# 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 only basic statechart elements and model patterns built upon these basic elements. We 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.**

## 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]. Most of the encodings are similar to the format of executable pseudo code which requires medical staffs 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 problems.

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], [6]. 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 avionics [7], air traffic control systems [8], and medical systems [9]. 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 [10], STATEMATE [11], Safe State Machine [12], Stateflow [13], and Yakindu Statecharts [14], 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.

Both the statecharts definition [5] and most of the statecharts variants contain *primitive* elements, such as *states* and *transitions*, and *advanced* elements, such as *composite states*. These advanced elements are useful to model medical guidelines. For instance, the cardiac arrest guideline involves both cardiovascular organ system and kidney organ system [15]. We could use a composite state to model the organs in the cardiac arrest guideline, where the composite state contains two sub-statecharts which represent the cardiovascular organ and the kidney organ, respectively. However, the use of advanced elements often result in high cost, as it requires medical personals to understand both syntax and execution semantics of the advanced elements, and can be a challenge for them who only have limited computer science knowledge. From computer professionals' perspective, the advanced elements introduce unnecessary difficulty of formal verification and tracing back unsatisfied properties in model debugging.

One key to achieving system safety at reasonable cost is a serious and sustained commitment to simplicity, including simplicity of critical functions and simplicity in system interactions [16]. This commitment is often the mark of true expertise. Advances in technology or development methods will not make simplicity redundant; on the contrary, they will give it greater leverage [17]. Reduce the difficulty of formal verification correctness is a key to succeed in safety critical systems. For example, it is a standard practice in aviation to forbid complex C++ constructs and to forbid the use of dynamic priority and dynamic memory allocation, as they make certification difficult [18]. 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 C/C++ avoids use of GOTO statement since 1960s.

To improve safety of medical cyber-physical systems and reduce the difficulty in both clinical validation and formal verification of medical guideline statechart models, the paper presents a pattern-based statechart modeling approach for medical best practice guidelines, i.e., model medical guidelines with only basic statechart elements and model patterns built upon these basic elements. The main contributions of the paper

are:

- We propose a pattern-based statechart modeling approach for medical best practice guidelines;
- We validate the developed approach through a simplified cardiac arrest scenario provided by Carle Foundation Hospital.

## II. Medical Guideline Modeling with Yakindu Statecharts

### A. Cardiac Arrest Scenario

Cardiac arrest is the abrupt loss of heart function and can lead to death within minutes. In a cardiac arrest scenario [6], 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. Fig. 1 shows a simplified cardiac arrest treatment workflow.

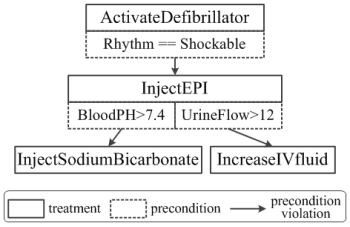



Fig. 1. Simplified Cardiac Arrest Treatment Workflow

There are two medical properties needed to be verified in the simplified cardiac arrest treatment workflow: (1) **P1**: Defibrillator is activated only if the EKG rhythm is shockable and airway and breathing is normal; and (2) **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 Models with Yakindu Statecharts

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 [6]. In this paper, we use Yakindu statecharts to model the simplified cardiac arrest treatment procedure with the validation protocol. The model consists of two statecharts: *Treatment* and *Pump*. The *Treatment* statechart implements the validation protocol and treatment procedure. The *Pump* statechart models infusion pumps [21] to inject epinephrine (EPI), sodium bicarbonate, and intravenous (IV) fluid. The simplified cardiac arrest statechart model is shown in Fig. 2. The simulation results through Yakindu show that both medical properties, i.e., **P1** and **P2**, are satisfied.

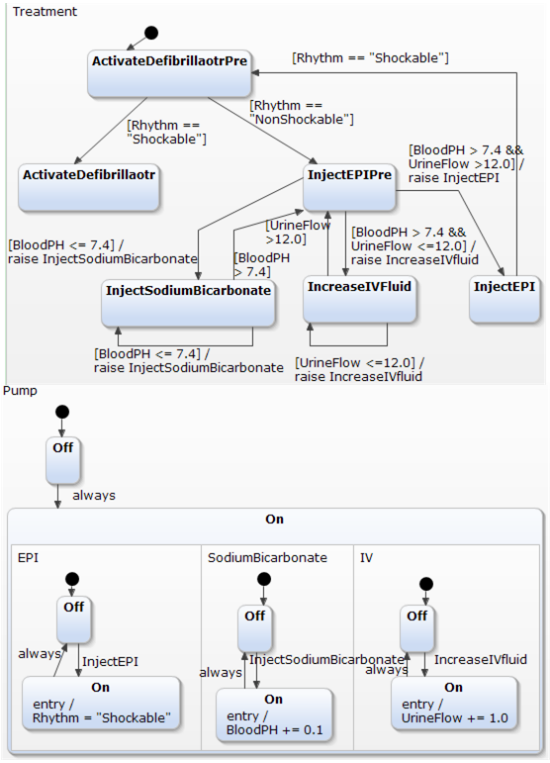



Fig. 2. Simplified Cardiac Arrest Yakindu Model

We use the Y2U[1] tool [9] 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: (1) **P1**: $A[\,]$`Treatment.ActivateDefibrillaotr` $imply$ `Rhythm` $==$ 1 and (2) **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 satisfied.

### C. Case Analysis

Safety is critical to medical cyber-physical systems. To improve safety of medical guideline systems, both clinical validation from medical professionals and formal verification are required. Our previous work [9] proposed an approach to transform medical best practice guidelines to verifiable statechart models and to support both clinical validation in collaboration with medical professionals and formal verification. In particular, the approach uses Yakindu statecharts to model best practice guidelines and use the statechart to interact with doctors for validating the model correctness.

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

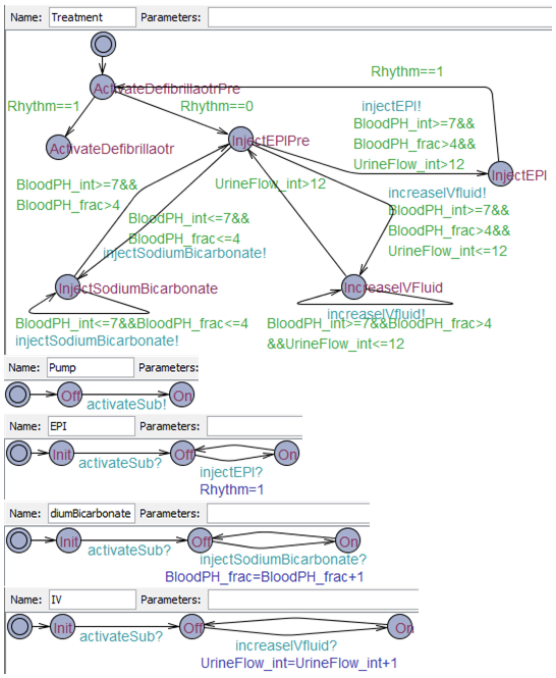



Fig. 3. Simplified Cardiac Arrest UPPAAL Model

The Yakindu statecharts are then automatically transformed UPPAAL timed automata by the presented Y2U tool, so that the model can be formally verified for required safety properties. The approach also provides the ability to trace back to the paths in the Yakindu statecharts when a specific property in its associated UPPAAL timed automata fails. However, our previous work does not address medical guideline modeling issues with statecharts. How to model medical guidelines with statecharts can influence the difficulty of both clinical validation and formal verification. We take the simplified cardiac arrest models in Section II-B as an example to explain the influence of modeling approaches as follows.

Clinical validation requires medical professionals to understand medical guideline statechart models. The modeling approaches can influence understandability of statechart models, hence can affect difficulty of clinical validation. In the simplified cardiac arrest statechart model shown in Fig. 2, the *Treatment* statechart 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 validate the *Treatment* statechart. However, *Pump* statechart uses an advanced statechart element *composite state* to model infusion pumps which can inject multiple medicine fluid. As shown in Fig. 1, 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 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, etc. Hence, the *Pump* statechart is more difficult to understand and 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 mentioned 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.

In our approach presented in [9], the medical guideline modeling approaches with statecharts can influence the time complexities of transformation from Yakindu statecharts to UPPAAL timed automata and tracing back failures. By comparing the simplified cardiac arrest Yakindu model shown in Fig. 2 with the transformed UPPAAL model shown in Fig. 3, all elements of the *Treatment* statechart in two models have one-to-one mapping. Hence, for the *Treatment* statechart, the time complexities of transformation and tracing back failures are both $O(n)$. As UPPAAL timed automata does not support *composite states*, the transformation of the *Pump* statechart is required to flatten the composite state `On`. We proposed a transformation rule to flatten composite states in [9]. According to the transformation rule, 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 mentioned before, statecharts can be used as an effective tool to model medical best practice guidelines. 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*. However, as discussed in Section II-C, some advanced statechart elements could result in high cost, such as low understandability, high difficulty in clinical validation, formal verification, and failure trace back. To overcome these disadvantages of advanced statechart elements, we intend to model medical guidelines with only basic statechart elements.

To fulfill the above intention, we still need to implement advanced statechart elements. In different statecharts variants, there are two approaches to implement advanced elements: (1) represent advanced elements by basic elements, such as Esterel [19]; or (2) implement the advanced elements by code directly, such as Yakindu statecharts [14]. 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. While 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. Hence, 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, hence reduces system failure rate due to hidden implementation rules.

In summary, our pattern-based statechart modeling approach uses only basic statechart elements and model patterns built upon these basic elements to model medical best practice guidelines. 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. Model Patterns

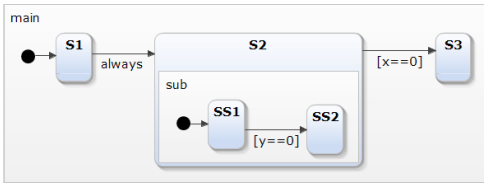
For Yakindu statecharts, we implement three model patterns: *composite state*, *state action*, and *choice*.

*1) Composite State:*

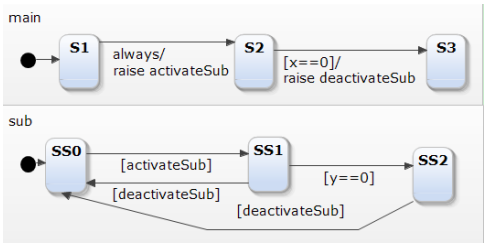
For a composite state, we use a model pattern to flat the hierarchical structure and represent it with basic elements. In particular, we separate the sub-statechart contained in the composited state from the main statechart containing the composite state, and implement the interactions between the sub-statechart and the main statechart through *events* between them. If a statechart model contains nested composite states, we flat it recursively starting from the out most composite state.

In 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.




(a) Statechart Model with Composite State




(b) Statechart Model with Pattern

Fig. 4. Composite State Model Pattern

**Algorithm 1** COMPOSITE STATE PATTERN

**Input:** 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**

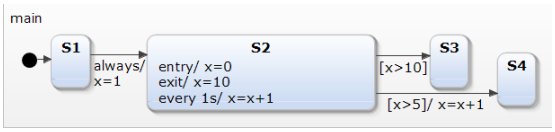
*2) State Action:*

In Yakindu statecharts, the actions can be associated with both transitions and states. We use the state action model pattern to represent *state actions* by *transition actions* which are selected as basic elements.

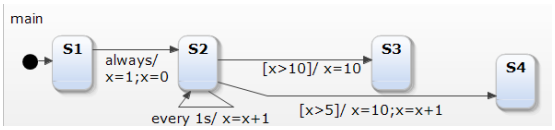
Yakindu statecharts have two types of state actions: *entry/exit* actions and *timer* actions. The *entry/exit* actions are carried out on entering or exiting a state. In the state action model pattern, *entry/exit* actions are combined into actions on all incoming/outgoing transitions of the corresponding 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 Actions




(b) Statechart Model with Pattern

Fig. 5. State Action Model Pattern

**Algorithm 2** STATE ACTION PATTERN

**Input:** 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 $\mathcal{G}^{\mathtt{t}} == \{G_1^t, G_2^t, \ldots, G_l^t\}$,

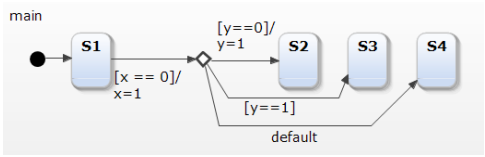
1: **for** each incoming transition $T_i^I$ in $\mathcal{T}^{\mathcal{I}}$ **do**
2: $A_i^I = A_i^I; A^{\mathtt{en}}$
3: **end for**
4: **for** each outgoing transition $T_j^O$ in $\mathcal{T}^{\mathcal{O}}$ **do**
5: $A_j^O = A^{\mathtt{ex}}; A_j^O$
6: **end for**
7: **for** each timer action $A_k^t$ 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) Choice:*

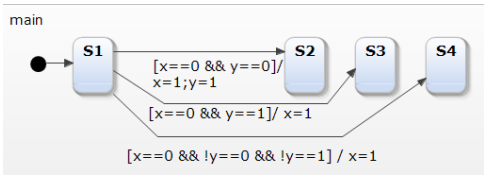
In Yakindu statecharts, a *choice* node is a pseudo state which can be used to model a conditional path [20]. It divides a transition into multiple sections, each section can carry a guard and an action.

To implement *choice* element, 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

**Input:** 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 $\mathcal{A}^{\mathcal{O}} = \{A_1^O, A_2^O, \ldots, A^I O_n\}$

**Output:** 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 (Section III-B1) to the *Pump* statechart in Fig. 2. The three sub-statecharts in the composite state `On` of *Pump* statecharts are extracted out. Hence, the modified infusion pump statechart model contains four statecharts, as 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**, are still 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.

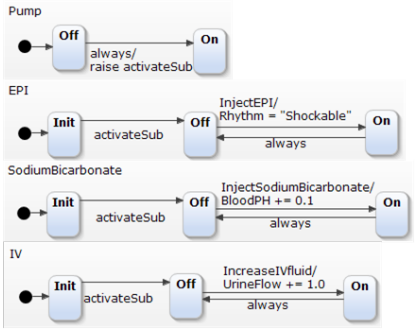



Fig. 7. Infusion Pump Model with Model Patterns

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 still satisfied.

After applying the proposed composite state model pattern, the modified infusion pump statechart model shown in Fig. 7 only contains basic statechart elements *states* and *transitions*. Hence, it is more understandable for medical professionals than the *Pump* statechart in Fig. 2. In addition, the modified pump statechart model shown in Fig. 7 and its corresponding transformed UPPAAL model shown in Fig. 3 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 can improve understandability of medical guideline statechart models for medical professionals; and (2) the composite state model pattern can decrease 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 only basic statechart elements and model patterns built upon these basic 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. We 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.

## Acknowledgment

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

## 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] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

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

[15] 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.

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

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

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

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

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

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