Pattern-Based Statechart Modeling Approach for Medical Best Practice Guidelines - A Case Study

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

Maryam Rahmaniheris, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{rahmani1, lrs}@illinois.edu

***Abstract*—Improving effectiveness and safety of patient care is an ultimate objective for medical cyber-physical systems. Many medical best practice guidelines exist in the format of hospital handbooks which are often lengthy and difficult for medical staff to remember and apply clinically. Statechart is an effective tool to model medical guidelines and enables clinical validation with medical staffs. However, some advanced statechart elements could result in high cost, such as low understandability, high difficulty in clinical validation, formal verification, and failure trace back. The paper presents 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 practical use, we implement the proposed approach based on open-source Yakindu statecharts. We also use a simplified cardiac arrest scenario provided to our team by Carle Foundation Hospital as a case study to validate the proposed approach.**

I. Introduction and Related Work

Medical best practice guidelines play an important role in medical care. Over the past decade, many text-based best practice guidelines have been represented and encoded into computer interpretable formats, such as Asbru [1], GLIF [2], and PROforma [3], to name a few. Most of the encodings are similar to the format of executable pseudo code which requires medical professionals to have some computer coding knowledge to understand. Furthermore, those formats are not visual nor user friendly for physicians to validate their correctness, especially for complicated clinical cases.

In most of today's hospital handbooks, flowcharts are often used to represent medical best practice guidelines [4]. These flowcharts and many medical disease and treatment models are very similar to statecharts [5]. In addition to the high similarities between medical models and statecharts, statecharts are executable and can be indirectly verified, and hence have become a widely used model in designing complex systems, such as medical systems [6], [7], [8], avionics [9], and air traffic control systems [10]. These distinguishing features of statechart inspire us to use it as a computerized representation for medical best practice guidelines.

Since the concept of statecharts was proposed by Harel [5], many variants of statecharts have been proposed, such as UML Statecharts [11], STATEMATE [12], Safe State Machine [13], Stateflow [14], and Yakindu Statecharts [15], to name a few. Yakindu statecharts tool is an open-source tool kit based on the concept of statechart. It has a well-designed user interface, provides simulation and code generation functionalities which enable rapid prototyping and validation with domain experts, and has been applied in real-world applications such as autonomous driving smart cars [16] and Lego Mindstorms robot kits [17].

Both the statecharts definition [5] and most of the statecharts variants contain basic elements, such as *states* and *transitions*, and advanced elements, such as *composite states*. Although the advanced elements are useful to model medical guidelines, the use of advanced elements often requires medical personals to understand both syntax and execution semantics of the advanced elements which can be a challenge for medical professionals with limited computer science knowledge. Furthermore, the advanced elements often introduce unnecessary difficulty in formal verification and tracing back unsatisfied properties. We will further analyze these disadvantages of advanced statechart elements in Section II-C.

Based on National Academies Press and CRC Press, the key to improve system safety at reasonable cost is a serious and sustained commitment to simplicity [18], [19]. For example, it is a standard practice in aviation to forbid complex C++ constructs and the use of dynamic priority and dynamic memory allocation in embedded systems, as these features make certification difficult [20]. Similar philosophy is taken in modern programing language design, where some advanced language constructs are removed to improve safety and reduce complexity. For instance, Java removes pointers and GOTO statement has been avoided in C/C++ since 1960s.

The same principle is applied in modeling safety-critical medical guidelines by avoiding the use of complex modeling constructs. In particular, we implement advanced statechart elements with model patterns which are built upon basic elements. Then we present a pattern-based statechart modeling approach for medical best practice guidelines, i.e., model medical guidelines with basic statechart elements and model patterns which implement advanced elements. For practical use, we implement the approach based on open-source Yakindu statecharts [15], which have a well-designed user interface and provid simulation and code generation functionalities. The main contributions of the paper are:

- We propose a pattern-based statechart modeling approach for medical best practice guidelines;
- We implement the proposed approach based on Yakindu statecharts;
- We validate the developed approach through a simplified cardiac arrest scenario provided by Carle Foundation

The research is supported in part by NSF CNS 1545008 and NSF CNS 1545002.

2372-9198/17 $31.00 © 2017 IEEE
DOI 10.1109/CBMS.2017.14

Hospital.

The paper is organized as follows. In Section II, we use a simplified cardiac arrest scenario to illustrate medical guideline modeling process with statecharts and analyze high cost of advanced statechart elements. Section III presents the pattern-based statechart modeling approach, implements three model patterns based on Yakindu statecharts, and validates the proposed approach by the simplified cardiac arrest case study. We conclude in Section IV.

II. Medical Guideline Modeling with Statecharts

In the section, we use a simplified medical example to illustrate how to model medical guidelines with statecharts and to analyze high cost of the use of advanced statechart elements in medical guideline modeling.

A. Simplified Cardiac Arrest Treatment Case Study

Cardiac arrest is the abrupt loss of heart function which can lead to death within minutes. American Heart Association (AHA) has provided resuscitation guidelines for the urgent treatment of cardiac arrest [4].

In a simplified cardiac arrest treatment scenario [21], 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. To so so, the medical staff need to check a precondition, i.e., whether the EKG (electrocardiogram) monitor shows a shockable rhythm[1]. If the EKG monitor shows a non-shockable rhythm[2]. In order to induce a shockable rhythm, a drug, called epinephrine (EPI), is commonly given to increase cardiac output. To give epinephrine, nevertheless, also has two preconditions: (1) patient's blood pH value should be larger than 7.4[3], and (2) urine flow rate should be greater than 12 mL/s[4]. In order to satisfy 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. Fig. 1 shows a simplified cardiac arrest treatment workflow. In the cardiac arrest treatment procedure, epinephrine (EPI), sodium bicarbonate, and intravenous (IV) fluid are injected using infusion pumps [22].

There are two medical safety properties needed to be verified in the cardiac arrest treatment workflow:

- **P1**: Defibrillator is activated only if the EKG rhythm is shockable;
- **P2**: 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. Simplified Cardiac Arrest Treatment Models with Statecharts

In our previous work, Wu *et al.* developed a validation protocol to enforce the correct execution sequence of performing a treatment regarding preconditions validation, side effects monitoring, and expected responses checking based on the pathophysiological models [21]. In this paper, we use Yakindu statecharts to model the simplified cardiac arrest treatment procedure with the validation protocol, i.e., the *Treatment* statechart shown in Fig. 2(a). In the cardiac arrest treatment procedure, a infusion pump [22] can inject epinephrine (EPI), sodium bicarbonate, and intravenous (IV) fluid. Intuitively, we could use a composite state to model the infusion pump, where the composite state contains three sub-statecharts which represent the epinephrine injector, the sodium bicarbonate injector, and the intravenous fluid injector, respectively. The infusion pump statechart model is shown in Fig. 2(b). Hence, the simplified cardiac arrest statechart model consists of two statecharts: *Treatment* and *Pump*, as shown in Fig. 2. We run simulations on the simplified cardiac arrest statechart model through Yakindu. The simulation results through Yakindu show that both medical properties, i.e., **P1** and **P2**, are satisfied.

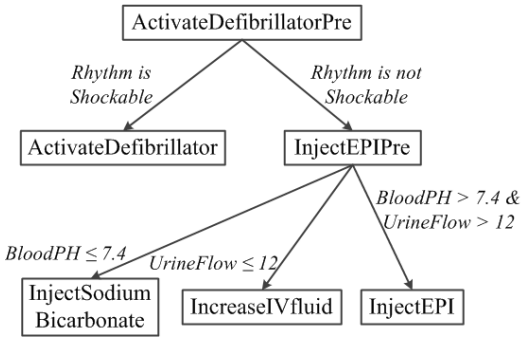


Fig. 1. Simplified Cardiac Arrest Treatment Workflow

We use the Y2U[5] tool [6] to transform the simplified cardiac arrest Yakindu model given in Fig. 2 to UPPAAL timed automata to verify the two medical properties **P1** and **P2**. The transformed simplified cardiac arrest UPPAAL model is shown in Fig. 3. The two medical properties **P1** and **P2** can be checked in UPPAAL by following two formulas:

- **P1**: $A[\,]$`Treatment.ActivateDefibrillaotr` $imply$ `Rhythm` $== 1$;
- **P2**: $A[\,]$`Treatment.InjectEPI` $imply$ $\texttt{BloodPH}_{\texttt{int}} >= 7$ && $\texttt{BloodPH}_{\texttt{frac}} > 4$ && $\texttt{UrineFlow}_{\texttt{int}} > 12$.

The verification results also show that both **P1** and **P2** are always satisfied.

C. Case Analysis

In the simplified cardiac arrest statechart model shown in Fig. 2, the *Treatment* statechart (Fig. 2(a)) is similar to the simplified cardiac arrest treatment workflow shown in Fig. 1 and only uses basic statechart elements *states* and *transitions*. Hence, medical professionals can easily understand and clinically validate the *Treatment* statechart. However, the *Pump* statechart (Fig. 2(b)) uses an advanced statechart element *composite state* to model infusion pumps which can inject multiple medicine fluid. As shown in Fig. 2(b), the composite state named `On` contains three sub-statecharts: *EPI*, *SodiumBicarbonate*, and *IV*. To understand and validate the *Pump* statechart, medical professionals are required to

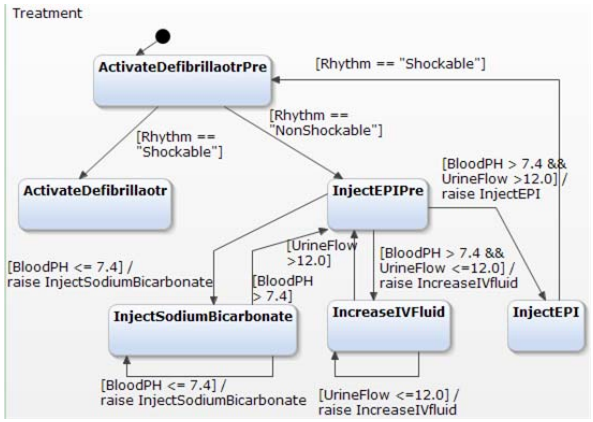
[1]The shockable rhythms are ventricular fibrillation and ventricular tachycardia [4].

[2]Non-shockable rhythms are asystole and pulseless electrical activity [4].

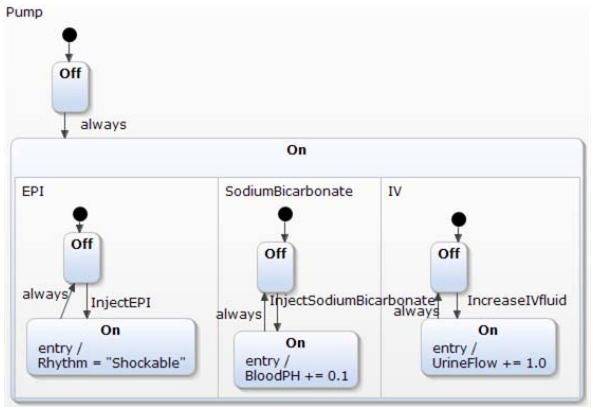
[3]Severe acidosis, which is an increased acidity in the blood and other body tissue, will significantly reduce the effectiveness of epinephrine [4].

[4]If a patient suffers from kidney insufficiency, giving epinephrine may worsen the kidney function and cause acute renal failure [4].

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

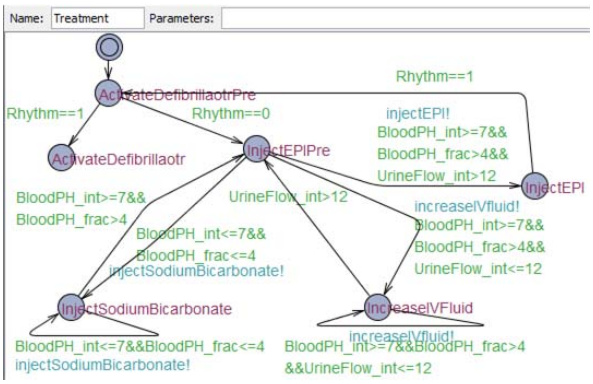


(a) Treatment Model

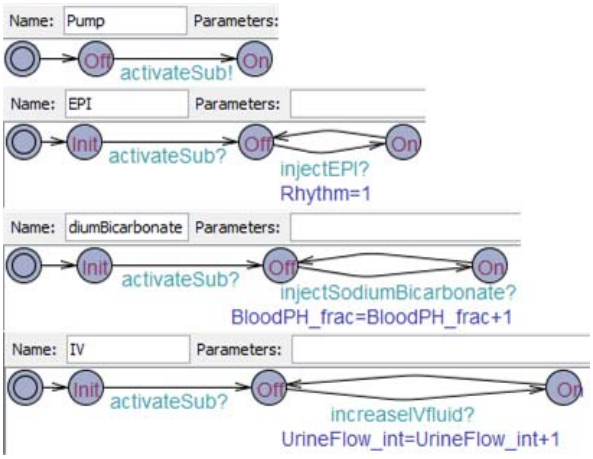


(b) Infusion Pump Model

Fig. 2. Simplified Cardiac Arrest Yakindu Statechart Model



(a) Treatment Model



(b) Infusion Pump Model

Fig. 3. Simplified Cardiac Arrest UPPAAL Model

fully understand the execution semantics of *composite states*: (1) the interaction mechanism between the entire statechart model and sub-statecharts in a composite state, (2) when to activate/deactivate sub-statecharts, (3) the execution orders of main statecharts and sub-statecharts, (4) the interaction mechanism among different sub-statecharts. Hence, the *Pump* statechart is more difficult to understand and clinically validate for medical professionals than the *Treatment* statechart. One main reason is that the *Pump* statechart uses advanced statechart elements which require more computer science knowledge to understand. Noting, a physician from Carle Foundation Hospital stated that even the horizontal graph organization of statechart elements can increase the difficulty of understanding medical guideline statechart models as medical workflows are usually vertical in medical guideline handbooks.

For formal verification purpose, we need to transform medical guideline statecharts to UPPAAL timed automata to formally verify safety properties and trace back failed properties [6]. By comparing the simplified cardiac arrest *Treatment* statechart model shown in Fig. 2(a) with its corresponding transformed UPPAAL model shown in Fig. 3(a), all elements in the two models have one-to-one mapping. Hence, for the *Treatment* statechart, the time complexities of transformation and tracing back failures are both $O(n)$, where n is the sum of states' number and transitions' number. As UPPAAL timed automata does not support *composite states*, the transformation of the *Pump* statechart (Fig. 2(b)) is required to flatten the composite state `On` into three individual automata as shown in Fig. 3(b). According to transformation rules in [6], the time complexities of transformation and tracing back failures of the *Pump* statechart are both $O(n^2)$. Therefore, the *composite state* element can exponentially increase the time complexities of transformation and tracing back failures.

III. Pattern-Based Medical Guideline Modeling

A. Pattern-Based Statechart Modeling Approach

As analyzed in Section II-C, some advanced statechart elements could result in high cost. To avoid these disadvantages, there are two approaches to implement advanced elements in the literature: (1) represent advanced elements by basic elements, such as Esterel [23]; or (2) implement the advanced elements by code directly, such as Yakindu statecharts [15]. Although the first approach uses basic elements to represent advanced elements, the translate process from advanced elements to basic elements is hidden for model developers; whereas the second approach hides all the implementation details. However, the visibility of implementation details is critical for validating the safety of medical cyber-physical systems. We propose an approach to make the translation

process of advanced elements visible to medical professionals through explicitly designed model patterns. The visibility provides a friendly interface between medical staffs and computer professionals, and improves medical guideline models' understandability and quality of clinical validation.

In summary, our pattern-based statechart modeling approach models medical best practice guidelines with basic statechart elements and model patterns which are built upon these basic elements to implement advanced statechart elements. The pattern-based approach not only increases statechart models' understandability for medical professionals, but also reduces the difficulty in clinical validation, formal verification, and failure trace back.

B. Implementation of Advanced Statechart Elements with Model Patterns

For Yakindu statecharts, we implement three model patterns: *composite state*, *state action*, and *choice*.

1) Model Pattern for Composite State:

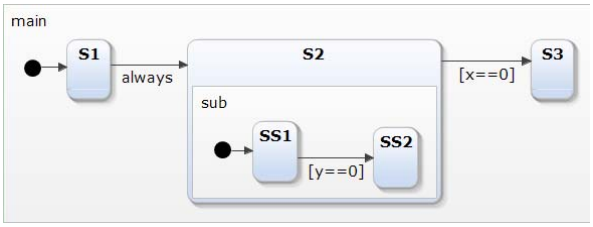
For a composite state, we separate the sub-statechart contained in the composited state from the main statechart containing the composite state, and implement the interactions between the sub-statechart and the main statechart through *events* between them. With the composite state model pattern, a composite state is represented by a simple state. To maintain the interaction between the main statechart and sub-statechart in a composite state, we declare two *events*: `activateSub` for activating/entering the composite state and `deactivateSub` for deactivating/exiting sub-statechart in the composite state.

In the main statechart, we raise two *events* `activateSub` and `deactivateSub` for incoming and outgoing transitions of the composite state, respectively. The sub-statecharts in a composite state are hence separated from the main statechart. The sub-statecharts' execution priorities are set one level lower than the main statechart. For the sub-statechart, we add an initial state $\texttt{SS}_0$ as the successor of the sub-statechart's entry node and add a transition from $\texttt{SS}_0$ to the original first state of the sub-statechart with guard [`activateSub`] to accept the sub-statechart's activation from the main statechart. For each states in the sub-statechart except $\texttt{SS}_0$, we add an outgoing transition, which has the highest priority among all outgoing transitions, to state $\texttt{SS}_0$ with guard [`deactivateSub`]. All transitions of the sub-statechart have lower priorities than each outgoing transition of the composite state, hence allows the main statechart be able to interrupt the sub-statechart's execution at any time.

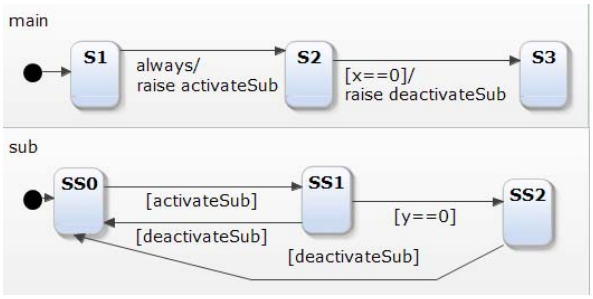
Fig. 4 shows an example of the composite state model pattern. In Fig. 4, the sub-statechart is activated/deactivated when the main statechart enters/exists state `S2`. The activation/deactivation of the sub-statechart is implemented by *event* `activateSub`/`deactivateSub`. Both state `SS1` and state `SS2` has a highest priority outgoing transition to added state `SS0`. Algorithm 1 depicts the implementation of the composite state model pattern.

2) Model Pattern for State Action:

In Yakindu statecharts, the actions can be associated with both transitions (Mealy state machine [24]) and states (Moore state machine [24]). To increase statechart models' understandability for medical professionals, we use the state action model pattern to represent *state actions* by *transition actions*.



(a) Statechart Model with Composite State



(b) Statechart Model with Pattern

Fig. 4. Composite State Model Pattern

Algorithm 1 COMPOSITE STATE PATTERN

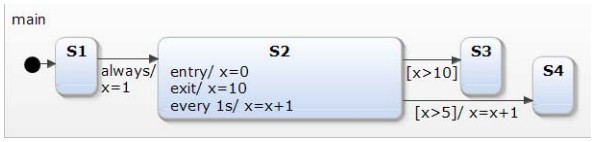
Require: A composite state `S` with incoming transitions $\mathcal{T}^{\mathcal{I}} = \{T_1^I, T_2^I, \ldots, T_m^I\}$ and outgoing transitions $\mathcal{T}^{\mathcal{O}} = \{T_1^O, T_2^O, \ldots, T_n^O\}$, the sub-statechart `Sub` in `S`

1: Declare two events `activateSub` and `deactivateSub`
2: Separate the sub-statechart `Sub` from composite state `S`
3: Replace `S` with a simple state
4: **for** each incoming transition T_i^I in $\mathcal{T}^{\mathcal{I}}$ **do**
5: Add the action *raise activateSub*
6: **end for**
7: **for** each outgoing transition T_j^O in $\mathcal{T}^{\mathcal{O}}$ **do**
8: Add the action *raise deactivateSub*
9: **end for**
10: Add state $\texttt{SS}_0$ as the successor of `Sub`'s entry node
11: Add a transition from $\texttt{SS}_0$ to original first state of `Sub` with guard [`activateSub`]
12: **for** each state in `Sub` except $\texttt{SS}_0$ **do**
13: Add a highest priority outgoing transition to state $\texttt{SS}_0$ with guard [`deactivateSub`]
14: **end for**

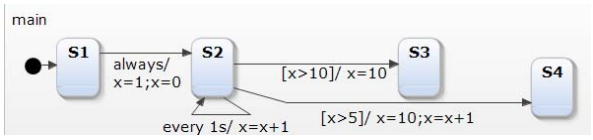
Yakindu statecharts have two types of state actions: *entry/exit* actions and *timer* actions. The *entry/exit* actions are carried out on entering or exiting a state. In the state action model pattern, *entry/exit* actions are combined into actions on all incoming/outgoing transitions of the corresponding state. For each timer action of a state, we add a self-loop transition with lowest priority for the state and represent the timer action by an action on the added self-loop transition.

Fig. 5 shows an example of the state action model pattern. For instance, the state `S2` in Fig. 5(a) has a *timer* action `every 1s/x = x + 1` which means if `S2` is active, x will be increased by 1 for every second. With the state action model pattern, we add a lowest priority self-loop transition with guard `every 1s` and action `x = x + 1` for `S2`, as shown in Fig. 5(b).

Algorithm 2 depicts the implementation of the state action model pattern.



(a) Statechart Model with State Action



(b) Statechart Model with Pattern

Fig. 5. State Action Model Pattern

Algorithm 2 STATE ACTION PATTERN

Require: A state S, S's incoming transitions $\mathcal{T}^{\mathcal{I}} = \{T_1^I, T_2^I, \ldots, T_m^I\}$ with actions $\mathcal{A}^{\mathcal{I}} = \{A_1^I, A_2^I, \ldots, A_m^I\}$, S's outgoing transitions $\mathcal{T}^{\mathcal{O}} = \{T_1^O, T_2^O, \ldots, T_n^O\}$ with actions $\mathcal{A}^{\mathcal{O}} = \{A_1^O, A_2^O, \ldots, A^I O_n\}$, S's *entry* action $A^{\mathtt{en}}$, S's *exit* action $A^{\mathtt{ex}}$, S's *timer* actions $\mathcal{A}^{\mathtt{t}} = \{A_1^t, A_2^t, \ldots, A_l^t\}$, and corresponding *timer* guards $\mathcal{G}^{\mathtt{t}} == \{G_1^t, G_2^t, \ldots, G_l^t\}$,

1: **for** each incoming transition T_i^I in $\mathcal{T}^{\mathcal{I}}$ **do**
2: $A_i^I = A_i^I; A^{\mathtt{en}}$
3: **end for**
4: **for** each outgoing transition T_j^O in $\mathcal{T}^{\mathcal{O}}$ **do**
5: $A_j^O = A^{\mathtt{ex}}; A_j^O$
6: **end for**
7: **for** each timer action A_k^t in $\mathcal{A}^{\mathtt{t}}$ **do**
8: Add a self-loop transition $T^{\mathtt{loop}}$ with lowest priority for state S
9: $G^{\mathtt{loop}} = G_k^t$
10: $A^{\mathtt{loop}} = A_k^t$
11: $\mathcal{T}^{\mathtt{loop}} = \mathcal{T}^{\mathtt{loop}} \cup T^{\mathtt{loop}}$
12: **end for**

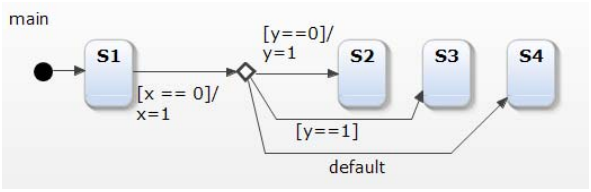
3) Model Pattern for Choice:

In Yakindu statecharts, a *choice* node is a pseudo state which can be used to model a conditional path [25]. It divides a transition into multiple sections, each section can carry a guard and an action.

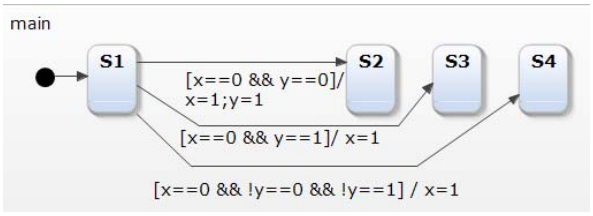
To increase statechart models' understandability for medical professionals, we design the choice model pattern to represent a *choice* node with the basic elements. The model pattern replaces the *choice* node and its incoming and outgoing transitions with added transitions that directly connect the *choice* node's predecessor states with successor states. Each added transition combines one incoming transition and one outgoing transition of the corresponding *choice* node using AND logic. Suppose a *choice* node has m incoming transitions and n outgoing transitions, the choice model pattern will add $m * n$ new transitions and delete $m + n$ original transitions.

Fig. 6 shows an example of the choice model pattern. For instance, the transition from S1 to S2 ($[x == 0 \&\& y == 0]/\ x = 1; y = 1$) is combined by the transition from S1 to the choice node ($[x == 0]/\ x = 1$) and the transition from the choice node to S2 ($[y == 0]/\ y = 1$) with AND logic.

Algorithm 3 depicts the implementation of the choice model pattern.



(a) Statechart Model with *Choice*



(b) Statechart Model with Pattern

Fig. 6. *Choice* Model Pattern

Algorithm 3 CHOICE PATTERN

Require: A *choice* node C, C's incoming transitions $\mathcal{T}^{\mathcal{I}} = \{T_1^I, T_2^I, \ldots, T_m^I\}$ with guards $\mathcal{G}^{\mathcal{I}} = \{G_1^I, G_2^I, \ldots, G_m^I\}$ and actions $\mathcal{A}^{\mathcal{I}} = \{A_1^I, A_2^I, \ldots, A_m^I\}$, and C's outgoing transitions $\mathcal{T}^{\mathcal{O}} = \{T_1^O, T_2^O, \ldots, T_n^O\}$ with guards $\mathcal{G}^{\mathcal{O}} = \{G_1^O, G_2^O, \ldots, G_n^O\}$ and actions $\mathcal{A}^{\mathcal{O}} = \{A_1^O, A_2^O, \ldots, A^I O_n\}$

Ensure: Combined transitions $\mathcal{T} = \{T_1, T_2, \ldots, T_{m*n}\}$ with guards $\mathcal{G} = \{G_1, G_2, \ldots, G_{m*n}\}$ and actions $\mathcal{A} = \{A_1, A_2, \ldots, A_{m*n}\}$

1: **for** each incoming transition T_i^I in $\mathcal{T}^{\mathcal{I}}$ **do**
2: **for** each outgoing transition T_j^O in $\mathcal{T}^{\mathcal{O}}$ **do**
3: Add a combined transition T_k from T_i^I's source state to T_j^O's destination state
4: **if** G_j^O = `default` **then**
5: $G_k = G_i^I$ && !G_1^O && ... && !G_{j-1}^O && !G_{j+1}^O && ... && !G_n^O
6: **else**
7: $G_k = G_i^I$ && G_j^O
8: **end if**
9: $A_k = A_i^I;\ A_j^O$
10: **end for**
11: **end for**
12: Delete the *choice* node C
13: Delete C's incoming transitions $\mathcal{T}^{\mathcal{I}}$ and outgoing transitions $\mathcal{T}^{\mathcal{O}}$
14: **return** $\mathcal{T}$

C. Simplified Cardiac Arrest Models with Model Patterns

We apply the composite state model pattern (Algorithm 1) to the *Pump* statechart in Fig. 2(b). The three sub-statecharts in the composite state On of *Pump* statecharts are extracted out. The *EPI*, *SodiumBicarbonate*, and *IV* statecharts contain state actions. We then apply the state action model pattern (Algorithm 2) to remove these state actions. The modified infusion pump statechart model is shown in Fig. 7. We run simulations on the modified simplified cardiac arrest statechart

model through Yakindu. The simulation results show that both medical properties, i.e., **P1** and **P2**, remain satisfied. We also simulate the execution of the two statechart models under the same environment. The simulation results show that the two statechart models have the same execution behavior, i.e., they are equivalent from execution behavior perspective.

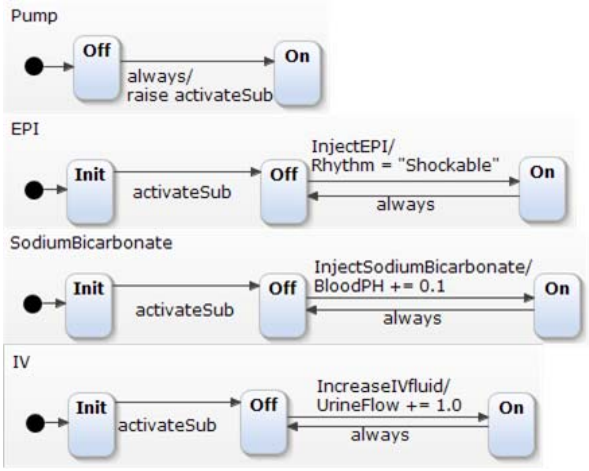


Fig. 7. Infusion Pump Model with Model Patterns

We also use the Y2U tool to transform the modified statechart model to UPPAAL timed automata to verify **P1** and **P2**. The transformed simplified cardiac arrest UPPAAL model is the same with the UPPAAL model in Fig. 3. The verification results show that both **P1** and **P2** are all satisfied.

After applying the proposed composite state model pattern and state action model pattern, the modified infusion pump statechart model shown in Fig. 7 only contains basic statechart elements *states*, *transitions*, and transition actions. Hence, it is more understandable for medical professionals than the *Pump* statechart in Fig. 2(b). In addition, the modified infusion pump statechart model shown in Fig. 7 and its corresponding transformed UPPAAL model shown in Fig. 3(b) have one-to-one mapping elements. Hence, the time complexities of transformation and tracing back failures are both decreased from $O(n^2)$ to $O(n)$.

The case study demonstrates: (1) the composite state model pattern improves understandability of medical guideline statechart models for medical professionals; and (2) the composite state model pattern decreases the time complexities of transformation and tracing back failures from exponential time to linear time.

IV. Conclusion

The paper presents 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 to implement advanced statechart elements. The proposed pattern-based approach not only increases statechart models' understandability for medical professionals, but also reduces the difficulty in clinical validation, formal verification, and failure trace back. For practical use, we implement the proposed approach based on open-source Yakindu statecharts. We also use a simplified cardiac arrest scenario provided to our team by Carle Foundation Hospital as a case study to validate the proposed pattern-based approach. It is worth pointing out that although the presented approach is designed for modeling medical guidelines, the approach can also be applied in modeling other safety-critical systems which require both validation with domain experts and formal correctness verification.

References

[1] 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.

[2] 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.

[3] John Fox, Nicky Johns, and Ali Rahmanzadeh. Disseminating medical knowledge: the proforma approach. *Artificial Intelligence in Medicine*, 14(12):157 – 182, 1998. Selected Papers from AIME '97.

[4] Mary Fran Hazinski, Michael Shuster, Michael W. Donnino, et al. 2015 american heart association guidelines update for cardiopulmonary resuscitation and emergency cardiovascular care. *Circulation*, 132(18):S315–S573, November 2015.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[6] 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.

[7] 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.

[8] Maryam Rahmaniheris, Yu Jiang, and Lui Sha. Model-Driven Design of Clinical Guidance Systems. *ArXiv e-prints*, October 2016.

[9] M. Romdhani, A. Jeffroy, P. de Chazelles, A. E. K. Sahraoui, and A. A. Jerraya. Modeling and rapid prototyping of avionics using statemate. In *Rapid System Prototyping, 1995. Proceedings., Sixth IEEE International Workshop on*, pages 62–67, Jun 1995.

[10] Jon Whittle, Richard Kwan, and Jyoti Saboo. From scenarios to code: An air traffic control case study. *Software & Systems Modeling*, 4(1):71–93, 2005.

[11] Michael von der Beeck. *Formalization of UML-Statecharts*, pages 406–421. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[12] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[13] Charles André. Semantics of s.s.m. (safe state machine). I3S Laboratory, University of Nice-Sophia Antipolis / CNRS, 2003.

[14] Stateflow. http://www.mathworks.com/products/stateflow/.

[15] Yakindu statechart tools (sct). https://www.itemis.com/en/yakindu/statechart-tools/.

[16] Autonomous driving with the yakindu smart car. http://blog.statecharts.org/2015/02/yakindu-smart-car.html, February 2015.

[17] Yakindu statecharts enter lego mindstorms. http://blog.statecharts.org/2014/11/yakindu-statecharts-enter-lego.html, November 2014.

[18] Daniel Jackson, Martyn Thomas, and Lynette I Millett. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[19] Pascale Carayon. *Handbook of Human Factors and Ergonomics in Health Care and Patient Safety*. CRC Press, 2011.

[20] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[21] 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.

[22] Infusion pumps. http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/.

[23] Grard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[24] M. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1982. Cambridge Books Online.

[25] Yakindu statechart tools documentation. https://www.itemis.com/en/yakindu/statechart-tools/documentation/user-guide/.