Transforming Medical Best Practice Guidelines to Executable and Verifiable Statechart Models

Chunhui Guo¹, Shangping Ren¹, Yu Jiang², Po-Liang Wu², Lui Sha², and Richard Berlin²,³
Email: cguo13@hawk.iit.edu, ren@iit.edu, {jy1989, wu87, lrs}@illinois.edu, Richard.Berlin@carle.com

¹Illinois Institute of Technology
²University of Illinois at Urbana-Champaign
³Carle Foundation Hospital

ICCPS
April 14, 2016
Outline

1. Introduction
2. Transforming Yakindu Statechart to UPPAAL Timed Automata
3. Trace Failed UPPAAL Properties back to Yakindu Statecharts
4. Cardiac Arrest Case Study
5. Conclusion
Outline

1. Introduction
2. Transforming Yakindu Statechart to UPPAAL Timed Automata
3. Trace Failed UPPAAL Properties back to Yakindu Statecharts
4. Cardiac Arrest Case Study
5. Conclusion
Medical Guidelines

- Medical best practice guidelines play an important role in improving effectiveness and safety of medical care.

- The existing medical best practice guidelines in hospital handbooks are often lengthy and difficult for medical staff to remember and apply clinically.

- The text-based best practice guidelines are represented and encoded into many computer interpretable formats, such as Asbru, GLIF, PROforma, etc.

- However, those formats are not user friendly for physicians to validate their correctness.

- Furthermore, it is not easy to formally verify those formats for life-critical medical cyber-physical systems.
Statechart has several advantages:

- have high similarity to disease models and treatment models
- allow quick clinical validations with medical doctors
- executable
- enable rapid prototyping
- widely used in modeling complex systems

However, clinical validation is often not adequate for guaranteeing the correctness and safety of medical guideline models, and formal verification is required.
Statecharts

Statechart has several advantages:

- have high similarity to disease models and treatment models
- allow quick clinical validations with medical doctors
- executable
- enable rapid prototyping
- widely used in modeling complex systems

However, clinical validation is often not adequate for guaranteeing the correctness and safety of medical guideline models, and formal verification is required.
We propose an approach that transforms medical best practice guidelines to verifiable statechart models and supports both clinical validation and formal verification.
Why choose Yakindu statechart tool to model medical best practice guidelines?

1. The Yakindu statechart tool has a well-designed user interface and has simulation and code generation functionality, which enables rapid prototyping and validation with medical staff.

2. Yakindu statechart tool is an open-source tool kit, which can be customized according to medical domain knowledge.
Guidelines to Verifiable Statecharts

Why choose Yakindu statechart tool to model medical best practice guidelines?

1. The Yakindu statechart tool has a well-designed user interface and has simulation and code generation functionality, which enables rapid prototyping and validation with medical staff.

2. Yakindu statechart tool is an open-source tool kit, which can be customized according to medical domain knowledge.

Why choose UPPAAL to verify guideline statecharts?

1. The structure of UPPAAL time automata and Yakindu statechart are similar, which makes the transformation and trace back easier.

2. UPPAAL has a graphical user interface and has simulation and counter example generation functionality, which advances model debug.
Why not encode best practice guidelines to UPPAAL timed automata directly?

1. According to discussion with medical staff, Yakindu is easier for them to understand and use.

2. UPPAAL does not provide code generation functionality, therefore, system designers must manually translate timed automata to executable code, which is error-prone.
Guidelines to Verifiable Statecharts

- Due to syntax and semantics differences, to bridge the gap between Yakindu and UPPAAL models is a challenge.

- Our strategies are to build transformation rules for each element used in the two models and use these rules as the foundation for the transformation.
Due to syntax and semantics differences, to bridge the gap between Yakindu and UPPAAL models is a challenge.

Our strategies are to build transformation rules for each element used in the two models and use these rules as the foundation for the transformation.

We develop the Y2U tool to transform the Yakindu statechart to UPPAAL timed automata and to trace back failed medical properties.

The tool is available on
www.cs.iit.edu/~code/software/Y2U.
1. Introduction

2. Transforming Yakindu Statechart to UPPAAL Timed Automata

3. Trace Failed UPPAAL Properties back to Yakindu Statecharts

4. Cardiac Arrest Case Study

5. Conclusion
Differences between Yakindu statecharts and UPPAAL timed automata:

1. Syntax: they have different syntactic element sets

2. Structure: Yakindu supports hierarchical structure, while UPPAAL only supports flat structure

3. Execution Semantics: Yakindu model is deterministic and has synchronous execution semantics while the execution of UPPAAL model is non-deterministic and asynchronous

4. Simultaneous Events: Yakindu supports simultaneous events while UPPAAL model does not
Transformation Principles

- **Principle I**: must have equivalent execution semantics
- **Principle II**: must maintain Yakindu syntactic elements when possible
- **Principle III**: should introduce minimal additional elements from the Yakindu model
Transformation Principles

- **Principle I**: must have equivalent execution semantics
- **Principle II**: must maintain Yakindu syntactic elements when possible
- **Principle III**: should introduce minimal additional elements from the Yakindu model

**Principle I** ensures that verification results from UPPAAL hold in the Yakindu model.

**Principle II** and **Principle III** ensure that an execution path in UPPAAL model can be traced back to the Yakindu model with reduced complexity.
Transformation Rules

Transformation rules to bridge four type differences between Yakindu statecharts and UPPAAL timed automata:

1. Syntax: **Rule 1 to Rule 6** for basic elements of Yakindu statecharts

2. Structure: **Rule 7** to flatten hierarchical structure

3. Execution Semantics: **Rule 8 to Rule 9** to model determinism and synchrony with UPPAAL

4. Simultaneous Events: **Event Stack** to support simultaneous events in UPPAAL
Transformation Rules: Syntax

- Rule 1: State
- Rule 2: Transition
- Rule 3: Data Type
- Rule 4: Event
- Rule 5: Timing Trigger
- Rule 6: State Action
Rule 7: Composite State

- Transformation Rules: Structure

[Diagram showing states and transitions]
Yakindu statecharts execution semantics:

- Internal Determinism of an Automaton: transition priority (Rule 8)
- External Determinism among Automata: automaton priority (Rule 9)
- Synchrony: synchronous execution (Rule 9)
Rule 8: Transition Priority

- Adjust transition guard to simulate transition priorities
Rule 8: Transition Priority

- Adjust transition guard to simulate transition priorities

Rule 9: Automaton Priority and Synchrony

- Use the lockstep method to force synchronous execution based on automaton priorities.
- Each automaton is associated with an integer to indicate how many steps the automaton has executed. (How to solve out-of-range problem of step indicators?)
- Add an additional guard on execution step indicator to each transition. (How to solve deadlock problem caused by additional guards?)
Transformation Rules: Simultaneous Events

Event Stack

1. An automaton can raise and accept multiple events at the same time.
2. Multiple automata can accept the same event concurrently.
3. An event can only be accepted by automata with lower priorities than the automaton which raise the event.
4. An event can only be accepted in the same execution time cycle in which it is raised.
Transformation Example

To transform a Yakindu model, we may combine multiple transformation rules.
We define a Yakindu model and its associated transformed UPPAAL model are equivalent if their observable executions are equivalent, i.e.,

1. the two models have the same execution path if the input settings are the same

2. all variable values at each execution step of the two models are equal
We define a Yakindu model and its associated transformed UPPAAL model are equivalent if their observable executions are equivalent, i.e.,

1. the two models have the same execution path if the input settings are the same
2. all variable values at each execution step of the two models are equal

**Theorem**

The UPPAAL model transformed from a given Yakindu model by applying **Rule 1 to Rule 9** maintains the Yakindu model’s execution behaviors.
Outline

1. Introduction
2. Transforming Yakindu Statechart to UPPAAL Timed Automata
3. Trace Failed UPPAAL Properties back to Yakindu Statecharts
4. Cardiac Arrest Case Study
5. Conclusion
Rule 4 and Rule 5 add auxiliary event automata and timer automata to the UPPAAL model.
Rule 4 and Rule 5 add auxiliary event automata and timer automata to the UPPAAL model.

However, the added automata does not affect the model’s execution behaviors.
Rule 4 and Rule 5 add auxiliary event automata and timer automata to the UPPAAL model.

However, the added automata does not affect the model’s execution behaviors.

Hence, we ignore these added events and timer automata when tracing back execution path from UPPAAL to Yakindu.
Theorem

Given a Yakindu model $Y$ and its transformed UPPAAL model $U$ (with auxiliary event and timer automata being removed), the mapping from UPPAAL state set $S_U$ to Yakindu state set $S_Y$ is bijective.
Given a Yakindu model $Y$ and its transformed UPPAAL model $U$ (with auxiliary event and timer automata being removed), the mapping from UPPAAL transition set $T_U$ to Yakindu transition set $T_Y$ is surjective, but not injective.
Theorem

Given a Yakindu model $Y$ and its transformed UPPAAL model $U$ (with auxiliary event and timer automata being removed), the mapping from UPPAAL transition set $T_U$ to Yakindu transition set $T_Y$ is surjective, but not injective.

- Rule 6, Rule 7, and Rule 9 add transitions into $T_U$. 
**Theorem**

Given a Yakindu model $Y$ and its transformed UPPAAL model $U$ (with auxiliary event and timer automata being removed), the mapping from UPPAAL transition set $T_U$ to Yakindu transition set $T_Y$ is surjective, but not injective.

- However, our analysis shows that the back trace of the added transitions can be ignored.
Trace Back Procedure

1. Delete the additional event automata added by Rule 4 and timer automata added by Rule 5 in the given UPPAAL execution path.

2. For each state in the path, find its corresponding state in Yankindu model.

3. For each transition in the path, find its corresponding transition in Yankindu model; if the corresponding transition is not found in the Yankindu model, ignore the transition.
Example

Given a Yakindu model whose transformed UPPAAL model is shown below, verify the property $A[\ ] T2. D2 \implies x > 0$. 

The property is not satisfied.
Trace Back Example

Example

Given a Yakindu model whose transformed UPPAAL model is shown below, verify the property $A[] T_2.D_2 \implies x > 0$.

The property is not satisfied.
Example

Counter Example:

\[(A_1, A_2, A_3) \rightarrow (A_1, B_2, A_3) \overset{\text{true}}{\rightarrow} (A_1, C_2, A_3) \rightarrow (A_1, D_2, A_3)\]

Yakindu Model Fix:

the action of transition \(A_2 \rightarrow B_2\) to \(y := 7, x := y - 6\)
Example

Counter Example:

\[(A_1, A_2, A_3) \rightarrow (A_1, B_2, A_3) \xrightarrow{\text{true}} (A_1, C_2, A_3) \rightarrow (A_1, D_2, A_3)\]

Yakindu Model Fix:

the action of transition \( A_2 \rightarrow B_2 \) to \( y := 7, x := y - 6 \)

The property still does not hold.
Example

Counter Example: \((A_1, A_2, A_3) \rightarrow (A_1, A_2, B_3) \rightarrow (B_1, A_2, B_3) \rightarrow (B_1, B_2, B_3) \xrightarrow{\text{syn}} (C_1, C_2, B_3) \rightarrow (C_1, D_2, B_3)\)

Yakindu Model Fix:
the action of transition \(B_1 \rightarrow C_1\) to \(z := 6, x := z - 5\)
Trace Back Example (cont.)

Example

Counter Example: 

\[(A_1, A_2, A_3) \rightarrow (A_1, A_2, B_3) \rightarrow (B_1, A_2, B_3) \rightarrow (B_1, B_2, B_3) \xrightarrow{\text{syn}} (C_1, C_2, B_3) \rightarrow (C_1, D_2, B_3)\]

Yakindu Model Fix:

the action of transition \(B_1 \rightarrow C_1\) to \(z := 6, x := z - 5\)

The property now is satisfied. In this example, two iterations are taken to fix the errors in the model.
Outline

1. Introduction
2. Transforming Yakindu Statechart to UPPAAL Timed Automata
3. Trace Failed UPPAAL Properties back to Yakindu Statecharts
4. Cardiac Arrest Case Study
5. Conclusion
A simplified cardiac arrest treatment scenario is used as a case study to validate the proposed medical guidelines transforming approach.

- Use Yakindu to model the simplified cardiac arrest treatment procedure.
- Inject an error into the Yakindu model.
- Transform the model built with Yakindu to UPPAAL model with the Y2U tool.
- Verify two medical properties in the transformed UPPAAL model. One property is satisfied, the other one is failed.
Trace the failed property back to the Yakindu model and modify the Yakindu model.

Re-transform the modified Yakindu model and re-verify the two medical properties. Both properties are satisfied.
Figure: Cardiac Arrest Treatment Yakindu Model
Figure: Cardiac Arrest Treatment UPPAAL Model
Outline

1. Introduction

2. Transforming Yakindu Statechart to UPPAAL Timed Automata

3. Trace Failed UPPAAL Properties back to Yakindu Statecharts

4. Cardiac Arrest Case Study

5. Conclusion
Conclusion

- The paper presents an approach to transform medical best practice guidelines to executable and verifiable statechart models.

- We develop the Y2U tool to transform a statechart model to a verifiable formal model.

- The designed transformation rules not only maintains the execution equivalence between the statechart model and the formal model, but also allow easy trace back of failed medical properties to the statechart model.

- A simplified cardiac arrest treatment scenario is used as a case study to validate the proposed approach.

- The tool is available on www.cs.iit.edu/~code/software/Y2U.
We thank Mohammad Hosseini and Maryam Rahmaniheris for their valuable suggestions to the Y2U tool.

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

Thank You
Transformation Rules: Syntax (1)

- **Rule 1: State**
  - \(state \rightarrow state\)

- **Rule 2: Transition**
  - \(transition \rightarrow transition\)
Transformation Rules: Syntax (1)

- **Rule 1: State**
  - $state \longrightarrow state$

- **Rule 2: Transition**
  - $transition \longrightarrow transition$

- **Rule 3: Data Type**
  - Real Number: two integers to store its integer part and fraction part, respectively
  - String: an integer variable with a dictionary
Rule 4: Event
Rule 5: Timing Trigger

Figure: Every Timer Transformation

Figure: After Timer Automaton
Rule 6: State Action

```c
void updateInS2(int& x) {
    if (x>0) { x=0; push(2); } }
void updateOutS2(int& x) {
    if (x<10) { push(1); x=10; } }
void updateEvery1sS2(int& x) {
    if (x>=0) { x=x+1; } }
```
Rule 8: Transition Priority

- A state has $n$ outgoing transitions \{ $T_1, T_2, \ldots, T_n$ \} sorted in non-increasing priority order.
- The original guard of transition $T_i$ is denoted as $G_i$.
- The transition guard for $T_i$ is adjusted to be $G_i \&\& !G_1 \&\& !G_2 \&\& \ldots \&\& !G_{i-1}$.
Rule 9: Automaton Priority and Synchrony

- Use the lockstep method to force synchronous execution based on automaton priorities.
- Suppose a model contains $n$ automata $\{A_1, A_2, \ldots, A_n\}$ that are sorted by its execution priority in decreasing order.
- Each automaton $A_j$ is associated with an integer $I_j$ to indicate how many steps $A_j$ has executed.
- For each state in $A_j$, we add a self-loop transition which is guarded by the negation of all existing outgoing transition guards of the state.
**Rule 9: Automaton Priority and Synchrony** (cont.)

- For each transition in $A_j$, we add an additional guard on execution step indicator $I_j$, which is conjuncted with the existing guard to force synchronous execution.
  
  - $A_1: I_1 == I_2 \&\& I_2 == I_3 \&\& \ldots \&\& I_{n-1} == I_n$
  
  - $A_j (j > 1): I_j < I_{j-1}$

- $I_j$ is increased by 1 when $A_j$ executes one step.

- When the automaton with the lowest priority executes a transition, it clears all events for current time cycle.