might be used instead.

The GLARE framework has been used to implement real-world guidelines in different domains, such as bladder cancer, reflux esophagitis and heart failure [115]. The approach has been instrumental in demonstrating viability of standardized executable guidelines. However, despite use is safety-critical settings, the execution module has an ad-hoc Java implementation with no correctness guarantees, and utilizes an informal and incomplete paper-based execution semantics [112].

4.1.9 Guideline PRocess cOnformance VErification Framework (GPROVE)

GPROVE is a set of tools that allow users to specify guidelines and reason about compliance of observed behavior. Specifically, the framework allows a-posteriori verification of workflow executions [116]. The GPROVE framework consists of:

- The Guideline prOcess Specification Language (GOSpeL) language for graphically representing guidelines.
- A translation to the Sciff language [117].
- Verification of compliance of an execution trace to the guideline.

Next, we briefly describe each component.

GOSpeL: GPROVE's modeling language uses flowchart-derived concepts of blocks and relations between blocks to represent medical guidelines. Blocks belong to one of three families:

- Activity blocks that contain various guideline actions that HCPs must perform. Said actions are expressed through activities. Atomic activities represent actions that cannot be further decomposed, and can be combined to form more complex composite activities.
- Gateway blocks that dictate divergence and convergence of control flow. These blocks enable fork and join constructs for modeling concurrent behavior.
- Start/end blocks that dictate terminal flowchart nodes.

Sciff Translation and Verification: The modeled guideline is mechanically translated to Sciff – a constraint logic programming framework. This translation consists of various constraints describing expectations of what should or should not happen. During runtime, a log of the actually observed events is generated from guideline execution. The log is then checked against the generated constraints, and a violation is raised if the log does not satisfy the said constraints [118].

The GPROVE framework is motivated by a different objective than other framework described in this chapter. Notably, GPROVE focuses on proving a-posteriori compliance to guidelines, and intends to serve as a complementary tool, instead of a standalone one [119]. But, the approach has some limitations. First,

the GPROVE modeling language does not have a formally defined semantics. Instead, the semantics are embedded in the Sciff translator and execution trace generator. Second, there exist no proofs of correctness of the Sciff translator, or methods of checking equivalence between the GOSpeL code and its Sciff translation, leading to a gap that can potentially result in the Sciff model to diverge from the underlying GOSpeL code.

4.1.10 HELEN

The HELEN-Project was implemented and tested at the Heidelberg University Medical Center's Neonatology Department, and specifically focused on the following:

- Standardizing guidelines from diverse sources and origins.
- Support for convenient local adaptions to existing guidelines.
- Direct utilization in decision support for use in actual practice [120].

In HELEN, guidelines are organized into classes, that may have attributes and subclasses. The subclasses represent an "is a" relation, i.e., the subclass inherits attributes of all its superclasses. For knowledge acquisition, HELEN uses Protégé framework [121]. To improve comprehensibility, the HELEN framework includes a graphical interface that allows HCPs to view the guideline logic.

The framework also includes the Guideline Execution Engine (GEE) that facilitates BPGs to be used in CDSSs. The GEE traverses the guideline in order to:

- Communicate with HCPs about recommendations and scheduled actions.
- Evaluate guideline logic to determine execution flow.
- Perform administrative tasks such as logging performed actions and manage HCP information (such as roles, access, etc.) and patient data [120].

As HELEN was developed at the Heidelberg University Medical Center's Department of Neonatalogy, it was used to implement diverse BPGs for use at the department. In particular, the following BPGs were modeled:

- Management of hyperbilirubinemia in healthy newborns, published by the American Academic of Pediatrics (AAP) [122].
- Management of apnea in pre-term newborns [123].

While the BPG for hyperbilirubinemia management is largely a textual document with an accompanying non-executable flowchart, and is utilized mainly for illustration and training purposes. The BPG for apnea management, on the other hand, forms a complex CDSSs that utilizes nested algorithms to administer both advice and timing support [120]. This highlights a versatile system capable of accommodating diverse needs.

While the framework can be used to implement and execute BPGs, it does not provide a way to systematically analyze them for semantic inconsistencies – limitations were recognized in [120]. The language does not have a formal syntax and semantics, and the main execution engine in Java has been implemented using an ad-hoc semantics. Thus, while HELEN moves the needle towards wider BPG adoption, as acknowledged by the developers themselves, more work is needed to address challenges for formal analysis and execution with correctness guarantees.

4.1.11 Prodigy

The Prodigy initiative was an effort by the UK's National Health Service (NHS) to develop CDSSs for use by general practitioners (GPs). There have been several iterations of the Prodigy system to address limitations observed in practice. The Prodigy I and Prodigy II were closely coupled with the EHR systems, and were not amenable to sharing. Prodigy II found wide use, demonstrating benefits of such systems to GPs, but was found to be inadequate to express important guidelines for chronic diseases. As a result, Prodigy III was developed to support modeling of such diseases [124].

In Prodigy III, guidelines are expressed through scenarios and action steps. Scenarios are easily recognizable patient states dictating if the guideline is applicable. Each scenario can have one or more outcome assessments with associated action steps that specify guideline-coded recommendations to manage the outcome. The action steps can:

- Start, stop or modify activities that occur over time, such as administering time-metered drug dosages).
- Specify instantaneous actions, such as giving an immediate drug dose, or recording a patient assessment.

In cases when a single action is insufficient to describe recommended treatment, a sub-guideline step that specifies additional treatment in a structured way can be utilized. This hierarchical representation methodology makes Prodigy guidelines resemble their paper-based counterparts.

Prodigy, like most CDSSs frameworks, depends on patient data to be available through EHRs or HCP assessments. In real clinical settings however, it is unreasonable to expect perfect data to be consistently available.It is often the case that (a) decision support is started after some treatment has already been performed, leading to an inconsistency between the perceived digital state and actual real-world state, and, (b) patient data is unreliable, obsolete, or simply unavailable. To address this, in Prodigy III:

- Scenarios can be broad and flexible, and serve as intuitive access points for the clinicians in situations where perceived digital state may be inaccurate.
- Default values for patient parameters and other assessments are utilized if patient data is unavailable or inconsistent [124].

Prodigy III differs in its approach towards decision support, in comparison to alternative approaches such as GLIF (**??**), EON (**??**) or Asbru (**??**). Unlike said alternatives, that offer complex concurrency and synchronization primitives to capture treatment details, Prodigy relies employs a simpler action plans-based approach that Prodigy's developers argue is sufficiently expressive for most guidelines. Other approaches emphasize HCP-system interactions and access to good quality and up-to-date clinical data to make complex assessments about patient state and dispense detailed advice. Prodigy, on the other hand, leaves the complex assessment of patient state to the HCP, and provides more abstract advice in the form of choices, expected outcomes and associated steps to reach said outcomes [124].

Prodigy, as an early CDSS framework, enabled greater adoption among HCPs. The focus of the Prodigy initiative was to standardize guidelines and improve uptake and adoption. Thus, efforts weren't expended into ensuring that the guidelines had a well defined semantics, and tools that were based on on it.

4.1.12 SAGE

The Standards-based Active Guidelines Environment (SAGE) was the result of a collaboration between groups at University of Nebraska Medical Center, GE Healthcare, Stanford University and Mayo Clinic. The

project focused on creating relevant infrastructure for shareable, standards-based guidelines amenable to use in decision support, across diverse EHR environments.

Integration with care workflow has been shown to be critical to CDSS effectiveness. SAGE, therefore, focuses on ensuring that the language and associated infrastructure support this goal. Instead of waiting for the user to control the workflow, SAGE uses available information to suggest applicable best-practices in the form of events. To this end, the language itself is expressive enough to specify workflow contexts that decide when guidelines become applicable [125].

In SAGE, guidelines are represented using recommendation sets, or, collections of recommendations applicable in one or more computable contexts. A context formalizes the clinical settings under which the recommendation is applicable. A recommendation further comprises of action specifications and decision logic that specify guideline-intended HCP behavior. The recommendation set itself is organized as a directed graph that captures relation between recommendations [125].

SAGE guidelines are interpreted by the SAGE execution engine. The engine interprets the context, action specifications and recommendation-relations and dispatches decision support to the EHR. The engine's event listener module is responsible for receiving any state changes from the EHR [126].

SAGE's design is motivated by: (i) using native EHR to dispense support, (ii) integrating with workflows as seamlessly as possible, and, (iii) enabling interoperability between different EHR systems. Its architecture and design choices enable interaction with heterogeneous sources of information, and, de-duplicating replication across EHR implementations [125]. This is reflected in the fact that SAGE has been used to implement guidelines at EHR systems at the University of Nebraska Medical Center and the Mayo Clinic [125]. But, SAGE does not provide tools for validating guideline content, and, execution engine doesn't have any correctness guarantees either. While the execution engine does provide some validation checks [126], SAGE does not have dedicated validation or verification tools.

4.2 General Approaches

4.2.1 T/Gen

The T/Gen is a lisp-based project that enables validation of CIGs through test-case synthesis. While primarily developed for GLIF (**??**), it can be made to work with any guideline modeling formalism.

The T/Gen tool takes as input a GLIF guideline encoded as a Lisp program, and an initial condition set that defines possible clinical scenarios for a program, and additional logical constraints that define when said clinical scenarios are feasible. T/Gen takes these initial conditions and constraints, and generates a set of meaningful clinical scenarios under which the guideline can be tested [127].

For instance, consider the guideline in **??** with initial conditions $\varphi_1, \varphi_2, \varphi_3$. A naive way to test the system would result in 8 distinct conditions $\varphi_i = Y/N$ for $i = 1, 2, 3$. T/Gen, however, would only generate 4 conditions $\{\varphi_1 = Y, \varphi_2 = Y\}$, $\{\varphi_1 = Y, \varphi_2 = N\}$, $\{\varphi_1 = Y, \varphi_3 = Y\}$, $\{\varphi_1 = Y, \varphi_3 = N\}$ that result in firing of $\alpha_1, \ldots \alpha_4$ under which the system should be tested to ensure complete coverage. Additionally, T/Gen further allows the user to specify constraints on the condition set both that can further reduce the number of test cases [127].

T/Gen's white-box testing approach allows significant reductions in the number of test cases required for high test-coverage, and has demonstrated effectiveness in real-world use cases [127]. But, it still has many limitations. Notably:

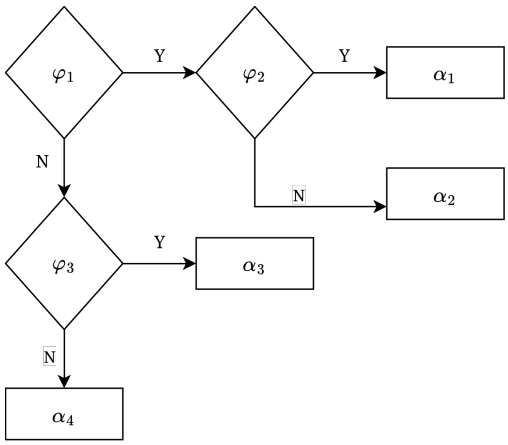


Figure 4.1: T/Gen Example

- T/Gen can work with any CIG language, and supports a simplified version of GLIF. But, this requires the CIG to be re-expressed in T/Gen's simplified language, which requires additional effort, especially in cases of languages that differ significantly from GLIF. Moreover, the translation effort itself may introduce bugs, or, may semantically be different from the original source.
- Uncovering additional constraints requires both efforts and expertise. As acknowledged in [127], a compromise must be made between finding constraints and allowing some redundancy in tests. Moreover, introducing constraints can omit certain scenarios, such as those dependant on ordering of actions, from being tested. Thus, using the tool effectively may require significant effort, depending on the complexity of the system under test.
- T/Gen utilizes an ad-hoc semantics of GLIF, which may differ from that used by GLIF execution engines. Morover, changes to the GLIF language over time would also necessitate corresponding updates to T/Gen.

4.2.2 Verification of Hierarchical Plans

In [128], formal verification of hierarchical plans-based guidelines is discussed. Although the work specifically focuses on guidelines in Asbru, it is applicable to other languages that utilize a plan-based approach to representing BPGs. Recall from **??** that Asbru guidelines consist of hierarchical plans, where a plan may contain other sub-plans, thus creating a tree of plans. Leaves of said trees are plans that cannot be decomposed any further into sub-plans.

A set of eight plan states describe the actual state of the plan during planning and execution. The following seven conditions define transitions between aforementioned states:

1) filter-preconditions that hold if an applicable plan cannot be achieved.
2) setup-preconditions that must hold for an applicable plan to start.
3) activate-condition that specifies whether plan starts automatically or manually.
4) suspend-conditions that specify when a plan must temporarily halt execution.
5) abort-conditions that specify when a plan must be stopped unsuccessfully.

6) complete-conditions that specify successful termination.

7) reactivate-conditions that enable suspended plans to resume.

The technique described in [128] utilizes static analysis to ensure plans are *meaningful*, or, verified to not contain well defined anomalies described in [129]. Given the hierarchical nature of plans, abnormalities are classified into: (a) level 1 that occur within a plan, (b) level 2 that occur within linked plans, (c) level 3 that occur at the scale of BPGs.

In [129], a general foundation for reasoning for reasoning about knowledge bases is provided. A knowledge-base $\mathcal{K} = \mathcal{R} \cup \mathcal{D}$, where:

1. $\mathcal{R}$ is a set rules $\{R_1, \ldots, R_n\}$. Rule R_i is an expression of the form of the form $L_{i_1} \wedge L_{i_2} \wedge \cdots \wedge L_{i_m} \rightarrow R_i$, where $L_{i_1}, \ldots, L_{i_m}, R_i$ are First-Order Logic literals. $\text{antecedent}\,(R_i) = L_{i_1} \wedge L_{i_2} \ldots L_{i_m}$ and $\text{consequent}\,(R_i) = M_i$.

2. The set of declarations $\mathcal{D} = \mathcal{G} \cup \mathcal{L} \cup \mathcal{C}$ representing the set of goal literals, input literals and constraints respectively.

An environment E is defined as a subset of input literals s.t. $E \rightarrow \neg(C\rho)$ for some $c \in \mathcal{C}$ and any substitution ρ. The set of inferrable hypotheses $\mathcal{H}$ is defined as literals that can be inferred using rules, i.e., for $H \in \mathcal{H}$, $\exists R \in \mathcal{R}$ s.t. $H = \text{consequent}\,(R)\,\rho$ for any substitution ρ. $(\mathcal{R} \cup E) \vdash H$ denotes that H is inferrable from rule base $\mathcal{R}$ if there is some environment E s.t. H is the logical consequence of supplying E to rule R.

In [128] describe a methodology to transform an Asbru plans-based guidelines to a rule base. base to capture Asbru's model of plan states MPS is described. The rules specify the sequence in which the conditions of a plan are considered. As MPS encodes the semantics of Asbru, it has to only be specified once.

Next, each plan P_i in the guideline is also encoded as a rule base PC_i, and the guideline is defined as the rule base $\text{PC} = \text{PC}_1 \cup \text{PC}_2, \cup \cdots \cup \text{PC}_n$. Additionally, the following sets are defined:

- PlanSet – the set of all plan
- PatientSet – the set of all patients
- ConditionSet = $\{\text{filter}, \text{setup}, \ldots, \text{complete}\}$ corresponding to plan conditions.
- FinalStateSet = $\{\text{rejected}, \text{aborted}, \text{completed}\}$.

The rule base MPS consists of rules encoding the generic model of plan states. Given parameters pl and pa for a plan and patient respectively, MPS has rules encoding states such as:

- $\text{Considered}(pl)$ encodes the starting state of each plan.
- $\text{Considered}(pl, pa) \wedge \text{filter}(pl, pa) \rightarrow \text{possible}(pl, pa)$ encodes the condition that a plan with filter conditions satisfied is possible for execution.

Next, Asbru plan i in the guideline need to be transformed into its corresponding rule base PC_i. For instance, a plan for treatment of non-insulation dependandant gestation diabetes militus, in patients with normal blood glucose levels, called GDM-TYPE-II, contains rules of the form:

- $\text{Female}(pa) \wedge \text{Pregnant}(pa) \rightarrow \text{filter}(\text{GDM-TYPE-II}, pa)$ encodes that pregnant female patients should be considered for the plan.

- Activate(GDM-TYPE-II, pa) encodes that the plan should be automatically activated.
- State(blood-glucose-high, pa) $\rightarrow$ suspend(GDM-TYPE-II, pa) encodes that if the plan be suspended in case of high blood sugar.
- State(blood-glucose-normal, pa) $\rightarrow$ reactivated(GDM-TYPE-II, pa) similary encodes that the guideline be reactivated when blood sugar returns to normal.

Once the guideline has been encoded as a rule base using the above mentioned transformation, important properties can be checked against the rule base. For instance, conditions in a plan must be satisfiable in order to have an effect on the plan. Formally, this is equivalent to checking all rules in the rule base are *fireable*. If a condition, such as Male(pa) $\wedge$ Pregnant(pa) $\rightarrow$ filter(plan, pa) contains a conjunction that cannot be satisfied, then it does not contribute to the plan's execution, and, is not *meaningful*.

The work methodology describe above represents significant progress towards verification of meaningful properties on real guidelines. But, the methodology also has limitations, such as:

- It only works for certain pre-defined anomalies.
- Expressing hierarchical plans in Asbru as knowledge-bases from [129] utilizes an ad-hoc semantics of Asbru.
- Any update to Asbru would necessitate corresponding updates to the verification tool.

4.2.3 Model-driven Development and Verification

In [130], the authors present a comprehensive approach to development and verification of guidelines, called the Model Driven Architecture (MDA). Instead of utilizing a domain specific language, they choose to model BPGs as UML Statecharts [131]. The choice is motivated by the fact that Statecharts have been used to build systems in diverse domains such as Air Traffic Control [132] and biology [133].

The development and verification methodology has several steps. First, software engineers and experts in medicine collaborate express the guideline using Statecharts. Next, patterns representing common errors in guidelines are utilized to write specifications for the guidelines. The the Statecharts code is automatically translated to PROMELA, the language of the SPIN model checker [134] for verification. To turn the verified Statechart guidelines into a CDSS, the MDA-based approach includes tools that translate the Statecharts into executable Java code that is combined with additional infrastructure to obtain a functioning CDSS.

Property Specification

In [130], the authors deal with medical properties, or properties that are not implementation-related, dealing with aspects such as medical parameters, HCP actions, and overall guideline intentions. As specifying properties requires expertise and a mathematical background. The MDA approach includes a number of common patterns applicable to all guidelines that can be used as templates for writing actual properties. Thus, through the use of these patterns, the authors attempt to enable even non-experts to use the MDA verification methodology.

Property patterns in the MDA verification methodology are based on existing work on simplifying verification for concurrent and reactive systems through safety and liveness patterns that contain informal descriptions of the property and well-defined translations to widely used temporal logics such as Computation Tree Logic (computation tree logic) and Linear Temporal Logic (linear temporal logic) [135]–[138].

To ensure that that patterns presented under the MDA framework cover a wide spectrum of possible properties, the authors collected properties relevant to BPGs from literature, and attempted to represent them using specification patterns proposed by Dwyer et al. in [135]. However, they realized that Dwyer patterns were insufficient to represent all properties of interest. To address this the authors: (a) utilized an expanded pattern representation schema proposed by Ryndina et al in [138], and, (b) proposed a new patterns to encompass all desired properties. This expanded representation schema enables the authors to represent useful and complex properties, at both local and global scopes, and dealing with both safety (something bad never occurs) and liveness (always something desirable eventually occurs).

In [139], MDA is used to implement and analyze the IRC guideline for management of infections due to intravenous catheters. The verification process revealed several inconsistencies, which were fixed. Moreover, the CDSS was shown to satisfy several important properties. Several important patterns were identified during the verification process, which are also applicable to other guidelines. For instance, it is important to ensure that if an empirical treatment has not been ordered, then it is not removed later on. This is identified as an instance of the absence-after pattern, and can be represented by the linear temporal logic formula $\diamond\neg$ (removeTreatment == Empirical $\rightarrow$ beginTreatment == Empirical). Another pattern of interest is possible-existence, stating that, starting from a certain patient state leads to a desired guideline-based outcome. For instance, in case of the IRC guideline, the authors verified that starting from clinical test results that were not indicative of IRC, it is never possible to reach a state where IRC is indicated. Thus, the verification process revealed both inconsistencies in the guidelines and ensured that other desired properties are satisfied [130].

The MDA approach has several strengths. Notably:

- The approach has been used to implement real-world systems and verify important properties against them, as demonstrated in [139].
- Utilizes existing work in modeling large concurrent systems as Statecharts, instead of implementing a DSL from scratch, allowing existing tools for visualization to be used.
- Unlike other approaches, emphasis is given to ensuring that expertise in mathematics is not needed for verification, through the use of property patterns for property specification, and automated translations of the BPG and property to PROMELA for verification using SPIN.

However, the approach also has some limitations. First, in order to execute the Statecharts-based BPG in a CDSS, a translation from the OMG Systems Modeling Language [131] to Java is utilized, and for verification via SPIN, a translation to PROMELA is utilized. Thus, it is possible for the translations to diverge from the underlying BPG in the OMG Systems Modeling Language. This can have serious consequences, as verification results may not be satisfied by the actual implementation. Thus, even though utilizing specification patterns may enalbe non-experts in verification to verify properties PROMELA model, expertise in formal reasoning may be required to ensure that the model corresponds to the original BPG and the corresponding Java code. Second, only a subset OMG Systems Modeling Language is supported by the tools in [130], [139], and the translation are tied to a particular version of the language. Supporting newer versions of the language and updates to the language would require corresponding updates to the entire toolchain. Finally, while several important properties can be established using model checking in SPIN, supporting other techniques such as symbolic execution or deductive verification would require similar translation to other languages, which would further require increase the chances of the many translations diverging from the underlying BPG.

4.3 Discussion

In **??**, we discussed several challenges that limit wider CDSSs uptake in practice. The approaches discussed in this chapter make significant headway into addressing said challenges. These challenges (Cs) were concisely described in **??** as:

C1. Absence of systematic ways of *validating content* in a *reliable*, *accessible* and *updateable* manner.

C2. Lack of *reliable*, *shareable* CDSS content that can be easily adopted across healthcare organizations and their (Information Technology) IT systems.

C3. Technical difficulties of sharing due to *need for adaptation* to diverse Electronic Health Records (EHR) systems.

C4. *Suboptimal* User Interfaces (UIs), implementation choices and workflows.

In **??**, we outlined major themes behind addressing aforementioned challenges. We briefly recap those themes here:

Implementation-Specification Gap Textual BPGs are often written by medical experts, but, the translation into an executable computer program is provided is performed by software developers and knowledge engineers who typically lack the necessary expertise to ensure medical knowledge is correctly encoded. The textual BPG is utilized as a functional specification to develop the CDSSs. This make it possible for the specification to diverge from the implementation, leading to an implementation-specification gap.

DSLs for computer-interpretable guidelines aim to eliminate the aforementioned gap by emphasizing HCP comprehensibility, thereby enabling the CIG to also serve as the guideline's textual description, eliminating the implementation-specification gap, and are vital to developing trusted guidelines with validated content.

Formal Semantics In order to utilize formal methods to ensure CDSS correctness, the underlying language utilized to express guidelines must have a complete, well-defined formal semantics [40]. The semantics can then be utilized to implement tools for execution and analysis, to support a development of verified guidelines.

Formal Analysis Tools A comprehensive suite of formal analysis tools—model checkers, symbolic execution engines, and deductive verifiers—is essential for developing systems with desired safety properties. Additionally, it is crucial to keep these tools updated as the language evolves. As DSLs for computer interpretable guidelines adapt to lessons from implementing best practice guidelines (BPGs), mechanisms must be in place to update all relevant tools, and not just ones needed for immediate use, such as execution engines.

The outlined languages can be categorized into either DSLs for CIGs, or, general ones for building concurrent systems. For instance, the Arden Syntax (**??**) was among the first languages specifically designed to be comprehensible to HCPs, and is still widely used for expressing medical knowledge. But, it's designed for simpler, independent guidelines, and is unsuitable for complex workflows. Other approaches such DILLEMA (**??**) and GEODE-CM (**??**) enabled expressing more complex guidelines, and modular architectures that integrate with diverse systems. However, since these were among the earliest attempts at building modular CDSSs, functionality was prioritized over formal correctness.

EON (**??**) switched from small, modular and independent guidelines from Arden Syntax to more complex, tightly coupled state machines-based representation to enable modeling interdependent workflows. Similarly,

	Implementation-Specification Gap	Complete Formal Semantics	Formal Analysis Tools
Arden Syntax	✓	✗	✗
DILLEMA	✓	✗	✗
GEODE-CM	✓	✗	✗
EON	✓	✗	✗
GLIF	✓	✗	✗
Asbru	✓	○	✓
PROforma	✓	✓	✗
GLARE	✓	○	○
GPROVE	✓	✓	✗
HELEN	✓	✗	✗
PRODIGY	✓	✗	✗
SAGE	✓	✗	✗
Hierarchical Plans	✗	○	✓
MDA	✗	✗	✓

Table 4.1: Comparison of Existing Approaches

GLIF (**??**) borrowed experiences from EON, GEODE-CM and Arden Syntax to facilitate modeling complex workflows. The design and implementation of GLIF was particularly influenced by the motivation to allow GLIF-based guidelines to be easily shared between medical establishments. Similary, Prodigy (**??**) tackled CDSSs challenges encountered in practice, such as diversity in data sources, and their unreliability, while SAGE (**??**) attempted to integrate with existing digital workflows as seamlessly as possible. It's important to note that these approaches demonstrated that computerized decision support systems can both (a) manage complex guidelines with interdependent concurrent tasks, and, (b) be modular and portable to accommodate hospital-specific customizations, and integrate with their EHR systems. But, system safety and correctness wasn't a challenge that these approaches were directly attempting to address. Thus, they lack a complete formal semantics, and a suite of tools accompanying analysis tools.

The need for a complete formal semantics to enable development of tools for execution and analysis of guidelines has already been recognized by several approaches. For instance, the lack of ways to systematically analyze guidelines in HELEN (**??**), and an execution engine based on ad-hoc semantics has already been recognized as a limitation of the framework. PROforma (**??**), and Asbru (**??**) attempt to address this by having formal semantics, albeit incomplete semantics, that are intended to guide tool development. The Asbru language has an incomplete, but formal SOS paper-based semantics. But, according to the authors of [39], it is insufficient for developing tools for the language. Despite its incompleteness, Asbru language's semantics has served as a calculus for deductive verification using KIV, as described **??**. Along similar lines, a formal but incomplete paper-based semantics of the PROforma language is presented in [39], with the aim of serving as a manual for developing PROforma tools. Both approaches take significant steps towards using recognizing formal semantics, and using them in some ways to establish CDSS safety. But, the support for formal analysis is very limited. Moreover, the tools aren't directly derived from the semantics, and need to be updated every time the language evolves, which can potential lead to a divergence between the official semantics, and the ones utilized by said tools.

Approaches such as GLARE (**??**) and GPROVE(**??**), also have tools that allow complex guidelines to be both modeled and analyzed, but said tools are based on ad-hoc language semantics. For instance, GLARE provides mechanisms to check for inconsistencies in the guidelines such as unintended cyclic behavior, choices without alternative options, and actions with unsatisfiable temporal constraints. Notably, GPROVE has

capabilities to enable a-posteriori compliance checking, i.e., checking whether a log generated during execution indicates adherence (or lack thereof) to the guideline. To this end, GOSpeL (the language for expressing guidelines in GPROVE) is translated to Sciff – a constraint logic programming framework. But, both GLARE and GPROVE lack formal semantics. GLARE's analysis and GPROVE's Sciff translator and execution trace generator are based on ad-hoc semantics of the underlying languages, making it possible for the analysis results to not hold on the actual programs.

Next, we consider approaches that rely on general solutions for building concurrent systems for building and analyzing CDSSs. The verification methodology defined in **??** utilizes foundational work in reasoning with knowledge bases to reason about Asbru guidelines. But, it can work with any language that utilizes a plan-based approach to represent guidelines. Similarly, the MDA-based approach relies on guidelines expressed as Statecharts in the OMG systems modeling language. The OMG Statecharts are then translated to PROMELA for verification using the SPIN, and into executable Java code for use as a CDSS. For convenience, we refer to the representation that guidelines are directly encoded in as "HCP-friendly", and the translated representation used for analysis as "analysis-friendly". For instance, for the MDA-based approach described in **??**, we refer to Statecharts as the "HCP-friendly" , and the translated PROMELA code as "analysis-friendly". Thus, for both these approaches: (a) These exists a specification-implementation gap, where the "HCP-friendly" guideline serves as the specification, and the translated PROMELA or Java code serves as the implementation. (b) the translation from the "HCP-friendly" representation to an "executable" one, or a "analysis-friendly" is typically based on an ad-hoc semantics of the "HCP-friendly" language. Thus, even though the "analysis-friendly" representation may have well-defined formal semantics, the results of analysis may not hold on the "HCP-friendly" notation. Moreover, in case of the MDA-based approach, the "HCP-friendly" notation is translated to Java for execution. But, as the Java and PROMELA translations are not proven to be equivalent, it is possible for the results of model checking through SPIN may not be applicable to the corresponding Java code.

Chapter 5

Semantics-First Approach to Clinical Decision Support

In **??**, we explained that, despite advances in medicine, mortality and costs associated with preventable medical errors (PMEs) remain unacceptably high. In **??**, we explained how systems that assist healthcare practitioners (HCPs) with situation-specific advice based on evidence-based best practice guidelines (BPGs), called clinical decision (CDSSs) can reduce both mortality and costs associated with PMEs. But, despite their potential, the uptake of such systems in practice is hindered by challenges that were introduced in **??**, and discussed in depth in **??**. In brief, the following challenges (Cs) were outlined:

C1. Absence of systematic ways of *validating content* in a *reliable*, *accessible* and *updateable* manner.

C2. Lack of *reliable*, *shareable* CDSS content that can be easily adopted across healthcare organizations and their (Information Technology) IT systems.

C3. Technical difficulties of sharing due to *need for adaptation* to diverse Electronic Health Records (EHR) systems.

C4. *Suboptimal* User Interfaces (UIs), implementation choices and workflows.

Over the years, significant progress has been made towards addressing these challenges. In **??**, we discussed how existing approaches have attempted to address said challenges, and their limitations. Specifically, in **??**, we outlined major themes that these approaches adopt to tackle these challenges. This is further illustrated by the pyramid diagram in **??**, where aforementioned themes are underlined in the pyramid's various rungs. As is typical, approaches that appear in higher rungs also have characteristics of ones below them. For example, while guidelines expressed in the Arden Syntax eliminate the specification-implementation gap by being both HCP-comprehensible and interpretable, they cannot be formally analyzed due to lack of analysis tools in the ecosystem. Asbru-based guidelines on the other hand not only eliminate the specification-implementation gap, but can also be formally analyzed using support for KIV-based verification in the Asbru ecosystem (see **??**).

As is evident in **??**, no existing approach covers the "holistic safety" rung of the pyramid. Recall from **??** that we say an approach tackles "holistic safety" if, besides support for analyzing guidelines, analysis and execution tools also have correctness guarantees. In this work, we argue that such guarantees are necessary for trustworthy CDSSs. We attempt to address "holistic safety" systematically by developing a *semantics-first approach* for building clinical decision support systems. In this context, by semantics-first we mean that:

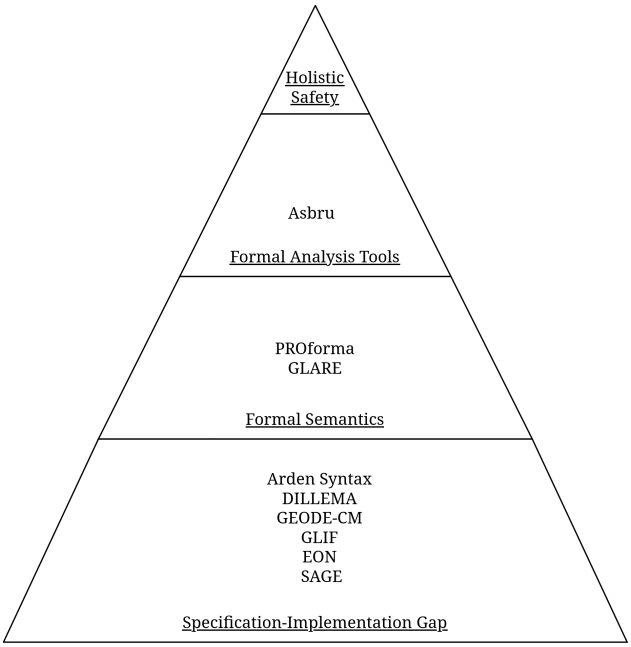


Figure 5.1: Existing DSLs for Computer Interpretable Guidelines

- The semantics of the programming language for defining said knowledge is formally defined, from which execution and analysis tools are derived in a correct by construction manner, leading to holistic safety.
- The semantics of medical knowledge are expressed accurately.

At the core of our approach is a novel domain-specific language for expressing medical knowledge called Medi$\mathbb{K}$ (pronounced Medi-Kay). By being comprehensible to domain experts in medicine, Medi$\mathbb{K}$-based computer interpretable guidelines can serve both as a guideline's non-executable HCP-comprehensible description, i.e., the specification, and its encoding in a computable medium, i.e., the implementation, thereby eliminating any specification-implementation gap.

The remainder of this chapter is structured as follows: **??** briefly describes the semantics-first philosophy. Next, **??** describes $\mathbb{K}$ – the language semantic framework that Medi$\mathbb{K}$'s are expressed in. Finally, **??** describes potential pitfalls of following the semantics-first philosophy.

5.1 Semantics-First Approach

The semantics-first approach prescribes a systematic way of developing programming languages. Instead of implementing tools for a language, such as interpreters, compilers and model checkers in ad-hoc manner, the approach states that the first step in developing said tools must be to formally define the language's semantics. As show in **??**, once defined, all tools for the language can then be automatically derived from the semantics. Moreover, since the tools utilize the semantics, they are, by definition, correct-by-construction.

While following the semantics-first philosophy might seem like an obvious choice in language design, its adoption in practice is far from ideal. Conventional practice in the programming language and formal methods community is still to develop analysis and execution tools for each programming language from scratch [140], as illustrated in **??** from [140]. But, this approach has several disadvantages:

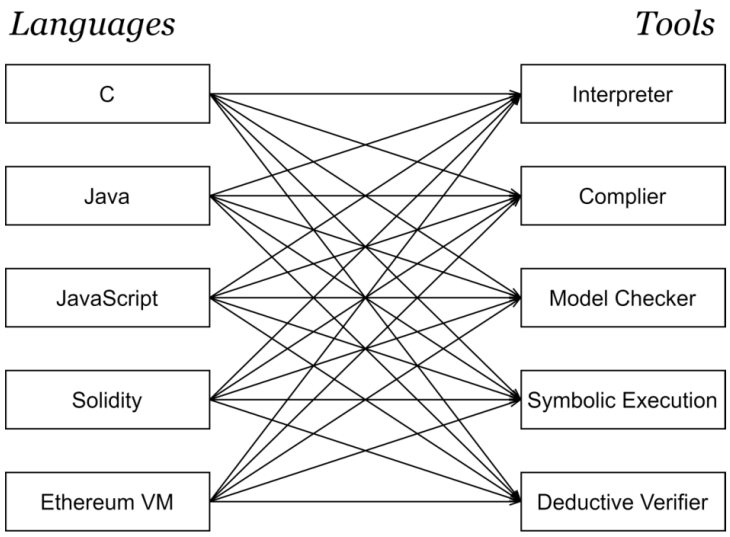


Figure 5.2: State-of-Art in Programming Language Design

- Implementing tools that perform the same function for different languages incurs unnecessary development and maintenance cost. As shown in **??**, if there are l languages, where each has t tools, then a total of $l \times t$ tools have to be developed and maintained over time.
- Tools are often based on informal descriptions of language semantics, leaving developers to extrapolate finer details of the language's semantics, leading to inconsistencies. For instance, in [141], it was found that ECMAScript 5.1-compliant JavaScript engines in mainstream web browsers behaved differently from each other for certain complex JavaScript programs.
- As newer versions of a language are introduced, each tool for the language has to be updated to ensure support for the latest version. This again results in duplicated work.

5.1.1 Why build CDSSs using Semantics-First?

In section **??**, we described benefits of using the semantics-first approach for developing regular programming language. But, these differences become starker when semantics-first is compared against the conventional approach shown in **??** in context of domain-specific language for expressing medical guidelines. Specifically, as such a language will be utilized in safety-critical settings, it is vital that the language:

- Has an *unambiguous*, *formal* semantics that can serve as a reference for developing tool support for it. This is necessary to ensure that tools are free of behavioral inconsistencies due to ambiguities in the semantics.
- Is supported by a rich formal analysis tools that can be used to analyze programs. Implementing such tools from scratch would require significant effort.
- Can evolve quick to incorporate (i) lessons from expressing medical guidelines in it, and, (ii) HCP feedback, specifically regarding comprehensibility. This can be challenging when using the approach shown in **??**, as every change to the language's semantics would require corresponding changes to all

relevant tools, and additional effort to maintain different versions, making the development process extremely tedious.

5.2 The $\mathbb{K}$ Framework

In this section, we introduce $\mathbb{K}$: a rewrite-based executable semantics in which programming languages can be defined through configurations and rules [41]. Once the semantics of a programming language has been defined, $\mathbb{K}$ automatically generates all tools depicted in **??**, such as an interpreter, compiler, model-checker and deductive verifier for the language. $\mathbb{K}$ has been successfully utilized to formalize semantics of large real-world languages, such as C [42], Java [43] and Javascript [141], and analyze non-trivial programs [45], [46].

The remainder of this section introduces relevant features of $\mathbb{K}$ by describing the $\mathbb{K}$ semantics of an example language called Imp. This introduction to $\mathbb{K}$ provides necessary background for upcoming chapters that discuss the Medi$\mathbb{K}$ DSL through the use of $\mathbb{K}$ notation and concepts.

5.2.1 Defining Languages in $\mathbb{K}$

A typical $\mathbb{K}$ definition of a language consists of the following components:

- Syntax: Defined in BNF-like notation, and utilized by $\mathbb{K}$ to generate a parser for the language.
- Configuration: Organizes the program execution state into units called *cells* that may be nested.
- Rules: Operate over configuration segments and define program evolution via rewrites.

We now discuss aforementioned components in the context of the Imp language. Imp is a simple imperative programming language inspired by C and Java that supports arithmetic and boolean expressions and statements such as variable declaration and assignment, branching (`if`) and looping (`while`).

?? and **??**, define syntax and semantics of Imp respectively. $\mathbb{K}$ code must be place inside an organizational unit called a `module` that has a name, and can import other `module`(s). For example, `module IMP-SYNTAX ... endmodule` between **??** and **??** of **??** defines a $\mathbb{K}$ `module` named `IMP-SYNTAX` containing Imp's grammar. $\mathbb{K}$ provides builtin support for domains such as natural numbers, integers, booleans and program identifiers under a `module` named `DOMAINS`. **??** `imports` the syntax definition of $\mathbb{K}$'s `DOMAINS` module to enable parsing integers and booleans in Imp programs.

Syntax in $\mathbb{K}$ is defined using BNF-like notation; terminals are enclosed in quotes, and non-terminals begin with an uppercase. For example, consider the declaration of Imp arithmetic expressions between **??** and **??**. On **??** $\mathbb{K}$'s builtin support for integers (`Int`) and program identifiers (`Id`) is used to support arithmetic expressions over program variables.

Beyond simply defining the syntax, $\mathbb{K}$ also allows assigning semantics to constructs through the use of *attributes* specified as a comma-separated list inside square brackets (`[..]`) placed immediately after the BNF production. For example, On **??** and **??**, the attribute `left` is used to declare the corresponding operator as left associative. The `strict` attribute is used to assign *evaluation strategies*. Note its use on **??** and **??**, signifying that both operand sub-expressions must be completely evaluated before the before the corresponding operator (`/`, `+`) is evaluated. Note that the order in which the arguments are evaluated is non-deterministic, and, all operands will be evaluated before the operator is evaluated. To enforce a particular order, or, to choose a subset of the operands, a list of operand positions specifying the desired order can be

```
module IMP-SYNTAX
  imports DOMAINS-SYNTAX
  syntax AExp  ::= Int | Id
                 | "-" Int
                 | AExp "/" AExp              [left, strict]
                 | "(" AExp ")"               [bracket]
                 > AExp "+" AExp              [left, strict]
  syntax BExp  ::= Bool
                 | AExp "<=" AExp             [seqstrict]
                 | "!" BExp                   [strict]
                 | "(" BExp ")"               [bracket]
                 > BExp "&&" BExp             [left, strict(1)]
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"
  syntax Stmt  ::= Block
                 | Id "=" AExp ";"            [strict(2)]
                 | "if" "(" BExp ")"
                   Block "else" Block         [strict(1)]
                 | "while" "(" BExp ")" Block
                 > Stmt Stmt                  [left]
  syntax Pgm ::= "int" Ids ";" Stmt
  syntax Ids ::= List{Id,","}
endmodule
```

Listing 5.1: Imp Syntax in $\mathbb{K}$

supplied to `strict`. For example, `strict(2, 1)` would make $\mathbb{K}$ evaluate the second argument before the first. This is particularly useful for defining constructs like short-curcuit (`&&`) boolean expression on **??**, where `strict(1)` ensures that the left operand is evaluated first, allowing the right to only be evaluated if the left evaluates to `true`. Similarly, for an assignment statement on **??** the `strict(2)` indicates that the second argument, i.e., the expression to the right of the `=` sign must be evaluated before the identifier on the left of the `=` is updated.

$\mathbb{K}$ also allows operator precedence to be specified as a part of the syntax definition. On **??**, the `>` signifies that all preceding productions have higher precedence, i.e., bind tighter, than the production for addition (`+`). Similarly, **??** defines statement composition. Thus, preceding `Stmt` productions (**??**-**??**), that define blocks (`{..}`) and standalone statements such as variable assignment and while loop have higher precedence, indicated by `>` on **??**.

On **??**, an Imp program is defined to start with a list of program variable declarations (`"int" Ids ";"`) followed by other statements. Note the definition of a list of identifier (`Ids`) on **??**. In $\mathbb{K}$, `List{...}` is used to define syntactic-lists, where the first argument is the production of list elements, and the second the list de-limiter. Thus, `List{Id, ","}` defines a comma-separated list of program identifiers.

Once the *syntax* has been defined, $\mathbb{K}$ can utilize it to generate a parser, which can be used to generate abstract syntax trees (ASTs) for programs. The $\mathbb{K}$ grammar declares an order-sorted algebra over AST terms, where the non-terminals are the sorts. The program's AST forms a part of the larger program state, over which semantics are defined through rules. The $\mathbb{K}$ semantics of any language has two components:

- A *Configuration* that organizes the state into units called *cells*, that may be nested.
- *Rules* that operate over *configuration*-segments to desribe evolution of state during execution.

For Imp, **??** has aforementioned components inside `module IMP`. Note that the syntax and semantics exist in

```
module IMP
  imports IMP-SYNTAX
  imports DOMAINS
  syntax KResult ::= Int | Bool

  configuration <T>
                  <k> $PGM:Pgm </k>
                  <state> .Map </state>
                </T>

// AExp
  rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
  rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
  rule I1 + I2 => I1 +Int I2
  rule - I1 => 0 -Int I1
// BExp
  rule I1 <= I2 => I1 <=Int I2
  rule ! T => notBool T
  rule true && B => B
  rule false && _ => false
// Block
  rule {} => .   [structural]
  rule {S} => S  [structural]
// Stmt
  rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
  rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
  rule if (true)  S else _ => S
  rule if (false) _ else S => S
  rule while (B) S => if (B) {S while (B) S} else {}  [structural]
// Pgm
  rule <k> int (X,Xs => Xs);_ </k> <state> Rho:Map (.Map => X|->0) </state>
    requires notBool (X in keys(Rho))
  rule int .Ids; S => S  [structural]

endmodule
```

Listing 5.2: $\mathbb{K}$ Semantics of Imp

separate modules, where the semantics module `imports` the syntax module. This is by convention, and has the following advantages:

a) Rules can operate directly over the language's syntax, making them easier to specify and comprehend.

b) With the syntax and semantics residing in separate modules, $\mathbb{K}$ can be instructed to use only the syntax module for parsing programs. This allows users to define additional syntactic constructs in the semantics module for use in rules. If a combined syntax and semantics module is used instead, any constructs defined solely for use in rules would unintentionally become part of the language's syntax, and be accepted by the parser.

Configurations

The configuration is defined using the keyword `configuration`, followed by an unordered list of *cells*, which may be nested and are specified using an XML-like notation. For instance, `<foo> <bar> ... </bar> </foo>` corresponds to $\mathbb{K}$ *cells* named `foo` and `bar` respectively, where `bar` is nested under `foo`. Imp's configuration is

defined between **??** and **??**, and consists of a top-level cell `<T>` containing cells `<k>` and `<state>` on **??** and **??** respectively. In $\mathbb{K}$, the `<k>` cell typically contains the AST of the executing program, indicated by its contents `$PGM:Pgm`. During execution, $\mathbb{K}$ will attempt to parse the program being executed using the production `Pgm` defined on **??** of **??**. If parsing is successful, $\mathbb{K}$ will replace `$PGM` with the parsed AST. The `<state>` cell, as the name suggests, will hold a map of program identifiers and values they acquire during execution. Said map is initially empty, denoted by `.Map`. The dot (`.`) in $\mathbb{K}$ denotes *nothing* or *empty*. Thus, `.Map` denotes an empty map.

Rules

$\mathbb{K}$ *rules* operate over configuration segments and define evolution of program state. When specifying a rule $\mathbb{K}$, only relevant parts of the configuration need to be mentioned—$\mathbb{K}$ completes the rest of the configuration through a mechanism called *configuration abstraction*. This allows rules to be concise, enhancing readability and making them easier to write. For example, consider the rule for addition on **??**. Simply put, the rule specifies that if the term at the top of the `<k>` cell is an addition expression, then it should be *rewritten* to addition in the integer domain `+Int`. But, even though the rule is intended to simplify the top of the `<k>` cell, no cells are explicitly mentioned. This is because:

- If no cell is mentioned, $\mathbb{K}$ assumes that rule applies at the top of the $\mathbb{K}$ cell.
- All other parts of the configuration are assumed to remain unchanged.

Now we describe rules in greater depth. A rule begins with the keyword `rule` and is a statement of the form $\varphi \Rightarrow \psi$, where φ, ψ are *patterns* over configuration terms and $\mathbb{K}$ variables. We say φ is the LHS and ψ is the RHS of the rule. We define a *substitution* θ to be a map from $\mathbb{K}$-variables to terms. Given pattern φ and *substitution* θ, we say $\varphi\theta$ is the pattern obtained by replacing every variable v in φ with $\theta(v)$. We say pattern φ matches term τ iff there exists a substitution θ s.t. $\tau = \varphi\theta$. During execution, if the current configuration term τ *matches* the LHS φ of rule $\varphi \Rightarrow \psi$ with substitution θ, then C is rewritten to $\psi\theta$. For example, the addition rule on **??**. $\mathbb{K}$-variables always begin with an uppercase, and may be suffixed with `:S`, where `S` is the variable's sort. In the case of the addition rule, $\mathbb{K}$ can *infer* that the sort of `I1` and `I2` is `Int`, as operands of the buitin operator `+Int` can only be integers (`+Int` is addition in the domain of integers). Thus, the LHS `I1 + I2` *match* term `2 + 3` (of sort `AExp`) with substitution $\theta =$`(I1` $\mapsto$ `2, I2` $\mapsto$ `3)` and rewrite it `2 +Int 3`, i.e. `5`, as `+Int` is $\mathbb{K}$-builtin for integer addition.

We now discuss execution of an entire program to introduce K nuances relevant to later chapter. Imp's grammar dictates that an Imp program (denoted by production `Pgm` on **??** of **??**) must declare all variables at the start of execution. Consider the simple program `int x; x = 2 + 3;`. At the start of execution, $\mathbb{K}$ replaces the `$PGM` in the configuration declaration with the program's AST. Thus, we get the following initial configuration:

```
<T>
  <k> int x, .Ids; x = 2 + 3; </k>
  <state> .Map </state>
</T>
```

where `.Ids` is the identity element for the user-defined list of program identifiers, and `.Map` is the map identity from the initial configuration declaration on **??** Now the rule for variable declaration on **??** can match with substition `(X` $\mapsto$ `x, Xs` $\mapsto$ `.Ids, Rho` $\mapsto$ `.Map)` to rewrite the configuration to:

```
<T>
  <k> int .Ids; x = 2 + 3; </k>
  <state> x |-> 0 </state>
</T>
```

Note the following conveniences that $\mathbb{K}$ offers to make writing rules easier:

i) The use of parenthesis limi scope of the rewrite to the list of program identifiers. Localized rewriting reduces redundancy as only relevant parts of a term have to be mentioned on the RHS of the rule.

ii) The underscore `_` following `int (X, Xs => Xs);` on **??** is an *anonymous* variable that matches the remainder to the program. Since it's not anywhere on the RHS, there is no need to provide an explicit variable name.

Also that the rule has a side condition, expressed through the keyword `requires` on **??**. This forces the rule to only apply if the program identifier has not already been declared. For instance, a program that starts with `int x, x;` will not execute to completion. Next, the rule on **??** will eliminate `int .Ids;`, leaving the following configuration after the rewrite:

```
<T>
  <k> x = 2 + 3; </k>
  <state> x |-> 0 </state>
</T>
```

Also note that the rule does not explicitly mention the cell on which it is intended to operate. In $\mathbb{K}$, a rule that doesn't mention any cells is implicitly assumed to apply on top of the `<k>` cell. As the `<k>` cell typically contains the program being executed, not having to explicitly mention the cell allows the rule to only show how constructs in the language affect execution. For instance, the rules on **??** and **??** depict that for an `if(condition) {...}` statement, if the `condition` is true, the `if` block is taken, else whatever comes after is taken.

Next, consider the assignment rule on **??** of **??**. The rule uses two variables: `X` and `I` with sorts `Id` and `Int` respectively. Since the `<k>` cell in our running example has `x = 2 + 3;`, we expect the assignment rule to apply at some point and update the `<state>` accordingly. But, the rule cannot apply as the RHS of the assignment, i.e., the expression `2 + 3`, is not of sort `Int`, which needs to be evaluated first. This in $\mathbb{K}$ is enabled by evaluation strategies. Recall that the attribute assignment statement has the attribute `strict(2)` (**??** of **??**), denoting that the second argument of the construct must be evaluated first. Thus, $\mathbb{K}$ will *heat*, or pull-out the second argument for evaluation. This results in the configuration:

```
<T>
  <k> 2 + 3 ~> x = []; </k>
  <state> x |-> 0 </state>
</T>
```

In $\mathbb{K}$, `~>` means *followed-by*, i.e., the evaluation of `2 + 3` must occur *before* the evaluation of `x = [];`. `[]` denotes a *hole* left in place of the argument that was *heated*. `<k> 2 + 3 ~> x = []; </k>` is re-written to `<k> 5 ~> x = []; </k>` by an application of the arithmetic addition rule on **??**. On **??**, we specify any term of sort `Int` to be a `KResult`. This signifies that the term can no longer be evaluated, and can be *cooled* or plugged-back into its corresponding hole, resulting in the configuration:

```
<T>
  <k> x = 5; </k>
  <state> x |-> 0 </state>
</T>
```

Now the LHS of the assignment rule can *match* with substitution θ `= (X` $\mapsto$ `x, I` $\mapsto$ `5)`, to update the value of `x` in the `<state>` cell to `5`, resulting in the final configuration:

```
<T>
  <k> . </k>
  <state> x |-> 5 </state>
</T>
```

Note the use of `...` in the assignment rule. In $\mathbb{K}$, the `...` is used to refer to parts of the term not relevant to the rule. For instance, in the `<state>` cell, it signifies that there may be other other identifier-value pairs in the map, but we are only concerned with the the binding that has the program identifier on the LHS of the rule. Similarly, the `...` after the assignment statement in the `<k>` cell signifies that there may be other statements to evaluate, but they are irrelevant to the assignment rule. To better illustrate the significance of the `...` in $\mathbb{K}$, we slightly modify the imp program to have two simple statements after the variable declaration instead of one. Assume the program has the form `int x; x = 5; x = x + 2`. variable declaration, we would have:

```
<T>
  <k> x = 5; x = x + 2; </k>
  <state> x |-> 0 </state>
</T>
```

Now, we would expect the assignment rule from **??** of **??** to apply, since the RHS of the assignment `x = 5` doesn't need any further evaluation via heating and cooling. But, the rule cannot apply, as the top of the `<k>` cell is a composite statement as dictated by the production on **??** of **??**, and the assignment rule can only match a standalone statement. Thus, the rule on **??** will apply with substitution θ `= (S` $\mapsto$ `x = 2 + 3;, Ss` $\mapsto$ `x = x + 2;)` to give us the following configuration:

```
<T>
  <k> x = 5; ~> x = x + 2; </k>
  <state> x |-> 0 </state>
</T>
```

As mentioned before, since `~>` means *followed-by*, $\mathbb{K}$ will evaluate `x = 2 + 3;` *followed-by* `x = x + 2;`. Now, the assignment rule on **??** can apply, where `...` after `x = x + 2;` in the `<k>` abstracts away the remaining computation, which in our example is `~> x = x + 2;`. The rule can then rewrite the assignment `x = 5;` to nothing (denoted by `.`), which leads to the configuration:

```
<T>
  <k> x = x + 2; </k>
  <state> x |-> 5 </state>
</T>
```

We had earlier mentioned that, in $\mathbb{K}$, any rule that doesn't mention a cell by default, is explicity assumed to operate on top of the `<k>` cell. We can now explain this in greater detail. Essentially, $\mathbb{K}$ considers any rule $\varphi \Rightarrow \psi$ where φ, ψ don't mention any cells equivalent `<k>` $(\varphi \Rightarrow \psi)$ ... `</k>`. This both simplifies the process of specifying semantics in $\mathbb{K}$, and, greatly improves readability of $\mathbb{K}$ definition. Moreover, the ability to use strictness atributes further reduces the need to write rules to specify evaluation strategies, enabling the semantics to serve as language specifications that are both concise and mathematically precise. This is demonstrated well by the example Imp defintion as:

- Only the rules for variable lookup (**??**), variable declaration (**??**) and variable assignment (**??**) explicitly mention cells. All other rules use terms *only involving* constructs whose semantics they define. For instance, the semantics of short curcuit boolean and (`&&`) is concisely captured by two rules depicting: (a) the need to evaluate the second argument only if the first is `true`, and, (b) discarding the second argument when the first is `false` (see **??** and **??** of **??**). The `strict(1)` attribute on **??** of **??** eliminates the need to write additional rules to ensure the first argument is already evaluated when the rules for `&&` apply.
- The entire executable semantics is fewer than 100 lines, indicating $\mathbb{K}$'s suitability for semantics-first language design.

Note the difference between program identifiers and $\mathbb{K}$ variables. While program variables are simply terms belonging to sort `Id`, $\mathbb{K}$ variables have logical meaning. If multiple rules can match the configuration term, then one rule is non-deterministically chosen. Execution is a sequence of rule applications that continues until no rule can match the configuration. Since in the running example the $\mathbb{K}$ the `<k>`-cell becomes empty after all statement are evaluated, indicating successful completion.

5.2.2 Matching Logic: Theoretical Foundations of $\mathbb{K}$

In **??**, we discussed how the need for a formal semantics for DSLs for expressing computer interpretable guideline has already been recognized. One of the motivations mentioned is the ability to utilize semantics for formal methods-based reasoning and analysis. In this section, we briefly introduce the theoretical foundations of $\mathbb{K}$ that provide a sound framework for formal reasoning.

The semantics of a $\mathbb{K}$ definition are given in Matching Logic (ML), which we briefly describe in this section. We only present concepts that are relevant to this proposal, and direct the reader to [142], [143] for a more thorough introduction to ML.

Traditionally, program verification approraches have relied on an axiomatic semantics for deduction, i.e., a proof system with *language-specific* proof rules, such as Hoare logic [144]. In $\mathbb{K}$ however, the semantics are defined in an operational style, as rewrite rules in Matching Logic, and reasoning can be performed through the language-independent proof system of ML [145]. ML as a logic was developed specifically to reason about programs compactly and modularly, and is expressive to enable other logics, such as First-Order Logic with least fixed-points, modal logic and temporal logics to be captured as ML theories [142], [143], [146], [147]. Next we present a brief introduction to Matching Logic. For a more thorough reading, we refer the reader to [146].

Syntax

The *syntax* of matching logic is parametric in two sets of variables EV and SV, called *element variables* and *set variables*. We use a lowercase letter x (uppercase letter X) to denote $x \in EV$ ($X \in SV$). A *(matching logic) signature* $\mathbb{\Sigma}$ is the tuple (EV, SV, Σ) where Σ is a set of symbols denoted $\sigma_1, \sigma_2, \ldots$. The set of $\mathbb{\Sigma}$-patterns is inductively defined by the grammar:

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

where in $\mu X.\, \varphi$ it's required that φ has no negative occurrences of X. $\varphi_1\varphi_2$ is an *application* pattern representing application of φ_1 to φ_2. All other constructs can be derived as follows:

$$\neg\varphi \equiv \varphi \to \bot \qquad \top \equiv \neg\varphi \qquad \varphi_1 \wedge \varphi_2 \equiv \neg\left(\varphi_1 \vee \varphi_2\right)$$
$$\varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \qquad \forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi \qquad \nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi\left[\neg X/X\right]$$

$\varphi\left[\psi/x\right]$ (respectively$\varphi\left[\psi/X\right]$) is the result of substituting ψ for x(respectively$X$), where bound variables in ψ are renamed to avoid capture.

Semantics

A $\mathbb{\Sigma}$-model $\mathbb{M}$ is the tuple $\left(M, \_\cdot\_, \{M_\sigma\}_{\sigma\in\Sigma}\right)$, where, M is a non-empty *carrier set*, $\_\cdot\_ : M \times M \to \mathcal{P}\left(M\right)$ is a binary function called *application*, and $M_\sigma \subseteq M$ is an *interpretation* of symbol $\sigma \in \Sigma$ in M. For brevity, we abuse $\_\cdot\_ : \mathcal{P}\left(M\right) \times \mathcal{P}\left(M\right) \to \mathcal{P}\left(M\right)$ to also mean the pointwise extension of the *application* function, i.e., $A.B = \bigcup_{a\in A, b\in B}\ a.b$ for $A, B \in \mathcal{P}\left(M\right)$.

Given a model $\mathbb{M} = \left(M, \_\cdot\_, \{M_\sigma\}_{\sigma\in\Sigma}\right)$, an M-valuation of variables $\rho : (SV \cup EV) \to (M \cup \mathcal{P}\left(M\right))$, define interpretation $\left|\varphi\right|_\rho$ of a pattern φ in M as:

- $\left|x\right|_\rho = \{\rho\left(x\right)\}$ for $x \in EV$.
- $\left|X\right|_\rho = \rho\left(x\right)$ for $X \in SV$.
- $\left|\sigma\right|_\rho = M_\sigma$ for $\sigma \in \Sigma$.
- $\left|\varphi_1\varphi_2\right|_\rho = \left|\varphi\right|_{\rho_1} . \left|\varphi\right|_{\rho_2}$
- $\left|\bot\right|_\rho = \emptyset$
- $\left|\varphi_1 \to \varphi_2\right|_\rho = M \setminus \left(\left|\varphi_1\right|_\rho \setminus \left|\varphi_2\right|_\rho\right)$.
- $\left|\exists x.\, \varphi\right|_\rho = \bigcup_{a\in M} \left|\varphi\right|_{\rho\left[a/x\right]}$, where $\rho\left[a/x\right]$ and ρ agree on all variables except x, and $\rho\left[a/x\right]\left(x\right) = a$.
- $\left|\mu X.\, \varphi\right|_\rho = \mu\mathcal{F}^\rho_{X,\varphi}$ where $\mathcal{F}^\rho_{X,\varphi} = \left|\varphi\right|_{\rho\left[A/X\right]}$ for every $A \subseteq M$, where $\rho\left[A/X\right]$ and ρ agree on all variables except X and $\rho\left[A/X\right]\left(X\right) = A$. For proof of existence of a unique least-fixed point for $\mathcal{F}^\rho_{X,\varphi}$, referred to as $\mu\mathcal{F}^\rho_{X,\varphi}$, see [143].

A pattern φ is said to hold in model $\mathbb{M}$, denoted $\mathbb{M} \vDash \varphi$ iff $\left|\varphi\right|_\rho = M$ for any $\mathbb{M}$ evaluation ρ. Similarly, set of patterns Γ holds in $\mathbb{M}$, i.e. $\mathbb{M} \vDash \Gamma$ iff $\mathbb{M} \vDash \varphi$ for all $\varphi \in \Gamma$. Finally, $\Gamma \vDash \varphi$ holds iff for any model $\mathbb{M}$, if $\mathbb{M} \vDash \Gamma$ then $\mathbb{M} \vDash \varphi$. In [148], the authors present a Hilbert-style proof system for ML that is amenable to automated fixed-point reasoning for working with inductive proofs [149]. The provability relation, denoted

$\Gamma \vdash \varphi$, is defined to mean φ can be deduced using ML's proof system utilizing Γ as additional axioms. This provability relation is *sound*, i.e., $\Gamma \vdash \varphi \implies \Gamma \vDash \varphi$.

Next, we give an intuive description of the semantics of ML. Given signature $\mathbb{\Sigma}$, a Σ-model $\mathbb{M} = \left(M, \_\cdot\_, \{M_\sigma\}_{\sigma \in \Sigma}\right)$, $\mathbb{\Sigma}$-patterns evaluate to sets of elements in M that *match* them. For instance, variable $x \in EV$ evaluates to a *singleton* containing *element*, while $X \in SV$ evalutes to some subset of M. $\bot$ evaluates to the $\emptyset$. $\varphi_1 \to \varphi_2$ is matched by all elements that match φ_2 if they also match φ_1. $\exists x.\, \varphi$ allows abstracting away irrelevant parts (i.e. x) of the pattern φ. $\varphi_1\ \varphi_2$ allows for application of a function/operation/constructor, i.e. φ_1, to its arguments, i.e. φ_2. $\mu X.\, \varphi$ evaluates to the *smallest* set X s.t. φ evaluates to X. Other patterns such as $\neg\varphi$, $\varphi_1 \vee \varphi_2$ can all be derived from the basic ones mentioned above, and have expected semantics. For instance, $\neg\varphi$ matches everything *not* matched by φ. $\varphi_1 \vee \varphi_2$ matches everything that either φ_1 or φ_2 matches. We refer the reader to [146] for a detailed discussion on the semantics of matching logic.

$\mathbb{K}$ in Matching Logic

Given a language L defined in $\mathbb{K}$, the $\mathbb{K}$ tool compiles the $\mathbb{K}$-definition of L into a matching logic theory Γ^L. Note that this compilation involves generation of additional axioms to capture $\mathbb{K}$ concepts such as definedness, order-sortedness, etc. For an in-depth discussion, see [148]. The key idea behind ML is that it is simple, yet expressive to formally capture capture the semantics of a $\mathbb{K}$ definition. Element variables can be used to *abstract* away parts of terms through existential quantification ($\exists$). Set variables range over sets and behave as modal logic propositional variables be by fixed-point quantification μ. Symbols can represent functions, constructors and modal operators in a uniform way. Together with application ($\varphi_1\varphi_2$), symbols can construct complex patterns from a simple signature. For example a symbol f applied to arguments $a_1, a_2, \ldots$ (see [143] for encoding tuples as patterns) can behave as a function. Similarly a symbol `<k>` can be used to construct `Configuration` terms. Recall from section **??** that a $\mathbb{K}$-defintion consists of rewrite rules of the form $\varphi \Rightarrow \psi$. These rules define a *transition system* over computation configurations. This is captured in ML by adding a symbol $\bullet \in \Sigma$, called *one-path-next*. Intuitively, for a given configuration γ, $\bullet\gamma$ is *matched* by all states that can transition to γ in one step [146]. We can then define $\diamond\varphi \equiv \mu X.\, \varphi \vee \bullet X$, which equals $\varphi \vee \bullet\varphi \vee \bullet\bullet\varphi \ldots$ or $\bigvee_{i \in \mathbb{N}} \left(\bullet^i \varphi\right)$. Intuitively, the $\diamond$ operator behaves exactly as its modal-logic equivalent. It matches all states that can *reach* φ in zero or more rewrite steps. Thus, a rewrite $\varphi \Rightarrow \psi$ is expressed in ML as $\varphi \to \diamond\psi$, or the set of all configurations that if match φ, can also reach ψ in zero or more steps. As mentioned earlier, ML has a sound proof system that is amenable to automated deduction. Thus, reasoning in $\mathbb{K}$ for a language L boils down to formal reasoning using its ML encoding Γ^L. Thus, reasoning that an initial configuration φ_{init} reaches a final configuration φ_{final} is showing $\Gamma^L \vdash \varphi_{\text{init}} \Rightarrow \varphi_{\text{final}}$.

5.3 Pitfalls of the Semantics-First Approach

In section **??**, we outlined advantages of the semantics-fist approach. In this section, we outline potential pitfalls of the approach, applicable specifically in context of clinical decision support systems.

5.3.1 Performance of Semantics-based Tools

$\mathbb{K}$ has been used to define several large real-world languages such as C [42], Java [43], and Javascript [141]. However, all these languages, had existing highly optimized language-specific compilers and interpreters for optimal performance, such as GCC [150], OpenJDK's compiler and java virtual machine for Java, and

javascript engines such as the V8 engine [151] that runs Chrome and Node, and firefox's SpiderMonkey [152]. Thus, even though $\mathbb{K}$-derived tools helped discover issues with either the language semantics or documentation, the $\mathbb{K}$-derived execution engines for these languages didn't replace their language-specific counterparts. This can be attributed to: (i) legacy use of existing toolchain, and, (ii) concerns of inferior performance of semantics-derived tools.

For the semantics-first approach, we intend to rely *solely rely* on semantics-derived tools for both execution and analysis. This leads to the question—are $\mathbb{K}$ semantics-derived tools performant enough to serve as execution and analysis engines in a production setting?

5.3.2 Developing a New Domain Specific Language from Scratch

$\mathbb{K}$ has been used to analyze programs in real-world languages, but these languages had existing informal language descriptions and reference test-suites to assist in development and testing of the semantics. In our case, we intend to implement the language from scratch, and come up with reference tests for it at the same time, where both the definition and the corresponding tests need to be comprehensive enough to enable independent development of tools utilizing the semantics as a standard and tests as measures of compliance.

Chapter 6

Evaluating $\mathbb{K}$

In **??**, we introduced the semantics-first approach to building systems, and how $\mathbb{K}$, a framework for defining semantics of programming languages and type systems, enables it. Specifically, **??** discusses the rationale for using the semantics-first approach over the traditional state-of-art for programming language design that leads to tools based on ad-hoc language semantics and duplicated effort. In **??**, we also touched upon some additional considerations for using $\mathbb{K}$ in the context of developing a language for developing a computer interpretretable giudeline DSL. We outlined the following concerns:

(Q1) Can the $\mathbb{K}$ generated tools be performant enough to serve as the sole tools?

(Q2) Can the $\mathbb{K}$ semantics replace an informal language description/manual?

(Q3) Can the $\mathbb{K}$-based semantics first approach be used to develop a semantics from scratch, without even an informal description of the language, and, comprehensive tests to go long with the same? Can the semantics serve as a comprehensive document to implement other tools for the language?

In the following sections attempt to answer aforementioned questions.

6.1 $\mathbb{K}$EVM: Semantics of the Ethereum Virtual Machine

The Ethereum ecosystem is a blockchain-based platform where accounts have snippets of computer code called *smart contracts*, that enable programmatic governance of transactions on the Ethereum network. All *smart contracts* on the blockchain must be specified in is an stack-based assembly-like language called the Ethereum Virtual Machine (EVM). The semantics of the Ethereum Virtual Machine are described semi-formally in a specification called the Ethereum YellowPaper [153].

Smart contracts execute when a transaction calls the account, and, among other features, these contracts can tally user votes, communicate with other contracts, store or represent tokens and digital assets, and send or receive money in cryptocurrencies, without requiring trust in any third party to faithfully execute the contract [154], [155]. The growing popularity of smart contracts has led to increased scrutiny of their security. Bugs in such contracts can be financially devastating to the involved parties. An example of such a catastrophe is the DAO attack [156], where 150 million USD worth of Ether was stolen, prompting an unprecedented hard fork of the Ethereum blockchain [157].

To address these issues in a principled manner, in [44] [1], we formalized the semantics of the EVM in $\mathbb{K}$, with the intention to utilize the semantics-generated tools for both execution and analysis of $\mathbb{K}$EVM program. Specifically, we intended to demonstrate that:

- The $\mathbb{K}$ definition can be used not only to derive semantics-based tools, but also as an *unambiguous alternative* to the informal YellowPaper semantics specification to implement new tools. In other words, the semantics themselves are *readable* to enable one to comprehend all subtleties and nuances of the language.
- The generated tools, including the interpreter, are *performant* enough for use in place of custom EVM-specific tools.
- The $\mathbb{K}$ definition is *complete*, as in, the interpreter generated from it passes all tests in the reference test suite.
- The definition is useful, i.e., the generated tools can be used for formal analysis in some meaningful way.

For a more complete reading, please see [44]. Here, we only present arguments that help answer the questions we posed at the beginning of this chapter. The $\mathbb{K}$ semantics of EVM presented in [44] has 2644 lines of non-blank and non-literate code, and passes all reference tests published by the Ethereum Foundation for testing EVM implementations. For reference, the EVM specific C++ implementation has 4588 lines of code. This measurement was taken on commit-hash ee0c6776c of `https://github.com/ethereum/cpp-ethereum` by counting non-blank lines of all `*.h` and `*.cpp` files in subdirectory `libevm`. We argue that these numbers are not atypical for implementing an interpreter for a small real-world programming language, not to mention the extra tools that $\mathbb{K}$ provides for analysis and security along the way.

In **??**, we present a performance comparison between KEVM, the Lem semantics [158], and the C++ reference implementation distributed by the Ethereum foundation. At the time of publication of [44], the Lem semantics: the only other executable formal specification of the EVM that we are aware of at the time of comparison.

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

Table 6.1: Lem EVM vs KEVM vs cpp-ethereum

All execution times are given as the full sequential CPU time (in MM:SS format) on an Intel i5-3330 processor (3GHz on 4 hardware threads) and 24 GB of RAM. The row Lem indicates a run of all the tests that the Lem semantics can run (a subset of the VMTests). The row VMStress indicates a run of all 18 stress tests in the test-suite, to compare the performance of KEVM with the C++ implementation. The row VMNormal is a run of all the non-stress tests in the test-suite (*not* the same set of tests as Lem). VMAll is the addition of the second and third rows and is included for completeness. The last three rows indicate a runs of the

[1] Joint work with E. Hildenbrandt et al.

GeneralStateTests; GSNormal are the non-stress tests, GSQuad are the stress tests, and GSAll is the addition of the two. Under the GeneralStateTests, our tools performs well except in the case of QuadraticStateTests (250 out of 22955).

As shown in the comparison, the automatically extracted interpreter for KEVM outperforms the currently available formal executable EVM semantics. $\mathbb{K}$EVM compares favorably to the C++ implementation, performing under 30 times slower on the stress tests, roughly 20 times slower on all tests, and only 11 times slower on the Lem and VMNormal tests. Note advancements to $\mathbb{K}$'s concrete execution capabilities since the publication of [44] may make the $\mathbb{K}$EVM interpreter compare even more favorably against the C++ implementation.

Next, we look at a more holistic comparison of $\mathbb{K}$EVM against related approaches, some of which are also semantics-based. We compare these approaches using the following metrics:

- **Spec.:** Suitable as a formal specification?
- **Exec.:** Executable on concrete tests?
- **Tests:** Passes the Ethereum test-suites?
- **Prover:** Serves as theorem prover for EVM?
- **Bugs:** Heuristic-based tools for finding bugs?
- **Gas:** Analyzes gas complexity of EVM programs?

Table **??** shows an overview of the results of our comparison. We briefly describe each effort and compare it to the relevant KEVM artifact. The projects fit two categories: semantic specifications and smart contract analysis tools.

6.1.1 Semantic Specifications

Ethereum YellowPaper: The YellowPaper is the official document describing the execution of the EVM [153], as well as other data, algorithms, and parameters required to build consensus-compatible EVM clients and Ethereum implementations. It cannot be tested against the conformance test-suite; instead it serves as a guide for implementations to follow. Much of the machine definition is supplied as several mutually recursive functions from machine-state to machine-state. The Yellow Paper is occasionally unclear or incomplete about the exact operational behavior of the EVM; in these cases it is often easier to simply consult one of the executable implementations.

Cpp-ethereum: Cpp-ethereum is the C++ implementation that at the time served as a de-facto semantics of the EVM [159]. The Yellow Paper and the C++ implementation were developed by the same group early in the project, so the Yellow Paper conforms mostly to the C++ implementation. In addition, the conformance test-suite is generated from the C++ implementation. This means that if the Yellow Paper and the C++ implementation disagree, the C++ implementation is favored.

Lem semantics: A Lem ([160]) implementation of EVM provides an executable semantics of EVM for doing formal verification of smart contracts [158]. Lem compiles to various interactive theorem provers, including Coq, Isabelle/HOL, and HOL4. The Lem semantics does not capture inter-contract execution precisely as it

models function calls as non-deterministic events with an external (speculated) relation dictating the "allowed non-determinism". This semantics is executable and passes all of the VMTests test-suite except for those dealing with more complicated inter-contract execution, providing high levels of confidence in its correctness.

GMS small-step specification: A small-step specification of the EVM inspired by the EtherLite semantics of [161]. The specification is non-executable, but provides a precise guide for implementers of the EVM [162].

Tool	Spec.	Exec.	Tests	Prover	Bugs	Gas
Yellow Paper	✓	✗	✗	✗	✗	✗
cpp-ethereum	✗	✓	✓	✗	✗	✗
Lem spec	✓	✓	✓	✓	✗	✗
Oyente	✗	✓	✗	✗	✓	✓
hevm	✗	✓	✗	✗	✗	✗
Manticore	✗	✓	✗	✗	✓	✓
REMIX	✗	✓	✗	✗	✓	✓
F*	✗	✓	✗	✓	✓	✗
KEVM	✓	✓	✓	✓	✗	✓

Table 6.2: Feature comparison of EVM semantics and other software quality tool efforts.

6.1.2 Smart Contract Analysis Tools

Oyente: Oyente is an EVM symbolic execution engine written in Python supporting most of the EVM, with heuristics-based drivers of the engine for bugfinding [163].

Manticore: Manticore is symbolic execution engine for virtual machines, including models for x86, x86_64, ARMv7, and EVM [164]. This tool exports a Python API for specifying programs, driving symbolic execution, and checking assertions.

REMIX REMIX is a JavaScript implementation of the EVM with a browser-based IDE for building and debugging smart contracts [165]. Some static analysis is built into the tool, allowing it to catch pre-specified classes of bugs in smart contracts.

Hevm: Hevm [166] is a Haskell implementation of EVM including on an interactive debugger mode, which allows stepping through contract execution one opcode at a time.

F* formalization of EVM: An implementation of the EVM in the F* language [167] which passes roughly half of the VMTests at the time of writing. The same paper discusses an on-paper small-step specification of the EVM as well [168].

Discussion: At the start of this chapter, we asked whether tools generated from the semantics can be performant enough to replace their language-specific counterparts. We argue that the semantics-first treatment of KEVM addresses said question. Specifically, the semantics-derived interpreter compares favorably against a language-specific C++ implementation, and other tools are able to perform as well as other language-specific tools on metrics from **??**.

We based our $\mathbb{K}$ semantics on the Yellow Paper, and found several inconsistencies that were confirmed by its developers [44]. This stems from the fact that unlike the $\mathbb{K}$ semantics, the Yellow Paper is an informal document in natural language. Often, the Yellow Paper was found to be unclear, underspecified or inconsistent with the behavior of actual implementations. As the $\mathbb{K}$ semantics are executable, all details have to considered for our interpreter to pass all tests in the reference test-suite. We utilized the developer documentation for our semantics to generate an alternative to the Yellow Paper itself, called the Jello Paper [169]. The Jello Paper has been well-received by the community, with discussions to designate it as the *canonical document* instead of the Yellow Paper. For more details, we refer the reader to section VII of [44]. Thus, it demonstrates the $\mathbb{K}$ semantics can serve as a standalone language manual instead of an informal paper-based one.

6.2 Ethereum ABI DSL

EVM lacks high level constructs including functions and explicit data types. Any invocation of an EVM program always begins with the VM's program counter set to 0. Higher level languages such as Solidity and Viper, however, have functions and types, allowing users to directly call functions in contracts without dealing with low-level EVM code. This lack of high level constructs makes writing specifications for programs at the EVM level tedious and error prone.

To address this, in [44], we introduced a DSL modeled on the Ethereum Application Binary Support Interface ABI. The Ethereum ABI [170] is a mechanism for simulating a function call at the EVM level. Contracts are written in higher level languages and call functions in other contracts. The ABI facilitates this by specifying the encoding and decoding rules for a transaction's calldata: an input field available to each transaction. The ABI is an additional protocol that higher level languages can choose to comply with to ensure compatibility with other smart contracts written in other languages. Thus, compilers for popular high-level smart contract languages often compile to ABI-compliant EVM code. As it's not an official part of the Ethereum protocol, it's not a part of the Yellow Paper.

Since most smart contracts needing formal verification are written in a high-level language like Solidity [171] that produce ABI-compliant code, we created a Domain Specific Language (DSL) for interacting with ABI-compliant compiled code. Through the use of our DSL, in [45] [2], we demonstrated that verification of complex smart contracts at the level of the EVM bytecode, was feasible. Operating at the EVM-level had many benefits, namely:

- Our methodology worked for any high-level language as long as it compiled down to ABI-compliant code.
- As our DSL is built atop the EVM semantics, updates to the semantics are automatically accommodated.

6.2.1 Verifying Contracts Using Our DSL

Specifying the ERC20 Standard in $\mathbb{K}$: The ERC20 standard [172] is a standard for implement tokens within smart contracts, essentially enabling trade of third-party tokens using the Ethereum infrastructure. As an informal document, it specifies the correctness properties that token contracts must satisfy. Unfortunately, however, it leaves several corner cases unspecified, making it less than ideal to use in the formal verification of token contracts.

[2] Joint work with Park et al.

As part of verifying ERC20 token implementations, we implemented the semantics of ERC20 in $\mathbb{K}$. Our definition clarifies what data (e.g., balances and allowances) are handled by the various ERC20 functions and the precise meaning of those functions on such data. It also clarifies the meaning of all the corner cases that the ERC20 standard omits to discuss, such as transfers to itself or transfers that result in arithmetic overflows, following the most natural implementations that aim at minimizing gas consumption.

To evaluate the effectiveness of both the EVM semantics, and the associated ABI DSL, we verified three popular smart contracts: the Vyper ERC20 token[3], the HackerGold (HKG) ERC20 token[4], and OpenZeppelin's ERC20 token[5]. Of these, the Vyper ERC20 token was written in Vyper, and the others are written in Solidity. We compiled the source code down to the EVM bytecode using each language compiler, and executed our verifier to verify that the compiled EVM bytecode satisfies the aforementioned specification. During this verification process, and found divergent behaviors across these contracts that do not conform to the ERC20 standard.

6.2.2 Discussion

Note that for a more thorough introduction to our DSL and corresponding verification efforts, we refer the reader to [44], [45]. Here, we only mention it in the context of answering the questions we posed at the start of this chapter. All previous attempts at defining semantics in $\mathbb{K}$, such as C, Java and EVM, had relied on existing informal or semi-formal semantics documentation. Moreover, existing test suites could be relied upon to *quantifiably test* the completeness of the semantics. The DSL described in this section, however, represents a break from traditional $\mathbb{K}$ development as: (a) the DSL had to be implemented from scratch without an existing language manual, and, (b) tests had to be written as the language was developed. Moreover, as discussed briefly in **??**, our DSL can be used to verify *real-world smart contracts* at the level of the EVM-bytecode.

At the beginning of this chapter (see **??**, we attempted to evaluate whether $\mathbb{K}$ would be a good-choice for a *solely* semantics-first approach to clinical decision support using the following questions:

(Q1) Can the $\mathbb{K}$ generated tools be performant enough to serve as the sole tools?

(Q2) Can the $\mathbb{K}$ semantics replace an informal language description/manual?

(Q3) Can the $\mathbb{K}$-based semantics first approach be used to develop a semantics from scratch, without even an informal description of the language, and, comprehensive tests to go long with the same? Can the semantics serve as a comprehensive document to implement other tools for the language?

By *solely*, we meant that we intend the semantics-based tools to be the *de-facto* tools for our language. In other words, the semantics-derived tools should supplant any language specific ones.

In **??**, we presented a formalization of the Ethereum Virtual Machine (EVM) in $\mathbb{K}$, called $\mathbb{K}$EVM. $\mathbb{K}$EVM was developed using an informal semantics of the the EVM described in a document known as the Yellow Paper. We demonstrated that the $\mathbb{K}$ semantics-derived interpreter not only passed all tests in EVM's conformance test suite, but also compared favorably in terms of performance to the de-facto C++ language-specific implementation, while offering stronger correctness guarantees, addressing (Q1). Moreover, we found inconsistencies in the original Yellow Paper, to which our semantics has since become a popular alternative, addressing (Q2).

[3] https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20_solidity_compatible/ERC20.vy
[4] https://github.com/ether-camp/virtual-accelerator/blob/master/contracts/StandardToken.sol
[5] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

Next, we implemented a DSL based on the Ethereum ABI to make verification of smart contracts easier. Unlike earlier $\mathbb{K}$-based efforts, the language was designed and implemented entirely using $\mathbb{K}$, without the help of any existing documentation or tests. Although it is much smaller in scope and less complex than other $\mathbb{K}$ languages (such as C or Java), our DSL could be utilized to verify real-world smart contracts against desired non-trivial specs (such as ERC20). Thus, our work demonstrated that $\mathbb{K}$ can be used to implement languages from scratch, with *performant* and *useful* semantics-derived tools that can function as the *sole* tools for the language, addressing (Q3).

Chapter 7

Separating Concerns: Modular and Safe Clinical Decision Support

In **??**, we briefly mentioned components that conceptually make up a CDSS. This chapter describes the motivations behind decomposing the system as such. First, we describe where CDSSs may loosely fit within the larger context of control theory and motivate the need for tailored architectures for CDSSs. Then we briefly identify some unique challenges of encoding medical knowledge. Finally, we talk about encoding medical knowledge directly through rewriting, and associated issues.

7.1 Control Theory and CDSSs

Control theory deals with the design and analysis of *closed-loop* systems, i.e., systems where the inputs are affected at-least in part by by outputs. Typically, such systems are characterized by:

- The *plant* represents the part of the system *to be controlled*.
- The *control* or *compensator* is the part of the system that provides *satisfactory characteristics* or *regulation* [173].

When viewed from the lens of a closed-loop system, the patient behaves as the plant, and the goal is optimizing patient outcomes [174][1]. The CDSSs collects patient data through sensor, assessments and electronic health records and recommends treatment strategies to optimize patient outcomes. But, unlike traditional closed-loop systems CDSSs have to additionally account for:

- **Patient Physiology:** While typical systems have plants whose dynamics can be mathematically described, doing so in case of CDSSs has traditionally been very challenging. This can partially be attributed to factors such as diversity among patients, age, etc. Moreover, unlike traditional systems, patient state is almost always *partially observable*. For instance, while data about a patient blood counts might be a few hours old, the heart rate might be almost real-time.
- **Absence of Direct Control:** While some medical devices (such as insulin pumps) may directly administer treatment, most complex CDSSs rely on HCPs to perform recommended treatment. But,

[1] Joint work with Song et al.

there is no guarantee of the recommendation being followed, leading to a *divergence* between expected and actual scenarios. Thus, the CDSS has to reconcile said divergences over time.

- **Customizations:** CDSSs rely on User Interfaces (UIs) for a significant part of their interaction with the physical world. The effectiveness of these interactions directly affect quality of care. To maximize effectiveness, CDSSs need to accommodate hospitals divergences in specializations and degrees of expertise, adaptations of BPGs, available pharmacy and medical devices, and User Interface (UI) design.

7.2 A Generic Treatment of Best Practice Guidelines

A DSL for representing BPGs must accommodate the diversity in BPGs. Thus, we need characteristics that our DSL must support that are common across guidelines. To arrive upon these, in [174], we attempted to come up with a convenient semi-formal *abstraction* that can describe most guidelines.

We observed that most guidelines consisted of statements that fit into the form "`if` φ `do` α". Note statements without a condition can be representated simply as `if` *true* `do` α. Thus, we can represent each statement as a tuple (φ, α). Given a BPG B, we say

$$\mathit{Statements}(B) = \{(\varphi_1, \alpha_1), (\varphi_2, \alpha_2), \ldots, (\varphi_n, \alpha_n)\}$$

to refer to all statements within B. Next we observed that:

- Statement were often required to be carried out *concurrently*.
- There may be *ordering* between statements.

To capture this, first, we define the concept of a *workflow* $W = (S_W, \leq_W)$ as a *lower semi-lattice* on a partition $S_W \subseteq \mathit{Statements}(B)$. Specifically, for any pair $((\varphi_1, \alpha_1), (\varphi_2, \alpha_2))$, there exists some $(\varphi, \alpha) \in S_W$ s.t.

$$(\varphi, \alpha) \leq_W (\varphi_1, \alpha_1) \quad \bigwedge \quad (\varphi, \alpha) \leq_W (\varphi_2, \alpha_2)$$

and there is no $(\varphi', \alpha') \in S_W$ s.t.

$$(\varphi', \alpha') \leq_W (\varphi_1, \alpha_1) \quad \bigwedge \quad (\varphi', \alpha') \leq_W (\varphi_2, \alpha_2)$$

and $(\varphi, \alpha) \leq_W (\varphi', \alpha')$.

We can then define the guideline B as the *partial order*:

$$(\{W_1, W_2, \ldots, W_n\}, \leq)$$

where, W_i is the workflow $(S_{W_i}, <_{W_i})$ and $S_{W_1} \cup S_{W_2} \cup \cdots \cup S_{W_n} = \mathit{Statements}(B)$. Note that a workflow can contain a single statement to enable partitioning $\mathit{Statements}(B)$ into disparate workflows. Thus, given a workflow $W = (S_{W_i}, <_{W_i})$ an execution $\alpha_1, \alpha_2, \ldots, \alpha_n$ is a valid execution of actions of a workflow iff:

$$(\varphi_1, \alpha_1) \leq_W (\varphi_2, \alpha_2) \leq_W \cdots \leq_W (\varphi_n, \alpha_n) \quad \text{and} \quad \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n \text{ is satisfiable.}$$

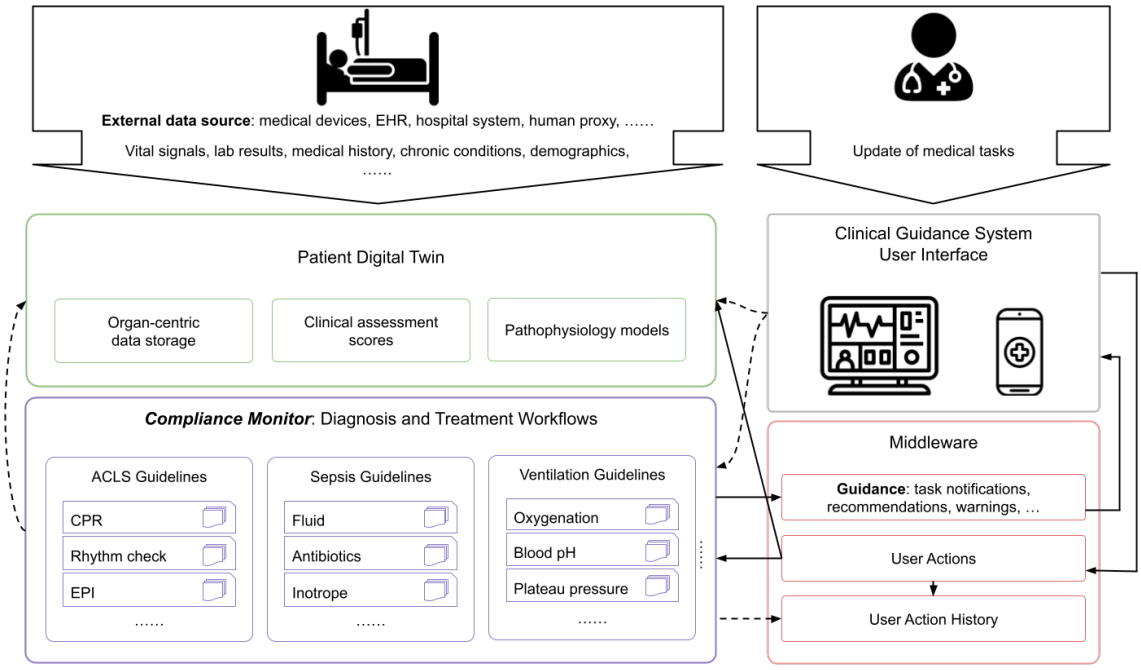


Figure 7.1: CDSSs Components

7.3 A Modular Architecture For CDSSs

This section describes the architecture for CDSSs that our work adopts. As shown in Figure **??** and **??**, we conceptually decompose CDSSs into three main components: (i) a *frontend* or a UI that forms the user-facing part of the system, (ii) a *backend* that encapsulates encoded medical knowledge, and, (iii) a *middleware* that connects the aforementioned two components. Next we describe each of these components in detail.

7.3.1 Frontend

The frontend is the user-facing component of a CDSS, and handles interactions with users by collecting user actions, delivering guidance, and displaying data and information. Ideally a CDSS frontend must be customizable according to user preferences. As these changes don't directly affect the underlying medical knowledge, they should remain *localized* to the frontend itself. Thus, the architecture shown in **??**, and many implementation from **??**, compartmentalize the frontend into a standalone component that can permit adaptions to accommodate user and hospital-specific preferences that remain isolated to the frontend itself.

Having a modular frontend is also necessitated by increasing adoption of devices such as phones and tables at medical establishments. Thus, frontends for modern CDSSs further need to accommodate diversity in devices, such as phones, portals at patient's bedside, etc., and the associated data entry methods, such as digital keyboards, voice assistants, etc. Thus, a modular, self-contained frontend is critical to solving problems with modern CDSS implementations.

7.3.2 Backend

Conceptually, the part of the system that encodes the BPG constitutes what we call the *backend*. **??** depicts these as treatment workflows. In practice, however, the backend can be intertwined with other components

in a monolithic system, be implemented as a shareable, standalone component, or be something in between. Moreover, the *computable encoding* may not resemble a traditional non-executable BPG.

As mentioned in **??**, having high-quality, guidelines with mechanism that enable systematic validation of content and ease of sharing is vital to wider adoption. This work argues that this is enabled through the use of computer-interpretable guidelines that can serve as both textual BPGs and their computer interpretable counterparts. Thus, ideally, the backend should comprise of a computer-interpretable guideline.

In **??**, we introduced existing approaches for expressing guidelines in a computer-interpretable manner. As discussed in **??**, several approaches also stress on using formal methods-based techniques to further ensure system correctness. Thus, good BPGs, and their computer-interpretable translations, require significant effort to develop and verify. This cost can be be offset by ensuring that the backend can be shared across CDSSs implementations. Thus, in this work, we build CDSSs using an architecture where the backend is a *standalone* component that be composed with setting-specific frontends to build customizable but safe CDSSs. By standalone, we specifically mean that the backend should be isolated such that any changes to any other component of the CDSSs should not percolate to the backend itself.

7.3.3 Middleware/Additional Infrastructure

?? and **??** talk about the importance of ensuring that conceptually different components are also implemented and maintained as completely *independent* entities in practice. This modularity necessitates the need for additional *glue code* to tie components in a functioning system. For example, a frontend and backend implemented completely independently of each other, in unrelated languages, require standardized communication protocols. Moreover, as CDSSs rely on a diverse spectrum of devices for data, such as patient health records, real-time monitors and sensors, additional code has to be written to connect the data to places where it's used in the backend and frontend.

In this work, we refer to any additional code outside the frontend and backend as the "middleware" or "additional infrastructure". Note that we use either term interchangeably throughout this work. Additionally, as shown in **??**, we try to *organize* patient data into *digital twins* of organ systems whenever applicable. Notionally, we use "digital twins" as virtual entities that encapsulate actual physical processes or objects, as is typical in modeling and simulation [175]. In practice, however, we recognize that such abstractions may be imprecise, due to differences between CDSSs and traditional closed-loop systems discussed at the start of this chapter. Thus, digital twins in our context may simply be simple structures that hold linked data. For example, while we may notionally refer to an abstract data type used to record a patient's heart rate and blood pressure as a digital twin of the patient's cardiovascular system, such a twin would be a very incomplete representation of the patient's actual cardiovascular system.

Chapter 8

$\mathbb{K}$-based Computer Interpretable Guidelines

In **??**, we described an architecture for building safe and modular clinical decision support systems. We split CDSSs into three separate components: (a) a frontend that facilitates interaction with HCPs, (b) a backend that serves as a computer-interpretable encoding of the BPG, and, (c) additional infrastructure that wires components together. This, and upcoming chapters, focus on building backends for CDSSs utilizing the *semantics-first* approach described in **??**. By semantics-first, we mean that: (a) the semantics of medical knowledge in the BPG is accurately captured, and, (b) the semantics of the language used to describe the BPG is formally defined.

In **??**, we briefly described characteristics of BPGs that a CIG language must accommodate. To this end, we attempted to come up with a framework that can accommodate expressing a large number of diverse BPGs. Thus, we broke BPGs into smaller statements that we organized into:

- a workflow containing statements that are executed sequentially, and,
- workflows within a guideline may be executed concurrently.

In this chapter, we describe a methodology to encode real-world BPGs as $\mathbb{K}$ definitions. Execution in $\mathbb{K}$ is inherently concurrent— if more than one $\mathbb{K}$ rule can apply, then $\mathbb{K}$ non-deterministically chooses one. Thus, we describe a way of systematically encoding medical knowledge in BPGs as $\mathbb{K}$ definitions. First, in **??**, we introduce a real world BPG that will be utilized as running-example in the rest of this chapter. Note that we intentionally choose real-word examples, instead of small toy cases, to highlight that our philosophy scales to work in the real-world. Next, **??** describes $\mathbb{K}$-ACLS, a $\mathbb{K}$-based tool to assist HCPs conform to ACLS guidelines that attempts to follow the *semantics-first* approach. Specifically, we come up with an abstract representation that captures the semantics of the BPG from **??** with enough details to enable computer-interpretation. Finally, in **??**, we describe a way to embed said abstract representation into a concrete $\mathbb{K}$ definition for execution.

8.1 Advanced Cardiovascular Life Support Guidelines (ACLS)

Advanced Cardiovascular Life Support (ACLS) are a set of BPGs periodically published by the American Heart Association (AHA) for management of life-threatening cardiac conditions that will cause of have caused

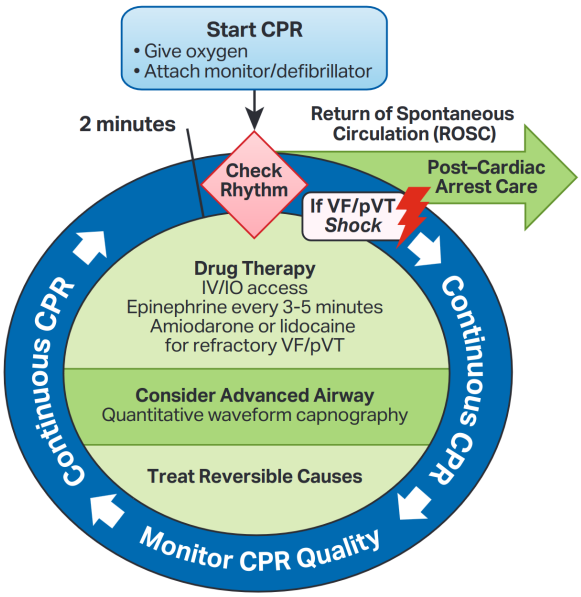


Figure 8.1: Advanced Cardiac Life Support Guidelines

cardiac arrest. The conditions that the guidelines treat range from dangerous arrhythmias, i.e., irregularities in heart's rhythm, to cardiac arrest.—a cardiac emergency where the heart stop pumping [176]. The guidelines, according to the AHA, "are based on the most current and comprehensive review of resuscitation science, systems, protocols, and education" [177].

?? shows the AHA's guidelines for advanced life support (ALS) for managing cardiac arrest in adults [178]. AHA publishes guidelines for basic life support (BLS), and pediatric counterparts of both. While BLS is meant for a first responder to provide treatment with limited resources, such as an automatic emergency defibrillator (AED), ACLS is supposed to be followed by teams of trained professionals with advanced equipment for airway management, drug delivery, etc.

Why build CDSSs for ACLS?

Cardiac arrest treatment is extremely time-critical, and outcomes become significantly worse with every passing minute. This gathering relevant data about the patient infeasible. Moreover, as the seriousness of the condition necessitates multiple, simultaneous treatments to be administered rapidly, intervention is usually executed following standardized ACLS algorithms [176].

Prompt ACLS guidelines-conformant treatment has been shown to improve patient outcomes [179]. Moreover, deviations from the guidelines in in-hospital cases has been associated with decreased rates of return of spontaneous circulation (ROSC), survival to discharge, and survival to discharge with favorable neurological outcomes [180]. Studies have also found that inadequate ACLS training can lead to poor retention, resulting in reduced ACLS conformance [181]. Thus, a CDSS that can provide situation-specific guidance about the next steps, especially in cases where HCP training might be inadequate, can potentially improve compliance, and consequently outcomes.

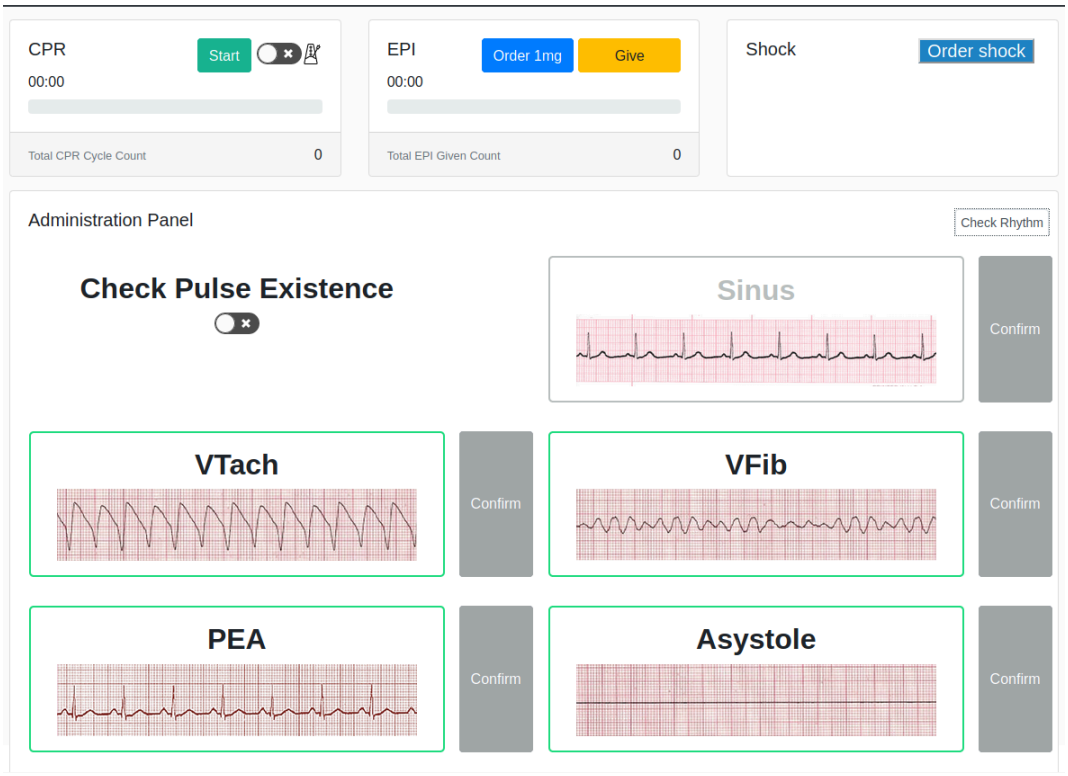


Figure 8.2: Snapshot of $\mathbb{K}$-ACLS

A Brief Overview of Advance Life Support Guidelines for Cardiac Arrest:

As we aren't concerned with intricacies of medical knowledge in the guideline, we present a brief and simplified description of the guideline-prescribed treatment. As ALS is supposed to be performed in settings where necessary equipment is available, a electrocardiogram (EKG) machine is utilized to monitor the patient's cardiac rhythm. Certain cardiac arrhythmias, characterized by cardiac rhythms referred to as *shockable*, must be treated by delivering a therapeutic electric shock [182]. As shown in **??**, treatment has several parallel workflows such as:

- Periodic monitoring of the cardiac rhythm, and in case of a *shockable* rhythm, using a defibrillator.
- Continuous cardiopulmonary resuscitation (CPR).
- Administration of vital drugs such as epinephrine every few minutes.

As **??** suggests, these workflows have to be performed rapidly and periodically. As the duration of the intervention is typically a few minutes, making optimal use of available time is critical to outcomes.

8.2 The $\mathbb{K}$-ACLS System

?? shows a snapshot of our CDSS for enabling compliance to the advanced life support guidelines described in **??**. Advanced life support is supposed to be performed by a team of HCPs, that take on roles such as a leader, CPR-provider, airway/respiratory specialist, Intravenous access (IV) and drug administration specialist, pharmacist, defibrillator attendant and members that serve as backups [176]. The leader is responsible for

coordinating treatment, our tool render support through a handheld tablet held by the leader. **??** show a snapshot of said tablet's main screen. Decision support, in the form of time-sensitive reminders, warnings when procedures are stopped prematurely, or exceed stipulated limitations, and confirmations regarding the patient rhythm's are provided on the screen through popups and progress bars. For example, when the leader instructs the team to start CPR, and presses the "Start" button under the "CPR" section of the screen. A timer displays the duration and appropriate warnings about prematurely stopping CPR, or exceeding the time for a CPR cycle are displayed.

In **??**, we described conceptual components of CDSSs, and argued in favor of developing CDSSs using independently developed and maintained codebases aligned with these components. The $\mathbb{K}$-ACLS system follows this philosophy, and has: (i) a simple *frontend* written in Javascript using React [183] that handles user interaction, and, (ii) a $\mathbb{K}$-based HTTP server that encodes the ACLS guideline and interacts with the frontend. The use of the Javascript based frontend allows our application to run on any modern web-browser. As shown in figure **??**, our frontend consists of forms and buttons that correspond to procedures in the algorithm in figure **??**. For example, the "Start" button in the CPR box results in a two minute timer corresponding to the "2 minute continuous CPR" procedure in the informal description. The frontend is simple and doesn't contain any guideline conformance related logic. Interaction with the frontend simply results in dispatch of *events* to the backend. For example, when the "Start" button is pressed, a "StartCpr" event is sent to the backend.

8.2.1 Capturing Execution-specific Details

A computer-interpretable version of a guidelines, unlike its textual counterpart, may require specification of additional details for execution. Such details can often be gathered from context in case of textual guidelines, or may be unintentionally missing. This was also discussed in context of related approaches in **??**, especially in **??**.

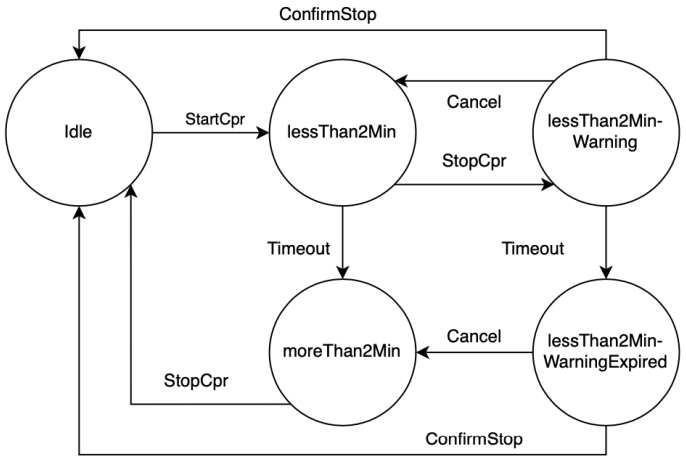
As discussed in **??**, BPGs can be notionally represented using a collection of workflows. Each workflow has steps executed sequentially, but may be concurrently executed with steps from other workflows. Consider the ACLS guidelines discussed in **??**. Several procedures, such as administering drugs, and performing CPR occur concurrently, where each procedure has a set of tasks that need to be performed sequentially. There may also be some implicit order across workflows, but we address that in later chapters.

Modeling BPGs as State Machines

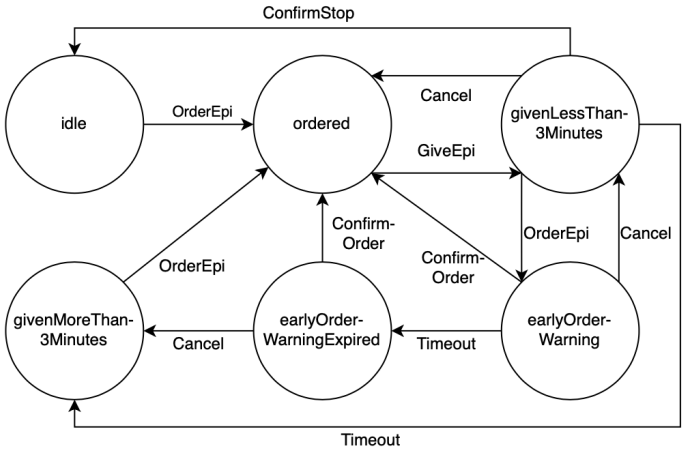
In this work, we choose to express medical logic using concurrently executing finite state machines that communicate with implicit queues for inter-machine communication. Communicating state machines (CSMs) is a well-understood model of concurrency [184], and is well-suited representing medical guidelines in a computer-interpretable format that is also HCP-friendly. We shall discuss the motivations behind using CSMs in upcoming chapters. For now, it suffices to describe how our choice addresses issues mentioned in **??**. Given a guideline represented as a collection of workflows, we describe each workflow via a state machine. The fact that CSMs can execute concurrently conveniently captures the inherent concurrency in workflows.

ACLS Workflows As State Machines

At the start of **??**, we mentioned that the *frontend* of our CDSS is a simple Javascript-based application to facilitate user interaction through buttons and popups. We can now elaborate what we mean by *simple*.



(a) CPR Machine



(b) Epinephrine Machine

Figure 8.3: Formal Machine Definitions

Button presses on the fronend correspond to *events* that trigger transitions in state machines shown in **??**. For instance, pressing the "Start" button under the CPR part of the tool shown in **??** results in a a `StartCPR` event that triggers the corresponding transition in the machine in **??**. Similarly, certain machine states result in events being dispatched to the frontend that cause popups or messages to be displayed on associated parts of the screen. Thus, the frontend itself contains no medical logic.

8.2.2 $\mathbb{K}$-ACLS backend

As described at the start of **??**, the backend receives events from the frontend, processes them and dispatches the results back to the frontend. In **??**, we described the process of details required for execution by expressing the BPG using communicating state machines. For a functioning backend through, we need to translate the abstract state machines into an executable medium. We do this by encoding the state machines in a $\mathbb{K}$ definition. Before we describing the encoding, we describe some additional challenges that we need to address in order to use $\mathbb{K}$ as a CDSS backend.

Additional Challenges

Asynchronous External Communication: Communication with external process in $\mathbb{K}$ is facilitated through the use of built-in I/O support. $\mathbb{K}$has built-in functions for many operations that map directly to their POSIX counterparts [185]. However, execution in $\mathbb{K}$ is single-threaded. Thus, to make $\mathbb{K}$ behave as an HTTP [186] server that accept messages and respond asynchronously, we need to write additional C++ initialization code and link it against $\mathbb{K}$'s LLVM backend [187].

Handling Timers: The ALS workflows shown in **??** require tasks being performed periodically. For instance, the algorithm calls for continuous CPR administration for two minutes between checking rhythm and performing defibrillation if needed. Similarly drugs like epinephrine must be administered every three minutes. $\mathbb{K}$ does not support setting timers that can interrupt the $\mathbb{K}$ process on expiration. Thus, we rely on external "timers" that, instead of interrupting the $\mathbb{K}$ process, place a `timeout` event at the head of the `<inputBuffer>` cell on expiration, which trigger corresponding transitions in state machines. For example, consider the CPR machine in **??**. When CPR is started through the press of a button on the frontend, an external two minute timer is set, and the machine transition from state `idle` to `lessThan2Min`. When the timer expires, the external timer sends a `Timeout` event to the $\mathbb{K}$, makes the CPR machine to transition from state `lessThan2Min` to `morethan2Min`.

$\mathbb{K}$-ACLS Backend

Recall from **??** that in $\mathbb{K}$ computation is described via rewrite *rules* that operate over a *configuration* that organizes data in labeled and potentially nested units called cells. **??** shows the initial configuration for the definition $\mathbb{K}$ of our state machines-based representation. Note that unlike a typical $\mathbb{K}$ definition, the initial `configuration` does not have a `<k>` cell containing a `$PGM` variable replaced by the program AST at runtime, as the $\mathbb{K}$ definition is the program itself.

```
1  configuration
2      <machines>
3          <machine multiplicity="*" type="Set">
4              <id>    .K   </id>
5              <state> .K   </state>
6              <store> .Map </store>
7          </machine>
8      </machines>
9      <inputBuffer>  .JSONs </inputBuffer>
10     <outputBuffer> .JSONs </outputBuffer>
```

Listing 8.1: Initial Configuration

Lines **??**-**??** declare a cell `<machines>` that will be used to multiple `<machine> ... </machine>` cells, declared between Lines **??** and **??**, where each cell stores all data related to a particular state machine. The `<id>` and `<state>` cells store the machine's name and active state respectively. The `<store>` cell is a map used to store any additional data. Note the following:

(i) The `<machine>` cell is declared with an attribute `multiplicity=*`, signifying that the configuration can contain multiple instances of such cells that are added or removed dynamically during execution. At the start of execution, the configuration has zero such cells.

(ii) As the number of machines to represent is known and remains constant during execution, one might ask why they are not declared as a part of the initial configuration itself. We clarify that we deliberate not to do, due to reasons we explain shortly.

Lines **??** and **??** declare cells `<inputBuffer>` and `<outputBuffer>` respectively that hold comma-separated lists of JSON objects. As their name suggested, they are used as buffer to communicate with the frontend. Incoming events from the frontend are placed at the end of the `<inputBuffer>` and outgoing events intended for the frontend are consumed from head of the `<outputBuffer>`.

Initialization

Initialization involves populating the configuration with the initial state of each state machine. For instance, **??** shows the rule that initializes the CPR Machine to start in state `idle` corresponding to the entry state of the machine in **??**.

```
1  rule <machines>
2       ( .Bag => <machine>
3                     <id> cprMachine </id>
4                     <state> idle </state>
5                     <store> "cprRunning" |-> false </store>
6                     ...
7                 </machine> )
8       ...
9      </machines>
10     <inputBuffer> "cprMachine.init" , IN => IN </inputBuffer>
```

Listing 8.2: CPR Initialization

Instead of starting `<machines>` cell that is pre-populated with `<machine>` cells from the start, we dynamically initialize machines once the corresponding frontend component is rendered and ready. For instance, when the CPR part of the tool from **??** has successfully loaded, it sends a `"cprMachine.init"`, which is added to the head of the input buffer. Lines **??**-**??** of the rule introduce a new `<machine>` cell with name `cprMachine` and initial state `idle`. The `.Bag` in $\mathbb{K}$, seen on **??**, represents the absence of a cell, which the rule rewrites to a `<machine>` cell, resulting in its addition to the configuration under the `<machines>` cell. On **??**, the $\mathbb{K}$ variable `IN` matches any list of JSON elements. Thus, `"cprMachine.init" , IN` matches any list with `"cprMachine.init"` at the head of the list (`IN` matches the tail of the list). When the rule fires, the list is rewritten to just the tail, resulting in the removal `"cprMachine.init"` event from the `<inputBuffer>` cell.

CPR Machine

The *CPR Machine* encodes the intended CPR procedure shown in figure **??**. As mentioned earlier, when the user clicks the "Start" button in the CPR pane in figure **??**, an external two minute timer is started. If the user decides to stop CPR before two minutes are up, a message is displayed to the user warning him of deviation from the intended guidelines. If the user stops CPR after two minutes, no warning message is displayed. **??** shows the rule that fires when the "Start" button in clicked. The rule does the following:

(a) Consumes the `"StartCpr"` event from the `<inputBuffer>` in **??**.

(b) Move the machine from state `idle` to state `lessThan2Min` in **??**.

(c) Sends a `startTimer` action to the frontend via the `<outputBuffer>` (encoded as JSON) in Lines **??**-**??**, which results in the start of a two minute timer in the frontend.

Note how features of $\mathbb{K}$ discussed in **??** make the definition *concise* yet *descriptive*. Specifically:

- The *configuration abstraction* mechanism enables only parts of the configuration that are not used in the rule to be easily ignored. The ... (dots) on **??** contents of cell not used in the rule to be safely ignored. Other parts of the configuration, such as the `<machines>` cell and the `<outputBuffer>` cell need not be mentioned at all.
- Localized rewriting enables the rewrite symbol `=>` to apply on deeply nested subterms. This enables de-duplication, as parts of the term that remain unchanged don't have to be specified both on the LHS and RHS of the rewrite. For instance, the `<store>` cell on **??** contains is a map that holds information about the machine, such as execution status. Recall that a map with n key-value pairs is depicted in $\mathbb{K}$ as `(`k_1 `|-` v_1`), (`k_2 `|-` v_2`),... , (`k_n `|-` v_n`)`. Localized rewriting enables us to only rewrite the value mapped to key `"cprRunning"` from any value to to `true`, and ignore other parts of the map using ... notation.

```
rule <machine>
          <id> cprMachine </id>
          <state> idle => lessThan2Min </state>
          <store> ( "cprRunning" |-> (_ => true) ) ... </store>
          ...
     </machine>
     <inputBuffer> "StartCpr" , IN => IN </inputBuffer>
     <outputBuffer>
         OUT => OUT +JSONs jsonResponse( cprMachine
                                       | lessThan2Min
                                       | startTimer( .JSONs ) )
     </outputBuffer>
```

Listing 8.3: CPR Machine in $\mathbb{K}$

Epinephrine Machine

The *Epinephrine Machine* encapsulates instructions for administering the drug Epinephrine to the patient. The corresponding machine is shown in figure **??**. The user orders the drug using the "Order 1mg" button and can administers it using the "Give" button in Epinephrine pane in figure **??**. Giving the drug results in the start of a three minute timer, during which if a new order is placed, a warning is issued.

?? encodes the transition $\mathsf{givenLessthan3Min} \xrightarrow{\mathsf{orderEpi}} \mathsf{earlyOrderWarningNotification}$ from **??**. The rule fires when it has been less than three minutes before the last dose and a fresh order for Epinephrine is put in (Lines **??**-**??**). Note that the input event here also has a payload corresponding to the dosage, depicted by a JSON object with fields `"event"` for the event's name and `"data"` for the dosage. As the guidelines dictate that Epinephrine must be administered every three minutes, ordering fresh Epinephrine is a deviation from the guidelines. Thus, an appropriate warning message is presented to the user (Lines **??**-**??**).

```
rule <machine>
       <id> epiMachine </id>
       <state> givenLessThan3Min
           => earlyOrderWarningNotification
       </state>
       <store> .Map => ("tentativeOrder" |-> ORDERED) ... </store>
     </machine>
     <inputBuffer> { "event" : "orderEpi"
                   , "data"  : [ ORDERED:Int ] } , IN
                => IN
     </inputBuffer>
     <outputBuffer> OUT
       => OUT +JSONs jsonResponse( epiMachine
                                 | earlyOrderWarningNotification
                                 | showEarlyOrderWarning( .JSONs ) )
     </outputBuffer>
```

Listing 8.4: Epi Machine in $\mathbb{K}$

Encoding Remaining Transitions

Note that we only show a single rule from the *CPR* and *Epinephrine* Machines as other rules resemble the ones shown, and have a one-to-one correspondence to transitions in the state machines in **??**. For instance, the rule in **??** encodes the transition "$\mathsf{idle} \xrightarrow{\mathsf{StartCpr}} \mathsf{lessThan2Min}$" in the CPR state machine in **??**. To encode the entire machine, along with the initialization rule, we encode every transition $u \xrightarrow{\sigma} v$ of machine $\mathcal{M}$ as a rewrite rule of the form shown in **??**

```
<machine>
  <id> M </id>
  <state> u => v </state>
  ...
  <machine>
<inputBuffer> σ, IN => IN </inputBuffer>
```

Listing 8.5: Transitions as $\mathbb{K}$-Rules

Shock Machine

We now desribe the *Shock Machine*. Intuitively, this machine ensures: (a) shocks are only administered if the rhythm is shockable, (b) CPR is administered every two minutes, and, (c) drugs (such as Epinephrine) are periodically administered. In case the of deviation appropriate warnings are issued. The *Shock* machine differs from other machines, as its transitions depend on the status of both the *CPR* and *Epinephrine* machines.

?? shows a rule from the *Shock* machine. The rule fires when the rhythm is *shockable*, and the user wants to administer a shock. Additionally, CPR hasn't been administered (**??**), but Epinephrine has been administered (**??**). In such a situation, the user is informed that it is safe to administer a shock, which should be followed by two minutes of CPR (Lines **??**-**??**). However, it is not safe to assume that the rhythm would remain *shockable* for subsequent shocks. Thus, the machine moves from state `shockableRhythm` to `checkRhythm`. If the user tries to administer another shock, a prompt to confirm the rhythm will be displayed to ensure a shock is not administered in case of an *unshockable* rhythm.

```
rule <machine>
       <id> shockMachine </id>
       <state>
         shockableRhythm => checkRhythm
       </state> ...
     </machine>
     <machine>
       <id> cprMachine </id>
       <store> ( "cprRunning" |-> false ) ... </store> ...
     </machine>
     <machine>
       <id> epiMachine </id>
       <store> ( "epiGiven" |-> true ) ... </store> ...
     </machine>
     <inputBuffer> "administerShock" , IN => IN </inputBuffer>
     <outputBuffer> OUT
     => OUT +JSONs jsonResponse( shockMachine
                               | checkRhythm
                               | confirmShock("CPR for
                                              2 Minutes" ) )
     </outputBuffer>
```

Listing 8.6: Shock Machine in $\mathbb{K}$

We draw the user's attention to the *succinctness* and *simplicity* of the rule. Despite expressing the interaction between three different machines, $\mathbb{K}$'s configuration abstraction mechanism ensures that only relevant parts of each machine are mentioned, making the transition *comprehensible*.

Chapter 9

MediK: Towards Safe Guidelines-based CDSSs

In **??**, we discussed the process of encoding best practice guidelines (BPGs) as K definitions to describe a systematic way of building CDSSs. Specifically, we described the process of encoding medical knowledge in guidelines notionally via workflows that can be expressed abstractly as concurrently executing finite state machines that communicate via passing messages. Next, we described the process of expressing state machines as K definition, where K features such as configuration abstraction and local rewriting enable *conciseness*. We then combined the K definition with a simple javascript-based frontend to develop a CDSS for assisting healthcare practitioners (HCPs) follow the Advanced Life Support (ALS) guidelines for managing cardiac arrest published by the American Heart Association.

In **??**, we described that wider CDSS adoption is incumbent on a having a systematic way of developing guidelines with validated content. The semantics-first approach we described in **??** attempts to enable such a way. It dictates that: (i) the semantics of the language be formally defined, from which tools are derived from a correct-by-construction fashion, and, (ii) ensuring that semantics of medical knowledge is accurately described. As is typically the case, the K-ACLS system from **??** through collaboration between HCPs and computer scientists. While the K-based representation of the ALS guideline was concise, it was not easily comprehensible to HCPs, or to other software engineers without prior experience of using K. This chapter addresses limitations of our work from **??**. Specifically, we describe MediK (pronounced "Medi-kay" [1]) a novel domain-specific language (DSL) for expressing medical knowledge that is designed from the ground-up with HCP-comprehensibility in mind. As the name suggest, MediK has a formal K semantics, from which all tools for it are derived, including its interpreter. MediK has been used to implement a complex CDSS for management of sepsis in pediatric cases that has multiple concurrent workflows. Our MediK-based system has been shown to satisfy desired safety properties, and to the best of our knowledge, is the first such system with formal safety guarantees.

The rest of this chapter is organized as follows. In **??** we recall the example BPG from section **??** for management of sepsis, to illustrate common requirements that DSL for modeling BPGs should satisfy. In **??**, we introduce the MediK DSL, and illustrate how it has been specifically designed to address said requirements. In **??**, we evaluate the effectiveness of our approach by utilizing it to build a CDSSs intended for real-world use. In section **??**, we discuss how MediK builds on existing approaches in **??** to advance the tate-of-art in

[1] MediK is a portmanteau of Medicine and K

addressing challenges from **??**.

9.1 Pediatric Sepsis Management BPG

In **??**, we presented a best practice guideline for management of sepsis in pediatric cases. We briefly describe the guidelines here. Recall that sepsis is life-threatening condition caused by the body's extreme response to an infection, and is a major cause of morbidity and mortality in children. Evidence indicates that timely identification and prompt treatment with antibiotics and intravenous (IV) fluids is *vital* for avoiding adverse outcomes [53], [54]. The BPG has several concurrent workflows with inter-workflow dependencies, making it suitable to study common characteristics of BPGs, which we recall here. Specifically, BPGs:

- 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 antibitiotics.
- Often specified in a *flowchart-like* notation. See [47] and [48] 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.

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.

9.2 Design Considerations

This section introduces the MediK DSL through its K syntax and semantics. We developed MediK to realize the semantics-first approach discussed in **??**. Thus, it has:

- A formally defined, unambiguous semantics.
- A correct-by-construction interpreter derived from the semantics. As discussed in **??**,
- A comprehensive suite of formal program analysis tools.
- The ability to quickly adapt physician feedback, as only the semantics need to be changed an all tools evolve automatically.

The remainder of this section introduces the MediK DSL, and describes how it's designed to accommodate characteristics of BPGs discussed in **??**.

9.2.1 HCP Comprehensibility

Our approach stresses that HCPs must be able to comprehend the guideline, and if necessary, making changes to on their own. As discussed in **??**, this has several hurdles, chief among which is the fact that HCPs are typically not trained to understand conventionally programming languages. Thus, HCPs need to work collaboratively to translate the BPG into a computable medium. In essence, the BPG serves as a functional specification for implementing the CDSS. But, this may lead to a gap in the HCPs' understanding of the system, and the actual behavior. To address this, we designed MediK s.t. encoded guidelines resemble their physical, non-executable counterparts, with the intention that familiarity with non-executable guidelines would also translate to computable MediK ones.

Recall from **??** that BPGs typically involve concurrent workflows, often expressed using a flowchart-like notation that may involve inter-workflow interactions. In **??**, we discussed the merits and suitability of concurrently executing state machines for modeling medical guidelines. Thus, we looked at state-of-art languages for modeling large concurrent systems using state machines, such as P [49], but made adaptions to make expressing BPGs easier.

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. Recall from **??**, 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 program in the language. We describe MediK's syntax in depth in Section **??**. 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 Sections **??** and **??**[2]

```
[init] machine <IDENTIFIER>
  receives <IDENTIFIER_LIST> {
  vars <IDENTIFIER_LIST>;

  [init] state <IDENTIFIER> {
    entry [(IDENTIFIER_LIST)] {
      <STMT> // entry block
    }
    on <IDENTIFIER> [(IDENTIFIER_LIST)] do {
      <STMT> // event handler
    }
  }
}
```

Listing 9.1: Skeleton of a MediK Machine

[2]The complete executable semantics is available at [188].

9.3 MediK Syntax

We use the skeleton of a MediK machine in **??**, 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.

A MediK program consists of a set of machine definitions, where a machine definition consists of:

- (Lines **??**-**??**) The keyword `machine`, followed by the name and a comma-separated list of identifiers signifying events that it `receives` via the broadcast mechanism. One state in every machine, and one machine in a program can be prefixed with the keyword `init`. On execution, an implicit instance of this machine the created, and the `entry` block of initial state executed.
- (**??**) A set of instance variables.
- (Lines **??**-**??**) A set of state declarations. Each state has a name, an optional entry block (Lines **??**-**??**), and a set of event handlers (Lines **??**-**??**). The entry block begins with the keyword `entry`, and may contain a list of variables that are bound to values when an instance enters the state during execution. One state in the machine may be prefixed with `init`, specifying the initial state that an instance starts execution in.
- Event handlers begin with `on` followed by the event name, an optional list of variables bound to data in

```
1  syntax StandaloneExp
2                  ::= "new" Id "(" Exps ")"                        [strict(2)]
3                  | "createFromInterface" "(" Id "," String ")" [strict(2)]
4                  | Id "(" Exps ")"                                [strict(2)]
5  syntax Exp ::= Id
6              | Val
7              | Rat
8              | FloatLiteral
9              | "this"
10             | UndefExp
11             | "obtainFrom" "(" Exp "," Exp ")" [seqstrict]
12             | "(" Exp ")"                      [bracket]
13             > Exp "." Exp                      [strict(1), left]
14             > Exp "+" Exp                      [seqstrict, left]
15             | Exp "-" Exp                      [seqstrict, left]
16             | Exp "*" Exp                      [seqstrict, left]
17             | Exp "/" Exp                      [seqstrict, left]
18             | Exp ">" Exp                      [seqstrict, left]
19             | Exp "<" Exp                      [seqstrict, left]
20             | Exp ">=" Exp                     [seqstrict, left]
21             | Exp "<=" Exp                     [seqstrict, left]
22             | "!" Exp                          [seqstrict, left]
23             | Exp "&&" Exp                     [strict(1), left]
24             | Exp "||" Exp                     [strict(1), left]
25             > Exp "==" Exp                     [seqstrict, left]
26             | "interval" "(" Exp "," Exp ")"
27             > Exp "in" Exp                     [macro]
28             | "parseInt" "(" Exp ")"           [strict]
29             | StandaloneExp
```

Listing 9.2: MediK Expressions Syntax

```
syntax Stmt ::= StandaloneExp ";"                                [strict]
              | "sleep" "(" Exp ")" ";"                          [strict(1)]
              | "send" Exp "," ExtId ";"                         [macro]
              | "send" Exp "," ExtId "," "(" Exps ")" ";"        [seqstrict(1, 3)]
              | "broadcast" Id ";"                               [macro]
              | "broadcast" Id "," "(" Exps ")" ";"              [strict(2)]
              | "goto" Id ";"                                    [macro]
              | "goto" Id "(" Exps ")" ";"                       [strict(2)]
              | "print" "(" Exp ")" ";"                          [strict]
              > "return" ";"                                     [macro]
              | "return" Exp ";"                                 [strict(1)]
              | "var" Id "=" Exp ";"                             [macro]
              > Exp "=" Exp ";"                                  [strict(2)]
              > "var" Id ";"
              | "vars" Ids ";"                                   [macro-rec]
              | Block
              > "if" "(" Exp ")" Block                           [strict(1)]
              | "if" "(" Exp ")" Block "else" Block              [strict(1)]
              | "while" "(" Exp ")" Block
              | "entry" Block                                    [macro]
              | "entry" "(" Ids ")" Block
              | "on" ExtId "do" Block                            [macro]
              | "on" ExtId "(" Ids ")" "do" Block
              | "fun" Id "(" Ids ")" Block
              | NonDetStmt
              | Exp "in" "{" CaseDecl "}"                        [macro-rec]
              | StateDecl
              > "machine" Id Block                               [macro]
              | "machine" Id "receives" Ids Block
              | "interface" Id Block                             [macro]
              | "interface" Id "receives" Ids Block
              | "init" "machine" Id Block                        [macro]
              | "init" "machine" Id "receives" Ids Block
              | "yield" ";"
              > Stmt Stmt                                        [right]
```

Listing 9.3: MediK Statement Syntax

the event, followed by the keyword `do` a block of the handler's code.

Each entry and event handler block contains statements defined by syntax shown in **??**. The statements are written over expressions given by the syntax in **??**.

Recall from **??**, in K, productions are defined using the keyword `syntax`, where terminals are enclosed in quotes (`""`), and non-terminals begin with an upppercase character.

MediK uses statement over expressions resemble counterparts in many commonly used programming languages. For instance, Lines **??**-**??** enable one to write expressions using program identifiers (denoted by the builtin K production `Id`), and values such as booleans, or rationals, or "`this`", which enables an instance to refer to itself. **??** defines the usual dot operator (`.`), which can be used to access members of an instance. Lines **??**-**??** declare common expressions such as `+, -. >, >=` over rationals and `&&, ||, !` over booleans. We use the production `StandaloneExp` to define certain expressions that are typically not used in conjunction with other expressions, such as `new (...)` on **??** that resembles constructs in OOP languages used to create object instances. Note however, that MediK also has several constructs not commonly found in other languages, such as `createFromInterface(...)` (**??**) and `obtainFrom(...)` (**??**). We shall describe their need and purpose

in upcoming sections.

As reflected in **??** MediK support many statements commonly found in conventional programming languages, such as variable assignment (**??**), `if-else` (Lines **??** and **??**) and `while` (**??**). Others, such as `broadcast` (Lines **??** and **??**), `goto` and `interface` declarations have nuanced meanings, and will be explained in upcoming sections.

Next, we use the K definition to describe MediK′*s* semantics. Recall the semantics in K has two components:

(1) Description of program state expressed as a configurations.

(2) Rewrite rules over configuration segments that define the program's transition system.

We discuss MediK's configuration in **??**, and corresponding semantics rules in **??**.

9.4 MediK Configuration

Recall that 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 configuration in **??** defines the initial state of any MediK program. For brevity, we only show a part of the configuration to illustrate organization of data in the semantics.

Recall that the keyword `configuration` (**??**) 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`.

A MediK program consists of instances of concurrently executing state machines that communicate by passing events, where every instance manages its own environment. This is clearly reflected in MediK's configuration in **??**. The configuration consists of the following cells, each with a similar composition:

(a) An `<instances>` cell (Lines **??**-**??**) that encapsulates `<instance>` cells.

(b) A `<machines>` cell (Lines **??**-**??**) that holds `<machine>` cells.

(c) An `<instances>` cell (Lines **??**-**??**) that holds `<interface>` cells.

The sub-cells under each of the aforementioned cells the attribute `multiplicity=*` (Lines **??**, **??**, **??**), denoting that zero-or-more copies of the cell may exist in the configuration. We briefly describe the purpose of each such sub-cell.

`<instance>` cells (Lines **??**-**??**) hold the state of each MediK machine instance during execution. As is the case with instances in OOP languages, each instance manages its instance variables using a map in the `<genv>` cell (**??**). Additional, each instance maintains a buffer of incoming events in the `<inBuffer>` cell (**??**) and the currently executing code in the `<k>` cell (Lines **??**). Note that unlike the example single-threaded language in **??**, MediK is inherently concurrent, as reflected by the presence of multiple `<k>` cells. At runtime, if multiple rewrite rules can apply on the configuration, K non-deterministically chooses a rule and applies it, allowing one to specify concurrency through interleavings. However, for MediK, we utilize a structured scheduling algorithm, which will be discussed in upcoming sections.

The `<machine>` cell (Lines **??**-**??**) holds information relevant to a machine definition, such as the name in the `<machineName>` cell (**??**) and states in the `<states>` cell (Lines **??**-**??**) The `<state>` cell holds information

```
configuration <instances>
                <instance multiplicity="*" type="Map">
                  <id> 0 </id>
                  <k> populateCells($PGM:Stmt) ~> createInitInstances </k>
                  <env> .Map </env>
                  <genv> .Map </genv>
                  <activeState> . </activeState>
                  <callerId> . </callerId>
                  <inBuffer> .List </inBuffer>
                </instance>
              </instances>
              <machines>
                <machine multiplicity="*" type="Map">
                  <machineName> . </machineName>
                  <declarationCode> nothing; </declarationCode>
                  <isInitMachine> false </isInitMachine>
                  <receiveEvents> .Set </receiveEvents>
                  <states>
                    <state multiplicity="*" type="Map">
                      <stateName> . </stateName>
                      <stateDeclarations> nothing; </stateDeclarations>
                      <entryBlock>  nothing; </entryBlock>
                      <eventHandlers>
                        <eventHandler multiplicity="*" type="Set">
                          <eventId> . </eventId>
                          <handlerCode> nothing; </handlerCode>
                        </eventHandler>
                      </eventHandlers>
                    </state>
                  </states>
              </machines>
              <interfaces>
                <interface multiplicity="*" type="Map">
                  <interfaceName> . </interfaceName>
                  <interfaceDeclarations> nothing; </interfaceDeclarations>
                  <interfaceReceiveEvents> .Set </interfaceReceiveEvents>
                </interface>
              </interfaces>
              <executorAvailable> true </executorAvailable>
              <stuck> false </stuck>
              <epoch> 0 </epoch>
              <shouldAdvanceEpoch> false </shouldAdvanceEpoch>
```

Listing 9.4: MediK Configuration

relevant to a state, such as the entry block in cell `<entryBlock>` (**??**) and event handlers in cell `<eventHandlers>` (Lines **??**-**??**). Similarly, an `<interface>` cell holds information regarding interface definitions. In MediK, interfaces facilitate communication with the external world, and will be discussed in upcoming sections.

Recall that on execution, K replaces `$PGM` (**??**) with the Abstract Syntax Tree (AST) of the program, obtained by parsing the program using the syntax defined in **??**. In case of a MediK program, K adds a `<instance>` cell that has a `<k>` cell containing `populateCells($PGM) ~> createInitInstances` to the initial configuration. This instance, also referred to as the *implicit* or *initial*, is responsible for populating the configuration with relevant information from the program AST before execution can begin. The `populateCells(...)` construct at the top of the `<k>` cell of the implicit instance is defined (using rewrite rules) to recursively traverse

the program AST translate each `machine` in the MediK program to a corresponding `<machine> ... </machine>` sub-configuration. Next, the `createInitInstances` creates an instance for the machine with the `init` keyword, and schedules its `entry` block for execution. Recall that `~>` symbol (**??**) is interpreted by K as "followed-by", i.e., execution of `populateCells` is followed by execution of `createInitInstances`. During execution, as instances are created using the `new` keyword, the configuration grows dynamically to accommodate them. In MediK, interaction with the external world is handled through `interface` definitions that make up `<interface>` under the `<interfaces>` cell (Lines **??**-**??**). Interface definitions will be discussed in detail in upcoming sections. Also note cells `<executorAvailable>`, `<epoch>`, `<stuck>` and `<shouldAdvanceEpoch>` (Lines **??**-**??**). These cells are utilized in implementing the scheduling strategy MediK uses, and will be discussed in subsequent sections.

9.5 Rules

We briefly recall K rules from **??**. A 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 rule in **??** for updating the value of a local program variable. In the rule, `I`, `V`, `Loc`, and `Store`

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

Listing 9.5: Variable Assignment

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 pt, and cell `<store>` containing a map ρ with some value pointed-to by pt. The LHS matches with substitution θ `= (I` $\mapsto$ `i, V` $\mapsto$ `2, Loc` $\mapsto pt$`, Store` $\mapsto \rho$`)`, resulting in the top of the `<k>` cell to be rewritten to the value 2, and pointer pt updated to point to 2 in ρ. Execution is a sequence of rule applications that continues until no rule can match the current configuration.

Next, we explain several features of MediK relevant to defining BPGs using their corresponding K-rules. Note that we only discuss rules related to idiosyntacries of MediK. Other rules, that implement statement that resemble other definitions are omittted for brevity. For instance, the rules for Arithmetic Expressions and boolean expressions, that resemble counterparts in other languages are ignored.

9.5.1 Instance Creation

MediK, machine definitions are analogous to OOP classes, and instances created using keyword `new` behave as objects. The rule in **??** depicts the semantics of the `new` operator. The rule will match any `<k>` cell that has

`new` at the top of the cell, and rewrite it to a temporary `wait` construct (**??**). Note that the same $\mathbb{K}$ variable, `MName`, is used at the following places in the rule's LHS:

- On **??** to match against the name of the machine whose instance is being created.
- On **??** to identify the `<machine>` (Lines **??**-**??**) with the same name as the one being initialized.

By matching a `<machineName>` cell that has the same name as the one being using `new` on **??**, appropriate instance variable and the state marked with the `init` keyword can be determined by matching against the contents of the `<declarationCode>` and `<state>` cells, as shown on **??** and Lines **??**-**??** respectively. Finally, a new `<instance>` cell can be added to the configuration (Lines **??**-**??**) containing:

- An `<id>` cell containing a unique identifier (**??**) that is also added to the set of active instances (**??**).
- A `<k>` cell (**??**) containing code to initialize instance variables and enter the initial state, obtained by matching on appropriate parts of the `<machine>` cell as described earlier.

Note that when `new` is executed, control is transferred from the caller, i.e., the original instance, to the callee, i.e., the created instance. Once the entry block of the callee completes execution, control is transferred back to the caller along with a *reference* to the newly created instance. We chose this behavior for the `new` operator to maintain consistency with most OOP languages where an object is considered instantiated and ready for use only after all relevant constructors have been executed. While MediK machines do not have direct analogues to OOP constructors, we treat the `entry` blocks of intial states as such, and typically place code commonly found in constructors, such as initialization of instance variables, under them. Upcoming sections shall discuss MediK's scheduling semantics, and our motivations behind them in greater depth.

```
rule <id> SourceId </id>
     <k> new MName (Args) => wait ... </k>
     ( .Bag => <instance>
                 <id> Loc </id>
                 <k> MachineDecls ~> enterState(InitState | Args ) </k>
                 <callerId> SourceId </callerId>
                 <class> MName </class> ...
               </instance> )
     <nextLoc> Loc => Loc +Int 1 </nextLoc>
     <store> ( .Map => (Loc |-> instance(Loc))) ... </store>
     <machine>
      <machineName> MName </machineName>
      <declarationCode> MachineDecls </declarationCode>
      <state>
        <stateName> InitState </stateName>
        <isInitState> true </isInitState> ...
      </state> ...
     </machine>
     <activeInstances> ... (.Set => SetItem(Loc)) </activeInstances>
```

Listing 9.6: New Instance Creation

9.5.2 Sending and Receiving Messages

MediK machines communicating by passing messages. This is facilitated by the `send` statement that enables a machine to deliver an event to a particular machine through a reference to the receiver machine. This is captured by the rule in **??**.

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

Listing 9.7: MediK Send Statement Semantics

When an instance the top of the `<k>` cell has `send`, the rule:

(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 (**??**) respectively.

(ii) Rewrites the top of the `k` cell to `done`, marking the completion of execution for the construct.

(iii) Adds the event and associated arguments to the buffer of incoming events of the instance with `id RecvId` (Lines **??**-**??**).

(iv) The *epoch* decides when the machine can run, and is discussed in Section **??**.

While the `send` construct enables communication between pairs of machines, MediK also supports *multicast* communication, that enables a machine to send to multiple machines at the same time. The `broadcast` construct that facilitates this is described in **??**. Recall from **??** (**??**) that a machine specifies the list of events it accepts as part of its definition, which we refer to as the "receives-list" of the machine. To define the semantics of `broadcast` for some event `EventName` (**??**), we utilize a helper function symbol `getReceivers` (**??**), to obtain all machines that declare `EventName` in their "receives-lists", and use `send` to add the `EventName` to each machine's `<inputBuffer>` (**??**).

```
rule broadcast EventName:Id , ( Args ) ;
  => performBroadcast ( EventName | Args | getRecievers(EventName))

rule performBroadcast ( EventName | Args | ListItem(Id) Recievers)
  =>   send instance(Id), EventName , ( Args ) ;
    ~> performBroadcast ( EventName | Args | Recievers)

rule performBroadcast(_ | _ | .List) => .
```

Listing 9.8: Broadcast Mechanism

9.5.3 External Agents

Recall from **??** that CDSSs need to interact with heterogeneous external agents, such as sensors and monitors for various patient parameters and EHR systems. Ideally, we would like a MediK program to be agnostic to specific details of external agents it interacts with, and for machines to utilize a *uniform* communication mechanism that serves both external and internal communication.

```
interface HeartRateSensor { }

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

Listing 9.9: MediK External Agents

We achieve this by treating all external agents as state machines with transition systems defined outside the MediK program, similar to other languages, such as Object-Oriented Maude, [189]. These external agents are modeled in MediK as `interfaces`. We define an interface as a state machine that has its transition system defined externally. In case of CDSSs, it is common to utilize external sensors to obtain various patient parameters. **??** shows such as example, where data is obtained from an external Heart Rate sensor.

Since we don't have the transition system for the heart rate sensor, we declare it as an interface (**??**). Next, instead of using `new` to create an instance, we use a builtin MediK construct `createFromInterface`, which takes as arguments (a) the inteface name (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 [190] message with the event data, the identifier from **??**, 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 case of the example in **??**, this involves reading from an external heart rate sensor and sending the data back to the MediK process that made the request. To send data to MediK, a JSON message in a pre-specified format that also contains the unique transaction id of the request needs to be written to the *write-end* of the input pipe. Note that the `obtainFrom` construct implements a *request-response* mechanism, where a request initiated from MediK is responded to by an external agent. But, external agents can also initiate communicate using the `broacast` construct described in **??**.

9.5.4 Tables

Recall from **??** that tables indicating normality/abnormality for parameters indexed by age and weight occur commonly in BPGs. For instance, in case of the sepsis BPG from **??**, once a measurement, such as the heart rate has been obtained from a sensor, we need to use a table, such as Table **??** 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 **??**. While the table in **??** could have also been implemented using simple `if-else` constructs, we chose to present data in form of a table to improve overall readability and HCP comprehensibility, as HCPs are already familiar with them from paper-based guidelines. If the heart rate lies any of the intervals (closed on the left, open on the right) on Lines **??**-**??**, the corresponding statement to the right of the colon (`:`) is run. Otherwise the default clause (**??**) is run. We can then define the semantics of tables in terms of `if-else` as shown in **??**.

Note the syntax declarations for tables on **??** of **??** and **??** of **??** have the attributes `[macro]` and `[macro-rec]` respectively. These attributes specify that the corresponding constructs are merely syntactic sugar, and rules written over such constructs "de-sugar" them into other symbols. Rules that operate over symbols with attribute `[macro]` are applied statically at parse time, before execution begins, allowing the

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

Listing 9.10: Vital Signs Tables

productions to be used as convenient syntactic sugar.

9.6 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. Recall from **??** that if multiple K rules can match the current configuration, then for execution, K will choose one non-deterministically. This, in case of a program comprising of concurrent MediK machines 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 **??**

Algorithm 1 MediK Scheduling Semantics

$epoch \leftarrow 0$
$scheduled \leftarrow \{\mathcal{I}^0_{\mathcal{M}_{0,0}}\}$
while $scheduled \neq \emptyset$ **do**
  $\mathcal{I}^{\tau}_{\mathcal{M}_{i,j}} \leftarrow choose\,(scheduled)$ s.t.
    $\tau \leq epoch \wedge enabled(\mathcal{I}^{\tau}_{\mathcal{M}_{i,j}})$
  $scheduled \leftarrow scheduled \setminus \mathcal{I}^{\tau}_{\mathcal{M}_{i,j}}$
  $execute(\mathcal{I}^{\tau}_{\mathcal{M}_{i,j}}, scheduled)$
  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**
    $epoch \leftarrow epoch + 1$
  end if
end while

```
rule E in interval(L, R) => (E >= L) && (E < R)
rule E in { interval(L, U): S:Stmt Cs:CaseDecl }
  => if ((E >= L) && (E < U)) { S } else { E in { Cs } }
rule E in { interval(L, U): S:Stmt }
  => if ((E >= L) && (E < U)) { S }
rule E in { interval(L, U): S1:Stmt default: S2:Stmt }
  => if ((E >= L) && (E < U)) { S1 } else { S2 }
```

Listing 9.11: Implementing Tables

Recall from Section **??** 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 $P = \{\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_{n-1}\}$ be a program with n machines, where $\mathcal{M}_0$ is prefixed with `init`. MediK 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.

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 (**??**) at *epoch* 0. On **??**, 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*$(\mathcal{I}^\tau_{\mathcal{M}_i,j}, \textit{scheduled})$ on Line **??** 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., $\textit{scheduled} \leftarrow \textit{scheduled} \cup \{\mathcal{I}^{epoch+1}_{\mathcal{M}_{i'},j'}\}$. 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 (**??**).

9.7 Timer Semantics

Next, we discuss how MediK handles temporal aspects of BPGs. For instance, consider the Fluid resuscitation guideline BPG from **??**. 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 rule in **??**:

```
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>
```

Listing 9.12: Semantics of `sleep`

`sleep` results in a JSON message being sent to a remote endpoint (Lines **??**-**??**) 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 **??**, **??** and **??**), 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 **??**, 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 rule that handles the external response is shown in **??**. The

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

Listing 9.13: Sleep Completion

`waitForSleepResponse(TId)` (**??**) blocks execution until the external response indicating the expiration of the sleep timer is received in the input buffer (Line **??**-**??**). Once the response is received, the machine instance resumes execution when (a) the execution lock becomes available (indicated by `true` on **??**), and, (b) the epoch the instance was scheduled to resume execution in (**??**) is less than or equal to the current epoch (Lines **??**-**??**).

An `obtainFrom` statement also results in a context switch as shown in **??**. Just as in the case of sleep, a json message is sent to the remote endpoint with the *name* of the parameter being requested and a unique transaction (Lines **??**-reflstline:obtain-from-end), while the machine instance releases the execution lock (**??**), and waits for a response, as shown in **??**.

```
1 rule <k> obtainFrom(instance(Id:Int), Field:String)
2       =>   jsonWrite( { "name"      : "Obtain"
3                       , "args"      : [ Field , .JSONs ] } ...)
4         ~> releaseExecutor
5         ~> waitForObtainResponse(TId) ...
6      </k>
7      <tidCount> TId => TId +Int 1 </tidCount>
```

Listing 9.14: Obtain From Request

As depicted in **??**, once data for the requested field is available, it's communicated as an event to the MediK process as a JSON message, where the response consists of a JSON message with the requested data and the unqiue `transactionId`, placed at the *head* of the `<inBuffer>` cell, and scheduled for execution in the next epoch (Lines **??**-**??**).

9.8 MediK Pediatric Sepsis System

9.8.1 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].

Recall from **??** the guideline for sepsis screening. In **??**, 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 Figure **??**, the step "Obtain Patient Age, Weight, and High Risk Conditions" is translated to states `ObtainAge` (Lines **??**-**??**), `ObtainWeight` (**??**), and `ObtainHighRiskConditions` (**??**) in **??**. Within each of these states, the code permits communication with heterogeneous external agents for obtaining required parameters. For instance, on **??**, 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 . Once all appropriate measurements have been obtained, they are checked for abnormality (Lines **??**-**??**) using tables shown in to arrive upon a diagnosis.

Apart from structurally resembling the workflow, the code permits communication with heterogenous external agents, and *obtains* patient parameters such as age and weight, and uses them to evaluate the patient for sepsis. For instance on **??**, we send an event to the external machine that is the tablet held by the HCP, to obtain the patient's age. Once the age is entered, an event `ConfirmAgeEntered` is sent to the machine, prompting the machine to move on to collecting the patient's weight. When evaluating whether the patient meets the criteria for sepsis, we use functions, such as the one on **??**, that obtain the value from a sensor, and check whether it lies in acceptable ranges using tables indexed by the patient's age and weight, similar to the paper-based BPG. Treatment machines *receive* events from the screening machine, and are triggered if the diagnosis is *sepsis positive*, suggesting commencement of treatment.

Recall from **??** that once a sepsis diagnosis has been arrived upon, one of the guideline suggested actions include administering fluids as shown in **??**. In **??**, 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 (**??**). The next steps include:

(a) Obtaining any *risks* associated with administering fluids (Lines **??**-**??**).

```
rule <k> waitForObtainResponse(_) => V ... </k>
     <inBuffer> (ListItem(event($ObtainResponse | V:Val  | Epoch )) => .List)
                ...
     </inBuffer>
     <executorAvailable> true => false </executorAvailable>
     <epoch> Epoch </epoch>
```

Listing 9.15: Obtain From Response

[3] the entire CDSS for sepsis management is available at [191].

```
machine SepsisScreening receives .. {
  init state Start {
    on StartScreening do {
      goto ObtainAge;
    }
  }
  state ObtainAge {
    entry {
      send tablet, Instruct, ("get age");
    } on ConfirmAgeEntered do {
      goto ObtainWeight;
    }
  }
  state ObtainWeight { ... }
  state ObtainHighRiskConditions { ... }
  state CalculateScore {
      var hrAbnormal = !isInNormalRange("HR", ...);
      var bucket1    = hrAbnormal ||  ...
      var bucket3    = mentalStatusAbnormal || ...

      var sepsisSuspected
        = bucket1 && bucket2 && bucket3;

      send tablet, SepsisDiagnosis
        , (sepsisSuspected);
  }
}
```

Listing 9.16: Sepsis Screening in MediK

(b) Suggesting an appropriate *dose* to administer based on the risks, if any (Lines **??**-**??**).

(c) Waiting for the HCP to confirm that the suggested dose was administered (Lines **??**-**??**).

Once the dose is administered, the machine waits for the for 15 minutes as specified by the guideline (**??**), before prompting the HCP to evaluate the patient's responsiveness to the administered fluid dose (Lines **??**-**??**), and check for any signs of evaluate overload (Lines **??**-**??**). If the patient exhibits any signs of fluid overload, then a recommendation to handle the overload is made (**??**). Otherwise, the total dose of administered fluid is obtained from an external source (**??**). 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 (**??**), or, (b) switch to inotropic support to address circulatory issues is made (Lines **??**-**??**). If the total dose of administered fluids is less than the maximum allowed limit, then a recommendation to administer one more fluid bolus made (Lines **??**-**??**)).

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

9.8.2 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 [49] 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

```
machine FluidTherapy
  receives StartFluidTherapy, ... {

  init state Start {
    on StartFluidTherapy do {
      goto ObtainRisks;
    }
  }
  state ObtainRisks {
    // Obtain fluid overload related risks
  }
  state SuggestFluidDosage {
    // Suggest a dosage based on risks
  }
  state WaitForAdministerFluidConfirmation {
    // Handler for Normal Saline Administration
    on ConfirmNormalSalineAdministered do {
      sleep(900);
      goto EvaluateResponsiveness;
    }
  }
  state EvaluateResponsiveness {
    entry {
      send tablet
        , Instruct
        , ("get responsiveness to fluids");
    }
    on FluidResponsivenessEntered(responsiveness) do {
      isResponsiveToFluids = responsiveness;
      goto ObtainFluidOverloadSigns;
    }
  }
  state ObtainFluidOverloadSigns {
    // Obtain signs of fluid overload
  }
  state AskNextStep {
    entry {
      var recommendation;
      if (this.fluidOverload) {
        recommendation = "handle fluid overload";
      } else {
        // obtain total saline dose
        if ((totalSalineDose >
              measurementBounds.salineDosageUpperBound) {
          if (isResponseiveToFluids) {
            recommendation = "maintainence fluids"
          } else {
            recommendation = "consider inotropic support";
            broadcast ConsiderInotropicSupport;
          }
        } else {
          recommendation = "repeat fluid bolus";
        }
      }
      // Send recommendation to tablet
      // Wait for HCP response
    }
  }
}
```

Listing 9.17: Fluid Therapy Guidelines

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

Listing 9.18: Modeling Non-Determinism

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 [49]. For program analysis, *ghost* machines substitute external agents, permitting exploration of the state space. During execution, *ghosts* are discarded and replaced by actual external agents. Due to this, ghosts machines may have statements to express non-determinism in processes. Consider, for instance, on a positive sepsis diagnosis, a HCP may chose to either administer fluids first, followed by antibiotics, or vice-versa. MediK supports such non-determinism using `either-or` statements as shown in **??**.

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, that have semantics shown in **??**.

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

Listing 9.19: Abstract Semantics

The use of abstract encodings leads to a reduction of the state space. Recall from Section **??** 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 shown in **??**. 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 **??**, 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.

9.8.3 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 rule in **??**, 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>
```

Listing 9.20: Unresponsive Machine Transition

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 analysis.

9.9 Discussion

In **??**, we described several characteristics of BPGs that a language for expressing computer interpretable best practice guidelines should ideally accommodate. These include (i) comprehensibility for healthcare providers to enable them to verify correctness of medical logic, (ii) ability to accommodate diverse external agents, (iii) execution and analysis tools with correctness guarantees.

In **??**, we discussed progress made by existing approaches towards addressing them. We measured progress along the following lines:

- **Implementation-Specification Gap:** The textual BPG can also be used "as-is" in a CDSS implementation, eliminating any translation from a textual document to its computable encoding.
- **Formal Semantics:** The language has a complete formally defined semantics that serves as a manual for implementing tools and associated ecosystem.
- **Formal Analysis Tools:** Existence of a comprehensive set of analysis tools that can be used to analyze BPGs implemented in the language that evolve as the language evolves.
- **Holistic Safety:** Comprehensive support for formal analysis and tools, including execution engines, with correctness guarantees.

When compared to existing approaches from **??**, MediK is the first computer-interpretable guideline language that has a complete executable semantics. As MediK's tools are derived from its semantics, they are by definition correct-by-construction. As the tools don't have to be maintained independently, updates to the language to accommodate HCP feedback only require corresponding changes to the semantics, and tools evolve automatically.

This work attempted to establish that the semantics-first approach embodied by MediK makes building safer CDSSs easier. We show that MediK's communicating state machines-based approach provides a convenient medium for expressing medical knowledge by utilizing it to build a real-world complex CDSS for management of sepsis in pediatric cases in collaboration with OSF St. Francis Medical Center. MediK's support for modeling external agents as *ghosts* provides a uniform way of formal analysis. We use this

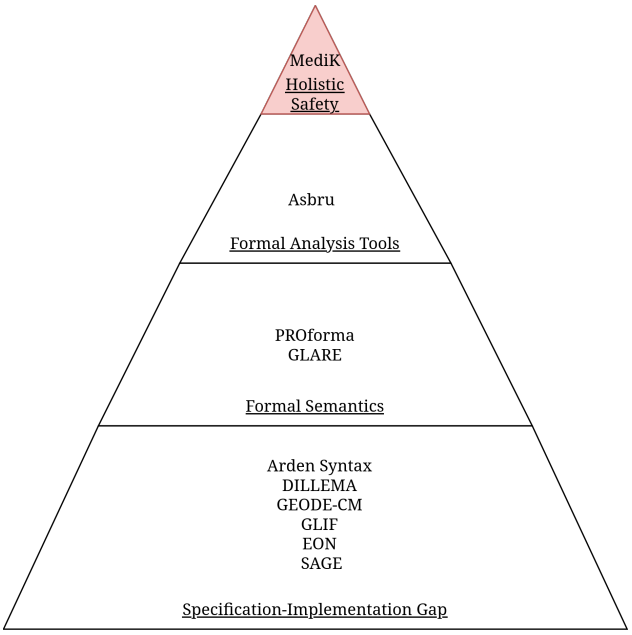


Figure 9.1: Our Contribution

to model-check the sepsis CDSSs for responsiveness, To the best of our knowledge, knowledge, our is the first system for sepsis management with any form of formal analysis. We be briefly summarize how MediK compares against existing approaches in **??**.

In **??**, we described incremental progress made by existing approaches towards enabling greater CDSS adoption through the use of **??**. Specifically, upper rungs of the pyramid incorporate features and learnings of the lower rungs. Our contribution can be described by the run marked in red, as our work incorporates the ability to implement HCP-comprehensible, computable BPGs of earlier approaches with a complete formal semantics, comprehensive support for formal analysis, and correctness guarantees for all tools.

	Implementation-Specification Gap	Complete Formal Semantics	Formal Analysis Tools	Holistic Safety
Arden Syntax	✓	✗	✗	✗
DILLEMA	✓	✗	✗	✗
GEODE-CM	✓	✗	✗	✗
EON	✓	✗	✗	✗
GLIF	✓	✗	✗	✗
Asbru	✓	○	✓	✗
PROforma	✓	✓	✗	✗
GLARE	✓	○	○	○
GPROVE	✓	✓	✗	✗
HELEN	✓	✗	✗	✗
PRODIGY	✓	✗	✗	✗
SAGE	✓	✗	✗	✗
Hierarchical Plans	✗	○	✓	○
MDA	✗	✗	✓	✗
MediK	✓	✓	✓	✓

Table 9.1: MediK vs Existing Approaches