Rewriting-Based Computer-Interpretable Clinical Practice Guidelines

Manasvi Saxena, Xiaohong Chen, Shuang Song, Shaoyu Meng, Lui Sha, and Grigore Rosu

University of Illinois at Urbana Champaign, Urbana, IL, USA
{msaxena2,xc3,shuangs3,smeng9,lrs,grosu}@illinois.edu

Abstract. Clinical Practice Guidelines (CPGs) are systematically developed statements to assist healthcare professionals with decisions about specific clinical circumstances [2]. Produced by domain experts, they they codify recommended treatments, and are often used as models for building Decision Support Systems (DSS) that assist healthcare professionals adhere to CPGs [11]. CPGs are usually specified informally, or semi-formally (via flowcharts). This, however, may cause the behavior of DSSs to deviate from intended treatment in the corresponding CPGs, and introduce challenges in verifying correctness of the DSSs. In this paper, we propose a novel method of formally specifying CPGs by formalizing them in $\mathbb{K}$, a rewriting-based executable semantics framework [13]. Our $\mathbb{K}$-based guidelines are *executable*, allowing them to serve as *correct-by-construction* models in DSSs, and can be analyzed using tools in the $\mathbb{K}$ ecosystem. To support our claims, we present a DSS based on Advanced Cardiac Life Support (ACLS) CPG formalized in $\mathbb{K}$, discuss challenges and present directions for future work.

Keywords: Computer-Interpretable Guidelines · Formal Semantics · Decision Support Systems

1 Introduction

Clinical Practice Guidelines (CPG) are systematically developed statements to assist healthcare professionals with decisions about specific medical circumstances [2]. Such guidelines are routinely published by hospitals and medical associations with the intention of standardizing treatment, reducing costs and providing assistance for decision making [16]. However, in order for CPGs to be effective, they must be integrated with care flow, and be available when needed [11]. To address this, CPGs are often used as models for Decision Support Systems (DSSs) that assist healthcare professionals adhere to CPGs. However, traditionally, CPGs are either specified informally, or semi-formally via flowcharts. This results in a gap between the model of a CPG used in a DSS and the actual CPG. In other words, the DSS' model may not conform to the intended semantics expressed informally in the CPG. One way to bridge this gap is to ensure CPGs are *computer-interpretable*, allowing them to be directly used as

models in DSSs. Computer-Interpretable Guidelines (CIG) however, (a) may be verbose and hard are hard to comprehend due to extra information needed for execution, and, (b) may require significant effort to ensure that the intended semantics of the CPG are captured. Due to these drawbacks, the traditional approach of defining CPGs informally and using ad-hoc models of said CPGs in DSSs remains prevalent.

In this paper, we show a novel method of developing CIGs by formalizing CPGs in an executable rewriting-based framework called $\mathbb{K}$ [13]. Rewriting is a well understood and commonly used paradigm for implementing concurrent systems such as Petri Nets and π-calculi [9]. Thus CPGs, which are often defined using concurrent state machines, are well suited for rewriting frameworks. We argue that using a rewrite framework like $\mathbb{K}$ provides a *convenient* and *natural* medium for obtaining *easily comprehensible* and *correct-by-construction* CIGs. Moreover, formalizing a CPG in $\mathbb{K}$ also exposes it to a rich set of formal analysis tools in the framework (see figure 1), which can be used to ensure correctness.

To support our claim, we present $\mathbb{K}$-ACLS: a CIG for management of resuscitation using a formalization of Advance Cardiac Life Support (ACLS) CPG in $\mathbb{K}$. We show that $\mathbb{K}$-ACLS is a *succinct*, *conforming* and *correct* formalization of ACLS. We demonstrate the succinctness of $\mathbb{K}$-ACLS by showing how rules from the $\mathbb{K}$ formalization concisely capture involved procedures in the guidelines.

We briefly describe the organization of the rest of this paper. In section 2, we describe the $\mathbb{K}$ framework, and the ACLS resuscitation CPG. In 3, we present a $\mathbb{K}$-based CIG of the ACLS resuscitation CPG. We then discuss related work in 4 and conclude in section 5.

2 Preliminaries

In this section we briefly describe the $\mathbb{K}$ framework and the ACLS CPG.

2.1 The $\mathbb{K}$ Framework

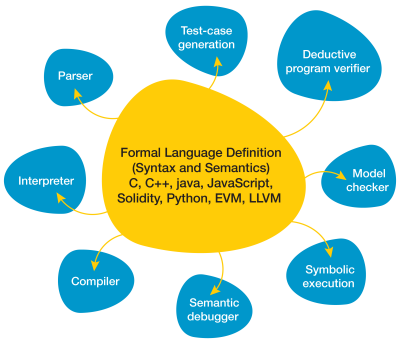
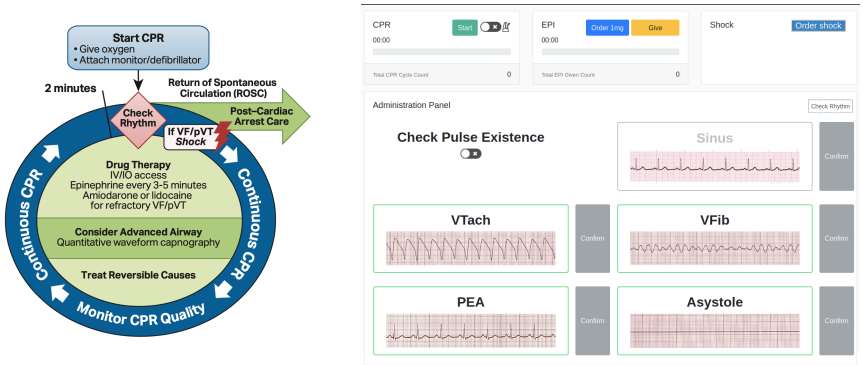


Fig. 1: Overview of the $\mathbb{K}$ Framework

$\mathbb{K}$ is a language semantics framework in which programming languages, calculi and transition systems can be defined [13,17]. $\mathbb{K}$ is centered around a *semantics-first* approach—once the semantics of a programming language or system has been formally defined, it can be used as a *canonical model* to obtain *correct-by-construction* tools such as interpreters, debuggers and program verifiers as shown in figure 1, eliminating the gap between the formal specification of a language and the model of said language used in tools. $\mathbb{K}$ has been designed with the intention of



(a) *Informal* Algorithm (b) *Formal* $\mathbb{K}$-ACLS DSS

Fig. 2: ACLS resuscitation CPG (left) and $\mathbb{K}$-ACLS DSS (right)

simplifying the task of capturing the semantics of a system. This is reflected in the fact that it has been used to successful capture the semantics of several real world languages such as C [6], Java [3], Javascript [10] and EVM [7]. We will further discuss features of $\mathbb{K}$ that make it suitable for formalizing CPGs in section 3.

We now breifly describe the process of formalizing systems in $\mathbb{K}$. The state of a running system in $\mathbb{K}$ is represented in a *configuration*. A $\mathbb{K}$ *configuration* is a nested structured to labeled cells containing various computation based data structures [17]. The behavior of the system is specified using rewrite rules that specify how the configuration of a running system evolves during execution. For brevity, we will briefly explain $\mathbb{K}$ rules when we describe $\mathbb{K}$-ACLS in section 3. For more information on the $\mathbb{K}$ framework and its underlying logical foundations, we refer the reader to dedicated resources on $\mathbb{K}$ [13,17,4].

2.2 ACLS Guidelines

Cardiopulmonary arrest can lead to death within minutes. The Advanced Cardiovascular Life Support (ACLS) guidelines by the American Heart Association (AHA) are clinical algorithms designed to assist healthcare professionals in management of cardiovascular emergencies. The guidelines require an electrocardiography (EKG) machine to monitor the patient's *cardiac rhythm*, which can be *shockable* or *unshockable*. In case of a *shockable rhythm*, a *defibrillator* device can be used to administer a therapeutic electric shock. Furthermore, *Cardiopulmonary Resuscitation* (CPR), and drugs such as *Epinephrine* must be periodically administered. The algorithms is succinctly presented in figure 2a taken from the AHA's ACLS documentation [1]

3 $\mathbb{K}$-ACLS Computer-Interpretable Guidelines

In this section, we briefly describe the $\mathbb{K}$-ACLS Computer-Interpretable Guidelines (CIG) (shown in figure 2b) that formalized the AHA's informal algorithm shown in figure 2a. Our tool consists of: (a) a simple *Javascript Frontend* written in ReactJS, and, (b) the $\mathbb{K}$ *Backend* that acts as an HTTP server that formalizes the guidelines. The use of a Javascript based simple frontend allows our application to run on any modern web-browser. As shown in figure 2b, our frontend consists of forms and buttons that correspond to procedures in the algorithm in figure 2a. 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.

3.1 $\mathbb{K}$ Backend

The $\mathbb{K}$ backend receives events from the frontend, processes them and dispatches the results back to the frontend. To formalize the informal algorithm from figure 2a, we formalize different procedures as *finite state machines* (FSMs) that can operate in parallel.

A system in $\mathbb{K}$ is defined using *configurations* and *rules*. *configurations* organize data in units called *cells* that can be labeled and nested. Following is a fragment of the $\mathbb{K}$-ACLS configuration:

```
configuration
        <machines>
            <machine multiplicity="*" type="Set">
                <id>    .K   </id>
                <state> .K   </state>
                <store> .Map </store>
            </machine>
        </machines>
        <inputBuffer>  .JSONs </inputBuffer>
        <outputBuffer> .JSONs </outputBuffer>
```

The `<machine>` cell stores all data related to an FSM. The `<id>` and `<state>` cells store the *name* and *state* of a machine respectively. The `<store>` cell is a map used to store any additional data. The `configuration` also specifies the initial value for each cell. The `<inputBuffer>` and `<outputBuffer>` cells are used to communicate with the frontend. The buffers are comma-separated lists of JSON objects. 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

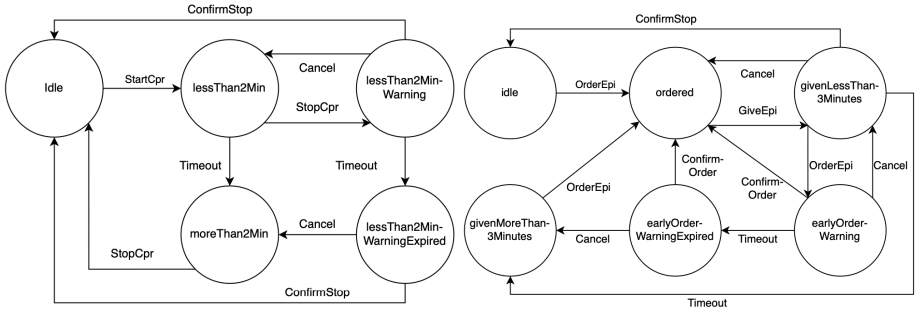
Initialization involves populating the configuration with the initial state of each state machine. When the frontend is started, it sends initialization events to the backend. $\mathbb{K}$ rules then initialize the FSMs by populating the `<machines>` cell. For instance, the following rule initializes the *CPR Machine*:

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

When `<inputBuffer>` cell has the event `"cprMachine.init"` at the head buffer, the rewrite rule above fires. Lines 2-6 of the rule introduce a new `<machine>` cell with name `cprMachine` and initial state `idle`. In line 9, 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 `<inputBuffer>` is rewritten to just the tail of the list, resulting in the removal `"cprMachine.init"` event from the cell.

FSM Definitions

$\mathbb{K}$-ACLS consists of multiple machines that run concurrently. We next describe the *CPR machine*, the *Epinephrine machine* and the *Shock machine* responsible for the *CPR Pane*, the *Epineprhine Pane* and the *Administer shock* pane respectively.



(a) CPR Machine (b) Epinephrine Machine

Fig. 3: Formal Machine Definitions

CPR Machine

The *CPR Machine* encodes the intended CPR procedure shown in figure 3a. When the user clicks the "Start" button in the CPR pane in figure 2b, a 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. The following rule is trigger when the "Start" button in clicked:

```
1    rule <machine>
2           <id> cprMachine </id>
3           <state> idle => lessThan2Min </state>
4           <store> ( "cprRunning" |-> (_ => true) ) ... </store>
5           ...
6        </machine>
7        <inputBuffer> "StartCpr" , IN => IN </inputBuffer>
8        <outputBuffer>
9            OUT => OUT +JSONs jsonResponse( cprMachine
10                                         | lessThan2Min
11                                         | startTimer( .JSONs ) )
12       </outputBuffer>
```

The rule above: (a) consumes the `"StartCpr"` event from the `<inputBuffer>` in line 9, (b) changes the state of the machine from `idle` to `lessThan2Min` in line 3 and, (c) Sends a `startTimer` action to the frontend via the `<outputBuffer>` (encoded as JSON), which results in the start of a two minute timer in the frontend. The rule uses several $\mathbb{K}$ features that greatly simplify the definitions. $\mathbb{K}$'s *configuration abstraction* mechanism allows user to *only* specify parts of the configuration needed in the rule. The ... (dots) on lines 4 and 5 are used to ignore parts contents of a cell not used in the rule. Another important feature is

$\mathbb{K}$'s support for *local rewrites*, which makes it easy to rewrite subterms in deeply nested configurations.

Epinephrine Machine

The *Epinephrine Machine* encapsulates instructions for administering the drug Epinephrine to the patient. The corresponding machine is shown in figure 3b. The user orders the drug using the "Order 1mg" button and can administers it using the "Give" button in Epinephrine pane in figure 2b. Giving the drug results in the start of a three minute timer, during which if a new order is placed, a warning is issued.

Next we show the rule formalizing the transition between `givenLessthan3Min` and `earlyOrderWarningNotification`. The rule fires when it has been less than three minutes before the last dose and a fresh order for Epinephrine. Since 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.

```
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>
```

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.

```
1       rule <machine>
2              <id> shockMachine </id>
3              <state>
4                shockableRhythm => checkRhythm
5              </state> ...
6            </machine>
7            <machine>
8              <id> cprMachine </id>
9              <store> ( "cprRunning" |-> false ) ... </store> ...
10           </machine>
11           <machine>
12             <id> epiMachine </id>
13             <store> ( "epiGiven" |-> true ) ... </store> ...
14           </machine>
15           <inputBuffer> "administerShock" , IN => IN </inputBuffer>
16           <outputBuffer> OUT
17           => OUT +JSONs jsonResponse( shockMachine
18                                     | checkRhythm
19                                     | confirmShock("CPR for
20                                                    2 Minutes" ) )
21           </outputBuffer>
```

The rule abovfires when the rhythm is *shockable*, and the user wants to administer a shock. Additionally, CPR hasn't been administered (line 9), but Epinephrine has been administered (line 13). In such a situation, a suggestion to administer CPR for two minutes is displayed to the user (lines 19-20). 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*.

4 Related Work

Decision Support Systems (DSSs) have found increasing adoption and have been shown to significantly improve the qualify of patient care [8]. However, in order to be accepted into routine workflow, DSSs need to be easily comprehensible to healthcare professionals [15]. In recent years, languages such as the Arden Syntax [14] and PROforma [5] have emerged as a viable option for encoding CPG rules. However, they lacks the readability and conciseness offered by the $\mathbb{K}$-based approach due to $\mathbb{K}$'s support for *configuration abstraction*, *local rewriting*

and inherent easy in expressing *concurrent computation*. Rahmaniheris et al. [12] have generated DSSs by embedding the underling semantics using Yakindu state machines. Unlike $\mathbb{K}$ however, Yakindu code cannot be directly executed. Yakindu models are used to generate Java code which is used in the DSS, leading to a gap in specification and implementation.

5 Future Work and Conclusion

The next step in our $\mathbb{K}$-based approach is to use tools in the $\mathbb{K}$ ecosystem, such as the deductive verifier to prove safety properties about the DSS. We also intend to build a Domain Specific Language to further simplify the process of formalizing CPGs in $\mathbb{K}$.

In this paper, we presented a novel approach for developing *correct-by-construction* DSSs by specifying the underlying CPGs in a rewriting framework called $\mathbb{K}$. Our $\mathbb{K}$-based approach results in *concise*, *performant* and *comprehensible* CIGs that can directly be used as models in DSSs. To buttress our claim, we presented a DSS for the ACLS cardiac resuscitation CPG that uses as its model the $\mathbb{K}$-ACLS CIG—a formalization of said CPG in the $\mathbb{K}$ framework. Our $\mathbb{K}$-ACLS CIG is *concise*, easily *comprehensible*, and directly serves as an executable model in our DSS.

References

1. American heart association emergency cardiovascular care, `https://cpr.heart.org/en/resuscitation-science/cpr-and-ecc-guidelines/algorithms`
2. Clinical practice guidelines, `https://www.nccih.nih.gov/health/providers/clinicalpractice`
3. Bogdănaş, D., Roşu, G.: K-Java: A complete semantics of Java. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15). pp. 445–456. ACM (January 2015). https://doi.org/10.1145/2676726.2676982
4. Chen, X., Rosu, G.: Matching mu-logic. In: 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–13. IEEE Computer Society, Los Alamitos, CA, USA (jun 2019). https://doi.org/10.1109/LICS.2019.8785675
5. Fox, J., Johns, N., Lyons, C., Rahmanzadeh, A., Thomson, R., Wilson, P.: Proforma: a general technology for clinical decision support systems. Computer Methods and Programs in Biomedicine **54**(1-2), 59–67 (1997). https://doi.org/10.1016/s0169-2607(97)00034-5
6. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of c. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 336–345. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737979
7. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022
8. Kawamoto, K., Houlihan, C.A., Balas, E.A., Lobach, D.F.: Improving clinical practice using clinical decision support systems: a systematic review of trials to identify features critical to success. British Medical Journal **330**(7494), 765 (2005). https://doi.org/10.1136/bmj.38398.500764.8F
9. Meseguer, J.: Rewriting as a unified model of concurrency. In: Proceedings of the Workshop on Object-Based Concurrent Programming. p. 86–88. OOPSLA/ECOOP '90, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/127056.127091
10. Park, D., Stefănescu, A., Roşu, G.: Kjs: A complete formal semantics of javascript. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 346–356. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737991
11. Peleg, M.: Computer-interpretable clinical guidelines: A methodological review. Journal of Biomedical Informatics **46 4**, 744–63 (2013). https://doi.org/10.1016/j.jbi.2013.06.009
12. Rahmaniheris, M., Jiang, Y., Sha, L.: Model-driven design of clinical guidance systems. ArXiv **abs/1610.06895** (2016)
13. Roșu, G., Șerbănuță, T.F.: An overview of the k semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
14. Samwald, M., Fehre, K., De Bruin, J., Adlassnig, K.P.: The arden syntax standard for clinical decision support: experiences and directions. Journal of Biomedical Informatics **45**(4), 711–718 (2012). https://doi.org/10.1016/j.jbi.2012.02.001
15. Shortliffe, E.H., Sepúlveda, M.J.: Clinical decision support in the era of artificial intelligence. Journal of American Medical Association **320**(21), 2199–2200 (2018). https://doi.org/10.1001/jama.2018.17163

16. Woolf, S.H., Grol, R., Hutchinson, A., Eccles, M., Grimshaw, J.: Potential benefits, limitations, and harms of clinical guidelines. British Medical Journal **318**(7182), 527–530 (1999). https://doi.org/10.1136/bmj.318.7182.527
17. Şerbănuţă, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Roşu, G.: The k primer (version 3.3). Electronic Notes in Theoretical Computer Science **304**, 57–80 (2014). https://doi.org/10.1016/j.entcs.2014.05.003, proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).