Towards Verifiable Safe and Correct Medical Best Practice Guideline Systems

Chunhui Guo, Zhicheng Fu, Shangping Ren
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
{cguo13, zfu11}@hawk.iit.edu, ren@iit.edu

Yu Jiang
School of Software
Tsinghua University
Beijing, China
jy1989@mail.tsinghua.edu.cn

Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
lrs@illinois.edu

Abstract—**Improving safety of patient care is an ultimate objective for medical systems. Though many medical best practice guidelines exist and are in hospital handbooks, they are often lengthy and difficult for medical professionals to remember and apply clinically. Hence, developing safe and correct medical best practice guideline systems is an urgent need. Many efforts have been made in modeling, clinical validation, model level formal verification of medical best practice guidelines. However, code level verification is also necessary to develop verifiable safe and correct medical guideline systems. The paper presents an approach to transform safety properties specified in verifiable medical guideline models to JavaMOP runtime monitor and specify JavaMOP monitors to runtime monitor these safety properties during execution of Java code generated from validated and verified statechart models. We use a simplified version of a cardiac arrest scenario provided by Carle Foundation Hospital as a case study to validate the proposed approach.**

I. Introduction and Related Work

Until today, safe and correct patient care is still a big challenge. The challenge is even bigger in rural area when it comes to emergency care [1], [2]. However, deciding the most patient effective care requires knowledge and experience. Though medical best practice guidelines exist and are in hospital handbooks, they are often lengthy and difficult to apply clinically. A safe and correct computerized medical best practice guideline system can help to improve patient care by assisting medical professionals to adhere to medical best practices.

Significant amount of efforts have been made in obtaining various computer-interpretable models and tools for the management of medical guidelines, such as GLIF [3], Asbru [4], EON [5], GLARE [6] and PROforma [7]. However clinical problems are complicated and those formats mentioned above are not visual nor user friendly for medical staffs to validate their correctness. They are also difficult to formally verify. Many medical errors found in the U.S. Food and Drug Administration (FDA) database are due to the lack of rigorous clinical validation and formal verification [8].

Developing computerized disease and treatment models from medical best practice handbooks needs close interactions with medical professionals. Furthermore, to satisfy the safety and correctness requirements, the derived models also need to be clinically validated and formally verified. To help improve clinical validation, Wu *et al.* have developed a workflow adaptation [9] to help physicians safely adapt workflows to react to patient adverse events and a treatment validation protocol [10] to enforce the correct execution sequence of performing a treatment based on preconditions validation, side effects monitoring, and expected responses checking. Rahmaniheris *et al.* [11] have developed organ-centric approaches to model medical best practice guidelines using Yakindu statechart [12] and enabled dynamic clinical validation. To improve statechart models understandability for medical professionals and reduce the difficulty in both clinical validation and formal verification of medical guideline statechart models, Guo *et al.* [13] have proposed a pattern-based statechart modeling approach for medical best practice guidelines, i.e., model medical guidelines with basic statechart elements and model patterns which are built upon these basic elements. For life-critical medical systems, validation by medical staffs alone is not adequate for guaranteeing correctness and safety, formal verification is required. Unfortunately, Yakindu statechart, which is used to model medical guidelines and interact with medical staffs for clinical validation, does not provide formal verification capability. To bridge the gap between validatable models and verifiable models, Guo *et al.* have developed an approach to transform Yakindu statechart models to UPPAAL timed automata and developed the Y2U[1] tool [14] to perform automatic transformation.

Once the statechart guideline models are clinically validated and formally verified, the next step is to generate an executable system based on the models. Instead of manually coding which is highly time consuming and errors-prone, Yakindu [12] code generator can be used to automatically generate executable Java code. However, the generated code is not certified, but code level verification is necessary to develop verifiable safe and correct medical guideline systems. Different approaches of applying model checking to source code verification have also been developed [15], [16], [17], [18], [19]. However, model checking large and complex source code often faces the notorious state explosion issue. In addition to complexity, medical guideline models' execution depend heavily on patient status which is only available and may change at runtime. Runtime verification, on the other hand, verifies properties based on its runtime information. Hence, we apply runtime verification techniques rather than model checking to verify generated code of medical guideline

[1]The Y2U tool is available at www.cs.iit.edu/~code/software/Y2U/index.html.

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.253

systems. Different runtime verification tools/frameworks are developed and applied in many applications [20], [21], [22], [23], [24], [25], [26], [27]. Among them, Monitor Oriented Programming (MOP) [25] is a generalized framework that incorporates monitors into programs. One of the advantage of the MOP over other approaches is that it can be easily extended with new logics and also supports self-recovery at violation parts. JavaMOP [28] is an open source tool that can runtime monitor Java code generated from medical guideline statechart models. However, to the best of our knowledge, there is no prior work correlating medical safety properties between verifiable medical guideline models and runtime code monitors.

The paper presents an approach to transform safety properties specified in verifiable medical guideline models to JavaMOP [28] runtime monitor and specify JavaMOP monitors to runtime monitor these safety properties during execution of Java code generated from validated and verified statechart models.

II. Correlate Safety Properties between Verifiable Models and Runtime Code Monitors

In this section, we present how to correlate safety properties' between verifiable UPPAAL models and generated Java code in four steps: (1) analyze generated code patterns, (2) permit read access of state variables in generated code, (3) transform safety properties from verifiable UPPAAL models to JavaMOP code monitors, and (4) exercise runtime monitoring on generated code executions. The simplified airway laser surgery scenario in Section II-A is used to illustrate our approaches.

A. Simplified Airway Laser Surgery Scenario

Laser surgery [29] is a surgical procedure that uses laser to remove problematic tissues, and is widely used in airway surgery, thoracic surgery, eye surgery, etc. For airway laser surgery, there is a potential danger of an accidental burn if the laser is activated while high oxygen concentration is supplied by the ventilator [30]. Hence, whenever the laser is being activated, the ventilator must be off to block air path from the oxygen concentrate. However, before the $\mathtt{SpO_2}$ *(Saturation of Peripheral Oxygen) level of the patient decreases below a given threshold (assume* 95%*), the laser must be deactivated to open the oxygen flow through the ventilator; otherwise, the patient can suffer a low-oxygen shock.*

In the simplified airway laser surgery example, we make following two assumptions on human operations and human reactions: (1) there is no delay between human operations and laser/ventilator actions; and (2) there is no delay between laser/ventilator deactivations/activations and the $\mathtt{SpO_2}$ level change. There are two medical properties needed to be verified in the simplified airway laser surgery: (1) **P1**: the laser and the ventilator must not be activated at the same time; and (2) **P2**: the laser is activated only if the $\mathtt{SpO_2}$ level is larger than 95%.

We use Yakindu statechart to model the simplified airway laser surgery, as shown in Fig. 1. In the statechart model, We consider two scenarios: (1) **S1**: when an operating surgeon sends the `startLaser` command to the laser to operate the surgery, the ventilator should be deactivated and the laser be activated, respectively; and (2) **S2**: when the $\mathtt{SpO_2}$ level is below a given threshold, the laser should be deactivated and the ventilator be activated, respectively.

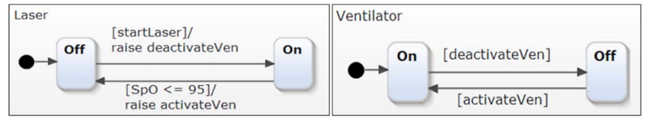

Fig. 1. Airway Laser Surgery Yakindu Statechart Model

We use the Y2U[2] tool [14] to transform the Yakindu statechart model to UPPAAL timed automata to verify medical properties **P1** and **P2**. The transformed airway laser surgery UPPAAL model is shown in Fig. 2. The property **P1** can be checked in UPPAAL by formula $A[\ ]$!(`Laser.On` $\&\&$ `Vent.On`). Similarly, the **P2** can be checked by formula $A[\ ]$ `Laser.On` $imply\ SpO > 95$.

B. Analyze Generated Code Patterns

To runtime monitor the execution of generated code and specify safety properties with JavaMOP, we first need to understand Java code generated by Yakindu. Our study indicates that all Java code generated from Yakindu models follows the patterns below:

- Each Yakindu model is represented by a Java class;
- Each class defines an *enum* type of `State` that contains all states' names in the Yakindu model and a `NullState`;
- Each class also declares a `State` array `stateVector`, whose size is the number of statecharts in the Yakindu model. The `stateVector` stores the current active states of all statecharts in a decreasing order based on the statecharts' priorities;
- Each class also has an `init()` function to initialize the Yakindu model and a function `runCycle()` to execute all statecharts by one step;
- All variables except *events* declared in the Yakindu model have *access* functions to get/set their values, and each *event* variable has a *raise* function to trigger the event.

C. Permit Read Access of State Variables in Generated Code

Most medical properties in medical guideline models involve states and statechart variables. The JavaMOP monitors need to access these variables in the model generated code. The statechart variables can be accessed by their corresponding *get* functions. The generated Java class defines function `isStateActive(Statestate)` to check if the input `state` is currently active. However, JavaMOP can not capture the specific values of Java functions' input parameter variables. Hence, for each statechart in a Yakindu model, we add a function `getActiveState`$\ast$`()` to access the statechart $\ast$'s current active state in the generated code. The `getActiveState`$\ast$`()` function returns the corresponding element of the array `stateVector` based on statechart $\ast$'s priority. As the added functions only read state variables, it does not change the execution behaviors of generated code nor safety properties'

[2]The Y2U tool is available at www.cs.iit.edu/~code/software/Y2U/index.html.

Fig. 2. Airway Laser Surgery UPPAAL Model

runtime monitoring results. We use Example 1 to show state variable access functions in the simplified airway laser surgery.

Example 1. *The statechart model in Fig. 1 contains two statecharts* `Laser` *and* `Ventilator`*, and the statechart* `Laser` *has higher priority. According to generated code patterns analyzed in Section II-B, current active states of statecharts* `Laser` *and* `Ventilator` *are stored in* `stateVector[0]` *and* `stateVector[1]`*, respectively. Listing 1 shows state variable access functions for statecharts* `Laser` *and* `Ventilator`.

```
public int getActiveStateLaser() {
return stateVector[0].ordinal();
}
public int getActiveStateVentilator() {
return stateVector[1].ordinal();
}
```

Listing 1. State Variable Access Functions

D. Transform Safety Properties from Verifiable Models to Code Monitors

JavaMOP supports different formalism logics, including JavaFSM, JavaLTL, JavaERE, etc. We choose to use JavaFSM to specify safety properties for the following two reasons: (1) both validatable models (Yakindu) and verifiable models (UPPAAL) are similar to finite state machines (FSM) and (2) most medical safety properties are related with states in medical guideline models. JavaLTL (Java linear temporal logic) could be an alternative, because in verifiable models (UPPAAL), safety properties are specified with CTL (computation tree logic) which is similar with linear temporal logic. However, based on JavaMOP syntax [31], a JavaLTL monitor can only specify one safety property, while a JavaFSM monitor supports multiple properties. In practice, most medical guidelines contain more than one safety properties. Hence, we choose JavaFSM to specify safety properties in JavaMOP runtime monitors in two steps: (1) define JavaMOP *events* to capture system runtime execution status and (2) specify JavaFSMs to represent safety properties based on runtime execution status.

As state transitions in JavaFSM are triggered by JavaMOP *events*. To specify safety properties with JavaFSM, we need to first define JavaMOP *events* that are related to given properties. To monitor the execution of statecharts, we define a JavaMOP *event* called `RUN`. The `RUN` event is triggered when the generated function `runCycle()` is called. To represent states in medical guideline models, we define an *enum* type `State` in the JavaMOP monitor which has the same `State` type as in the generated code. Suppose a property to be specified is P. For each state S involved in the property P, we define an *event* E_s to capture the activation of state S. The *event* E_s is triggered when the state S's corresponding access function F_s is called and the return of F_s is equal to state S. The state S's access function F_s is added in Section II-C. For each statechart variable V involved in the property P, we define an *event* E_v to capture the violation of variable V's requirement in the property P. The *event* E_v is triggered when the variable V's get function F_v generated by Yakindu is called and the return of F_v negates variable V's expression in the given property P. If a medical guideline has multiple properties to be monitored, we repeat the above procedure to define JavaMOP *events* for each property. We use Example 2 to show how to define *events* for properties **P1** and **P2** in the simplified airway laser surgery.

Example 2. *Consider the properties **P1** and **P2** defined in the simplified airway laser surgery example. First, the enum* `State` *type is copied from the generated code. It contains the four states as in statechart model given in Fig. 1 and a* `NullState`*. The event named* `RUN` *is triggered by function* `runCycle()` *calls. The two properties **P1** and **P2**, i.e., formula* $A[\ ]\ !(\texttt{Laser.On}\ \&\&\ \texttt{Vent.On})$ *and formula* $A[\,]\ \texttt{Laser.On}\ \textit{imply}\ SpO > 95$*, involve two states* `laser_On` *and* `ventilator_on` *and one statechart variable* `SpO`*. So we define events* `laserOn` *and* `ventOn` *to capture the activation of state* `laser_On` *and* `ventilator_on`*, respectively. The event* `SpOLow` *is to capture the violation of* `SpO`*'s requirement in property **P2**, i.e., when* $SpO > 95$*. The* `SpOLow` *event is triggered when the function* `getSpO()` *is called and the returned value is smaller than or equal to* 95*. Listing 2 shows the four JavaMOP events and the enum type* `State` *defined for **P1** and **P2**.*

```
enum State {laser_Off, laser_On, ventilator_On,
ventilator_Off, $NullState$};

event RUN after(Object o) :
call(void *.runCycle()) && target(o) {}

event laserOn after(Object o) returning(int s) :
call(int *.getActiveStateLaser()) && target(o)
&& condition(s==State.laser_On.ordinal()) {}

event ventOn after(Object o) returning(int s) :
call(int *.getActiveStateVentilator()) && target(o)
&& condition(s==State.ventilator_On.ordinal()) {}

event SpOLow after(Object o) returning(long val) :
call(long *.getSpO()) && target(o) && condition(val<=95) {}
```

Listing 2. JavaMOP *Events* for Properties **P1** and **P2**

Once we define JavaMOP *events* to capture system runtime execution status, we specify JavaFSM to monitor given safety properties. Assume a given medical guideline model contains m properties. For the given model, if the JavaMOP *events* defining procedure introduces n_s state related *events*, n_v statechart variable related *events*, and a `RUN` *event*. We use $n = n_s + n_v + 1$ to represent the total number of defined JavaMOP *events*. To specify a JavaFSM monitor for the given model, we add $1 + m + n_s$ states: a `INIT` state, m `UNSAFE` states for each property, and n_s `EVENT` states for each state related JavaMOP *event*. The `INIT` state is the entrance of the specified JavaFSM. Each `UNSAFE` state indicates that the corresponding property fails. Each `EVENT` state indicates that the corresponding state in the given model is currently active. For each JavaFSM state, we add n outgoing transitions which are guarded by each JavaMOP *event*. For all states, their transitions guarded by the `RUN` *event* transit to the `INIT` state to re-initialize the JavaFSM monitor after each execution step of the statechart model. All other $n - 1$ transitions of each JavaFSM state are specified based on state type as following:

- `INIT` state: transit to corresponding `EVENT` state for n_s state related JavaMOP *events* and transit to itself for n_v variable related JavaMOP *events*;
- `UNSAFE` state: transit to itself for all $n-1$ JavaMOP *events* except the `RUN` *event*;
- `EVENT` state: transit to corresponding `UNSAFE` state if the trigger of a JavaMOP *event* violates given properties, otherwise transit to itself.

We use Example 3 to show how to specify a JavaFSM monitor for properties **P1** and **P2** in the simplified airway laser surgery.

Example 3. *The simplified airway laser surgery model contains two safety properties (**P1** and **P2**) and two state related JavaMOP events (*`laserOn` *and* `ventOn` *shown in Listing 2). According to the procedure for specifying JavaFSM, the resulted JavaFSM monitor has five states:* `INIT`, `laserOn_S`, `ventOn_S`, `UNSAFE_P1`, *and* `UNSAFE_P2` *states corresponding to event* `laserOn`, *event* `ventOn`, *property **P1**, and property **P2**, respectively.*

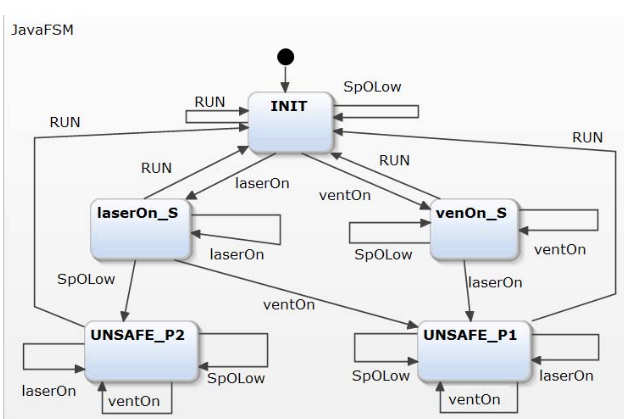

Fig. 3. Graphical Representation of JavaFSM Monitoring Properties **P1** and **P2**

As shown in Listing 2, we defined four JavaMOP events for the airway laser surgery model. Hence, we add four transitions for each state in the JavaFSM monitor. Each of these four transitions is guarded by an event. Take the `laserOn_S` *state as an example. The* `laserOn_S` *state is an* `EVENT` *state. The* `RUN` *event transits the* `laserOn_S` *state to the* `INIT` *state. If the* `ventOn` *event is triggered, the JavaFSM transits from the* `laserOn_S` *state to the* `UNSAFE_P1` *state, as it violates property **P1**. Similarly, the* `SpOLow` *event transits the* `laserOn_S` *state to the* `UNSAFE_P2` *state. Listing 3 shows the JavaFSM monitor specified for **P1** and **P2**.*

In the JavaFSM monitor, we also define two alias `fails1` *and* `fails1` *to indicate fail of **P1** and **P2**, respectively. This example also shows that a JavaFSM monitor support multiple safety properties. The graphical Yakindu statechart representation of the JavaFSM monitor is depicted in Fig. 3.*

```
fsm:
INIT [
RUN -> INIT
laserOn -> laserOn_S
ventOn -> ventOn_S
SpOLow -> INIT
]
laserOn_S [
RUN -> INIT
laserOn -> laserOn_S
ventOn -> UNSAFE_P1
SpOLow -> UNSAFE_P2
]
ventOn_S [
RUN -> INIT
laserOn -> UNSAFE_P1
ventOn -> ventOn_S
SpOLow -> ventOn_S
]
UNSAFE_P1 [
RUN -> INIT
laserOn -> UNSAFE_P1
ventOn -> UNSAFE_P1
SpOLow -> UNSAFE_P1
]
UNSAFE_P2 [
RUN -> INIT
laserOn -> UNSAFE_P2
ventOn -> UNSAFE_P2
SpOLow -> UNSAFE_P2
]

alias fails1 = UNSAFE_P1
alias fails2 = UNSAFE_P2
@fails1 {System.out.println("P1 Fails!!!");}
@fails2 {System.out.println("P2 Fails!!!");}
```

Listing 3. JavaFSM Monitoring Properties **P1** and **P2**

E. Exercise Runtime Monitoring

To runtime monitor generated code with JavaFSM monitors specified according to the procedure defined above, we need to add the `main` function to initialize and execute medical guideline validatable models (statecharts). The `main` function provides following four functionalities: (1) create an instance for the statechart model, (2) initialize the statechart model instance, (3) set initial states for each statechart, and (4) execute the statechart model step by step, i.e., call function `runCycle()`. After each execution step, we call *get* functions on states and variables involved in safety properties to trigger JavaMOP *events* defined based on the procedure for defining JavaMOP *events*. We exercise runtime monitoring of properties **P1** and **P2** in the simplified airway laser surgery example. Listing 4 gives an example of the `main` function for runtime monitoring of properties **P1** and **P2** for the two scenarios **S1** and **S2** in Section II-A. The two scenarios are simulated by raising the `startLaser` event and setting the value of $\mathtt{SpO_2}$ as 94, respectively. The monitoring results show that both property **P1** and **P2** are satisfied.

```java
public static void main(String[] args){
AirwayLaserSurgeryStatemachine sm =
new AirwayLaserSurgeryStatemachine();
sm.init();
sm.stateVector[0] = State.laser_Off;
sm.stateVector[1] = State.ventilator_On;

// scenario S1
sm.raiseStartLaser();
sm.runCycle();
sm.getActiveStateLaser();
sm.getActiveStateVentilator();
sm.getSpO();

// scenario S2
sm.setSpO(94);
sm.runCycle();
sm.getActiveStateLaser();
sm.getActiveStateVentilator();
sm.getSpO();
}
```

Listing 4. Runtime Monitoring `main` Function

III. Simplified Cardiac Arrest Case Study

In this section, we perform a case study on a simplified version of a cardiac arrest scenario provided to our team by Carle Foundation Hospital to validate the proposed approaches.

A. Simplified Cardiac Arrest Scenario

Cardiac arrest is the abrupt loss of heart function and can lead to death within minutes. In a simplified cardiac arrest scenario [10], medical staff intend to activate a defibrillator to deliver a therapeutic level of electrical shock that can correct certain types of deadly irregular heart-beats such as ventricular fibrillation. The medical staff need to check two preconditions: (1) patient's airway and breathing are under control and (2) the EKG (electrocardiogram) monitor shows a shockable rhythm. Suppose the patient's airway is open and breathing is under control, but the EKG monitor shows a non-shockable rhythm. In order to induce a shockable rhythm, a drug, called epinephrine (EPI), is commonly given to increase cardiac output. Giving epinephrine, however, also has two preconditions: (1) patient's blood pH value should be larger than 7.4 and (2) urine flow rate should be greater than 12 mL/s. In order to correct these two preconditions, sodium bicarbonate should be given to raise blood pH value, and intravenous (IV) fluid should be increased to improve urine flow rate.

There are two medical properties needed to be verified in the cardiac arrest treatment validation procedure: (1) **P3**: Defibrillator is activated only if the EKG rhythm is shockable and airway and breathing is normal; and (2) **P4**: Epinephrine is injected only if the blood pH value is larger than 7.4 and urine flow rate is higher than 12 mL/s.

B. Validatable and Verifiable Models

In our previous work, Wu *et al.* developed a validation protocol to enforce the correct execution sequence of performing treatment, regarding preconditions validation, side effects monitoring, and expected responses checking based on the pathophysiological models [10]. Both the simplified cardiac arrest scenario and the validation protocol in [10] are validated by physicians from Carle Foundation Hospital. Based on the validation protocol, we designed a validatable model of the simplified cardiac arrest treatment procedure using Yakindu statecharts and transformed the validatable model to a verifiable model (UPPAAL timed automata) with the Y2U tool [14]. Due to page limit, we omit the validatable model and the verifiable model which can be found in [14] (Fig. 11 and Fig. 12). Both models consist of the following statecharts/automata: `Treatment`, `Ventilator`, `EPIpump`, `SodiumBicarbonatePump`, `IVpump`, `LasixPump`. The `Treatment` statechart implements preconditions validation, side effects monitoring, and expected responses checking. The other statecharts implement treatment actions (such as medicine injection). We focus on runtime execution monitoring in this paper.

The two medical properties **P3** and **P4** can be checked in UPPAAL by following two formulas: (1) **P3**: $A[\,]\texttt{Treatment.ActivateDefibrillaotr}\ imply$ $\texttt{Breath} == 0\ \&\&\ \texttt{Rhythm} == 0$ and (2) **P4**: $A[\,]\texttt{Treatment.InjectEPI}\ imply\ \texttt{BloodPH}_{\texttt{int}} >= 7$ $\&\&\ \texttt{BloodPH}_{\texttt{frac}} > 4\ \&\&\ \texttt{UrineFlow}_{\texttt{int}} > 12$. The simulation results in Yakindu and verification results in UPPAAL show that both medical properties, i.e., **P3**, **P4**, are satisfied.

C. Runtime Execution Monitoring

We generate Java code of the simplified cardiac arrest treatment procedure using Yakindu and runtime monitor its execution with JavaMOP. Take the medical property **P3** as an example to show how to correlate safety properties between verifiable models and code with approaches presented in Section II.

First, for each statechart in the model, we add a function to get its current active state. For instance, the function for statechart `Treatment` is shown in Listing 5.

```java
public int getActiveStateTreatment() {
    return stateVector[0].ordinal();
}
```

Listing 5. State Variable Access Function

The medical property **P3** involves one state, i.e., state `ActivateDefibrillaotr` of statechart `Treatment`, and two patient status variables, i.e., `Breath` and `Rhythm`. According to the procedure for defining JavaMOP *events*, we define three events in JavaMOP specification for the property **P3**, i.e., `Treatment_ActivateDefibrillaotr`, `BreathAbnormal`, and `RhythmNonShockable`. These *events* are triggered during code execution when the state `ActivateDefibrillaotr` is active, `Breath` is abnormal, and `Rhythm` is non-shockable, respectively. Listing 6 gives the definitions of the *events* in JavaMOP.

```
event Treatment_ActivateDefibrillaotr after(Object o) returning(int s) :
    call(int *.getActiveStateTreatment()) && target(o) &&
    condition(s==State.treatment_ActivateDefibrillaotr.ordinal()) {}

event BreathAbnormal after(Object o) returning(String str) :
    call(String *.getBreath()) && target(o) && condition(str!="Normal") {}

event RhythmNonShockable after(Object o) returning(String str) :
    call(String *.getRhythm()) && target(o) && condition(str!="Shockable") {}
```

Listing 6. JavaMOP Events

Based on the procedure for specifying JavaFSM, we specify a JavaFSM for the property **P3**. The JavaFSM contains three states: `init`, `Def`, and `UNSAFE`. At the beginning of code execution monitoring, the JavaFSM is at state `init`. During monitoring, if the event `Treatment_ActivateDefibrillaotr` is triggered, the JavaFSM transits to state `Def`. If the state `Def` is active, once either event `BreathAbnormal` or `RhythmNonShockable` is triggered, the JavaFSM transits to state UNSAFE. The property **P3** fails if the state UNSAFE is reached. Listing 7 shows the JavaFSM monitoring **P3**.

```
fsm:
 INIT [
     Treatment_ActivateDefibrillaotr -> Def
     BreathAbnormal -> init
     RhythmNonShockable -> init
 ]
 Def [
     Treatment_ActivateDefibrillaotr -> Def
     BreathAbnormal -> UNSAFE
     RhythmNonShockable -> UNSAFE
 ]
 UNSAFE [
     Treatment_ActivateDefibrillaotr -> UNSAFE
     BreathAbnormal -> UNSAFE
     RhythmNonShockable -> UNSAFE
 ]
 alias fails = UNSAFE
 @fails {System.out.println("Fails!!!");}
```

Listing 7. JavaFSM Specification

Similarly, we can define JavaMOP events and specify JavaFSM for the medical property **P4**. To monitor the runtime execution, we set the initial value of variables Breath and Rhythm as abnormal and non-shockable, respectively, and run the statechart model 100 steps. The monitoring results also show that both property **P3** and **P4** are satisfied.

IV. Conclusion

Developing verifiable safe and correct medical best practice guideline systems is a focal challenge in medical system design. Many efforts have been made in modeling, clinical validation, model level formal verification of medical best practice guidelines. However, code level verification is also necessary to develop verifiable safe and correct medical guideline systems. The paper presents an approach to transform safety properties specified in verifiable medical guideline models to JavaMOP runtime monitor and specify JavaMOP monitors to runtime monitor these safety properties during execution of Java code generated from validated and verified statechart models. The case study of a simplified cardiac arrest treatment scenario provided by Carle Foundation Hospital demonstrates that our approach can improve safety of medical guideline systems.

Our future work includes: (1) trace failed safety properties from runtime monitors back to both validatable models and verifiable models; and (2) automate safety properties' transformation from validatable models to runtime monitors.

Acknowledgement

This work is supported in part by NSF CNS 1545008 and NSF CNS 1545002.

References

[1] LivingHistoryFarm. Farming 1970s to today. http://www.livinghistoryfarm.org/farminginthe70s/life_01.html.
[2] Karen E. Joynt. Mortality rate at rural hospital unusually high. http://www.medicalnewstoday.com/articles/258598.php.
[3] Vimla L Patel, Vanessa G Allen, José F Arocha, and Edward H Shortliffe. Representing clinical guidelines in glif. *Journal of the American Medical Informatics Association*, 5(5):467–483, 1998.
[4] Michael Balser, Christoph Duelli, and Wolfgang Reif. Formal semantics of asbru an overview. *Proc. of the 6th Biennial World Conference on Integrated Design and Process Technology*, 5(5):1–8, 2002.
[5] Samson W. Tu and Mark A. Musen. Modeling data and knowledge in the eon guideline architecture. *Medinfo*, pages 280–284, 2001.
[6] Paolo Terenziani, Stefania Montani, Alessio Bottrighi, Mauro Torchio, Gianpaolo Molino, and Gianluca Correndo. The glare approach to clinical guidelines: main features. *Stud. Health Technol. Inform.*, pages 162–166, 2004.
[7] John Fox, Nicky Johns, and Ali Rahmanzadeh. Disseminating medical knowledge: the proforma approach. *Artificial Intelligence in Medicine*, 14(12):157 – 182, 1998.
[8] U.S. Food and Drug Administration. Medical device databases. http://www.fda.gov/medicaldevices/deviceregulationandguidance/databases/.
[9] Po-Liang Wu, Lui Sha, Richard B. Berlin Jr., and Julian M. Goldman. Safe workflow adaptation and validation protocol for medical cyber-physical systems. In *To apper in 2015 EUROMICRO Conference on Software Engineering and Advanced Applications*, 2015.
[10] Po-Liang Wu, D. Raguraman, Lui Sha, R.B. Berlin, and J.M. Goldman. A treatment validation protocol for cyber-physical-human medical systems. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 183–190, Aug 2014.
[11] M. Rahmaniheris, P. Wu, L. Sha, and R. R. Berlin. An organ-centric best practice assist system for acute care. In *2016 IEEE 29th International Symposium on Computer-Based Medical Systems (CBMS)*, pages 100–105, June 2016.
[12] Itemis. Yakindu statechart tools (sct). https://www.itemis.com/en/yakindu/statechart-tools/, 2016.
[13] Chunhui Guo, Zhicheng Fu, Shangping Ren, Yu Jiang, Maryam Rahmaniheris, and Lui Sha. Pattern-based statechart modeling approach for medical best practice guidelines - a case study. In *2017 IEEE 30th International Symposium on Computer-Based Medical Systems (CBMS)*, June 2017.
[14] Chunhui Guo, Shangping Ren, Yu Jiang, Po-Liang Wu, Lui Sha, and Richard Berlin. Transforming medical best practice guidelines to executable and verifiable statechart models. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–10, April 2016.
[15] P. Godefroid. Model checking for programming languages using verisoft. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan- guages*, January 1997.
[16] P. Godefroid. A tool for the automatic analysis of concurrent reactive software. In *9th International Conference on Computer Aided Verification*, 1997.
[17] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 1052–1059, Nov 2005.
[18] Joseph J. Benich and Peter J. Carek. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.
[19] Claudio Demartini et al. Modeling and validation of java multithreading applications using spin. 1998.
[20] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1:172–179, 2004.
[21] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. Softw. Eng.*, 28:146–158, February 2002.
[22] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle monitors by collecting facts and generating obligations. In *Tech. Rep.*, 2003.
[23] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From eagleto ruler. In *Runtime Verification, ser. Lecture Notes in Computer Science*, 2007.
[24] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24:129–155, Mar 2004.
[25] F. Chen, M. DAmorim, and G. Rosu. A formal monitoring-based framework for software development and analysis. In *the 6th International Conference on Formal Engineering Methods (ICFEM04)*, 2004.
[26] Y. Jiang, H. Song, R. Wang, M. Gu, J. Sun, and L. Sha. Data-centered runtime verification of wireless medical cyber-physical system. *IEEE Transactions on Industrial Informatics*, PP(99):1–1, 2016.
[27] Yu Jiang, Han Liu, Hui Kong, Rui Wang, Mohammad Hosseini, Jiaguang Sun, and Lui Sha. Use runtime verification to improve the quality of medical care practice. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 112–121, New York, NY, USA, 2016. ACM.
[28] F. Chen and G. Rosu. Java-mop: A monitoring oriented programming environment for java. In *11th International Conference, TACAS*, 2005.
[29] Wikipedia. Laser surgery. https://en.wikipedia.org/wiki/Laser_surgery, 2016.
[30] Cheolgi Kim, Mu Sun, Sibin Mohan, Heechul Yun, Lui Sha, and Tarek F. Abdelzaher. A framework for the safe interoperability of medical devices in the presence of network failures. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 149–158, New York, NY, USA, 2010. ACM.
[31] Javamop4. http://fsl.cs.illinois.edu/index.php/JavaMOP4.