Transforming Medical Best Practice Guidelines to Executable and Verifiable Statechart Models

Chunhui Guo, Shangping Ren
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
Email: cguo13@hawk.iit.edu, ren@iit.edu

Yu Jiang, Po-Liang Wu, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {jy1989, wu87, lrs}@illinois.edu

Richard B. Berlin Jr.
CS Dept., UIUC and
Carle Foundation Hospital
Urbana, IL 61801, USA
Email: Richard.Berlin@carle.com

***Abstract*—Improving effectiveness and safety of patient care is an ultimate objective for medical cyber-physical systems. However, the existing medical best practice guidelines in hospital handbooks are often lengthy and difficult for medical staff to remember and apply clinically. Statechart is a widely used model in designing complex systems and enables rapid prototyping and clinical validation with medical doctors. However, clinical validation is often not adequate for guaranteeing the correctness and safety of medical cyber-physical systems, and formal verification is required. The paper presents an approach that transforms medical best practice guidelines to verifiable statechart models and supports both clinical validation in collaboration with medical doctors and formal verification. In particular, we use an open source statechart tool Yakindu to model best practice guidelines and use the statechart to interact with doctors for validating the model correctness. The statechart model is then automatically transformed to a verifiable formal model, such as timed automata, so that existing formal verification tool, such as UPPAAL, can be used to verify required safety properties. The approach also provides the ability to trace back to the paths in the statechart model (Yakindu model) when a specific property in its associated formal model (UPPAAL model) fails. A cardiac arrest scenario is used as a case study to validate the proposed approach. The tool is available on our website www.cs.iit.edu/∼code/software/Y2U.**

I. Introduction and Related Work

Medical best practice guidelines play an important role in today's medical care. More and more guidelines are encoded into computer-interpretable formats, based on which, decision supporting systems are developed to monitor actions and provide medical staff with appropriate suggestions. However, how to encode best practice guidelines correctly is still a challenge in medical cyber-physical system design.

Over last decade, text-based best practice guidelines are represented and encoded into many computer interpretable formats, such as Arden [10], GLIF [14] , and PROforma [7]. Then, decision supporting systems such as Spock [19] are developed to monitor actions and provide medical staff with appropriate suggestions. Most of the encodings are similar to the format of executable pseudo code, which is a bit low level for medical staffs. However, many clinical problems are complicated and those formats are not visual nor user friendly for physicians to validate their correctness. Furthermore, it is not easy to formally verify those formats for a rigorous correctness requirement of life-critical medical cyber-physical systems.

Noting that statechart is very similar to disease model and treatment model, and is executable and can be indirectly verified, we try to encode those guidelines to statechart. Statechart is a widely used model in designing complex systems, such as automobile, avionics, and medical systems [9]. Yakindu statechart tool is an open-source tool kit based on the concept of statecharts and has been applied in real-world applications such as autonomous driving smart cars [1] and Lego Mindstorms robot kits [3]. It has a well-designed user interface and simulation and code generation functionality and hence enables rapid prototyping and validation with domain experts. With this, medical staffs will understand the design more easily, validate the design model through user-friendly simulation, and give more meaningful suggestions to the model of the best practice guidelines.

However for life-critical medical cyber-physical systems, validation by medical staff alone is not adequate for guaranteeing safety, and formal verification such as model checking is required. Formal model based approach is appealing because it provides a unified basis for formal analysis to achieve the expected level of correctness and safety guarantees. It has been applied in many safety-critical areas such as automotive and aviation [11], [12]. Unfortunately, Yakindu does not provide formal verification capability. To bridge the gap between system modeling and formal verification, efforts are made from research community to transform system modeling specifications/languages, such as UML (unified modeling language) statecharts [13], [20], hierarchical timed automata (HTA) [5], discrete event system specification for real-time (RT-DEVS) [8], and parallel object-oriented specification language (POOSL) [18], to UPPAAL timed automata. Yakindu statecharts are similar to UML statecharts, but have major semantics differences [2]. For example, the execution semantics of Yakindu statecharts is cycle driven, while UML statecharts is event driven. Therefore, existing tools can not be directly applied to transform Yakindu statecharts to UPPAAL timed automata. Moreover, most of existing work focus on model translation without considering how failed properties can be traced back to the statechart model to correct design specifications.

The paper presents an approach that transforms medical best practice guidelines to verifiable statechart models and supports both clinical validation in collaboration with medical doctors and formal verification, as shown in Fig. 1. In particular, we use Yakindu statechart to model best practice guidelines and use the statechart to interact with doctors for validating the model correctness. The Yakindu statechart is then automatically transformed UPPAAL timed automata by the presented Y2U tool (www.cs.iit.edu/∼code/software/Y2U), 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 statechart when a specific property in its associated UPPAAL timed automata fails. The reason that we do not encode best practice guidelines to timed automata directly is that (1) according to discussion with physicians, Yakindu is easier for them to understand, (2) the disadvantages of model checking tools, such as UPPAAL [4], is that it does not provide code generation functionality, therefore, system designers must manually translate formal models, such as the timed automata, into an executable model or executable computer language source code, which is known to be error-prone. As both the syntax and semantics of Yakindu and UPPAAL are different, 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.

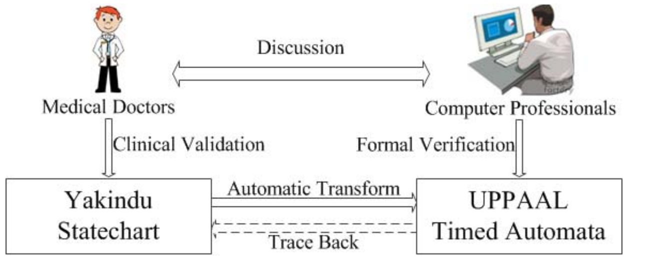


Fig. 1. Transforming Medical Best Practice Guidelines to Verifiable Statechart

The main contributions of the paper are:

- To the best of our knowledge, this is the first study on transferring best practice guideline into statechart for simulation and verification.
- We develop a tool to translate the Yakindu statechart to UPPAAL timed autoamta for verification and to trace back for model property debug.

The paper is organized as follows. Section II describes the transformation from Yakindu statecharts to UPPAAL timed automata. We trace failed UPPAAL properties back to Yakindu to correct the model in Section III. A case study of a simplified cardiac arrest treatment scenario is performed in Section IV. We conclude in Section V.

II. Transforming Yakindu Statechart to UPPAAL Timed Automata

To bridge the gap between Yakindu model and UPPAAL model, we develop a tool, called Y2U, to automatically transform Yakindu statecharts to UPPAAL timed automata. In this section, we first highlight the major syntactic and semantic differences between Yakindu statecharts and UPPAAL timed automata and transformation principles. We then define the transformation rules that meet the principles.

A. Transformation Principles

Yakindu statecharts and UPPAAL timed automata have three key differences: (1) syntactic difference: they have different syntactic element sets, for instance, *event*, *entry*/*exit* trigger are elements in Yakindu statecharts and they do not have counter parts in UPPAAL automata (2) structural difference: Yakindu supports hierarchical structure, while UPPAAL only supports flat structure; and (3) execution semantics difference: Yakindu model is deterministic and has synchronous execution semantics while the execution of UPPAAL model is non-deterministic and asynchronous. In addition, Yakindu supports simultaneous events while UPPAAL model does not. There may be many equivalent transformations for a given Yakindu model. As the purpose of the Y2U tool is not only to provide a formal verification means for a statechart model, more importantly it is for building executable statecharts for medical best practice guidelines that are provably correct. Hence, being able to track back to which states or transitions that cause a desired property failed in the UPPAAL model is critical. We therefore set the following three transformation principles:

- **Principle I**: transformed UPPAAL model has equivalent execution semantics as the statechart model;
- **Principle II**: transformed UPPAAL model maintains Yakindu syntactic elements when possible;
- **Principle III**: transformed UPPAAL model has 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.

Based on the transformation principles, we define a set of transformation rules. We currently focus on the basic elements supported by Yakindu statecharts. The transformation of syntactic sugar in the Yakindu model, such as *choice*, *junction*, and *history* that are for modeling conveniences and can be implemented by other basic elements, is not considered in this paper.

B. Transformation Rules

With respect to UPPAAL, the Yakindu syntactic elements can be categorized into three groups: (1) semantically equivalent elements, such as states and transitions; (2) semantically overlapping elements, for instance, both Yakindu and UPPAAL support data types, however UPPAAL does not support real numbers and strings, but Yakindu does; and (3) Yakindu specific elements, such as events, timing triggers, state actions, and composite states. The following subsections give the transformation rules for each category which are not supported by UPPAAL. For semantically equivalent elements, the transformation is simple and can be implemented by one-to-one syntax mapping. For semantically overlapping elements, our focus is on the domains differences. The transformations of Yakindu specific elements need additional process. The different execution semantics in Yakindu such as determinism, synchrony, and simultaneous events, needs to be implemented by UPPAAL semantics.

1) Basic Elements:

Rule 1: State

Both Yakindu and UPPAAL have *state* elements, each *state* in Yakindu model is transformed to a unique *state* in UPPAAL model. The *entry* state in Yakindu model is marked as *initial* state in UPPAAL model.

Rule 2: Transition

Both Yakindu and UPPAAL have *transition* elements, each *transition* in Yakindu model is transformed to a unique *transition* in UPPAAL model. If the transition guard contains *event*, we use the UPPAAL function `isEventValid(int event)` defined in Section II-B4 to check if the *event* guard is satisfied. If the transition's actions raise *event*, the function `push(int event)` is called to simulate the *event* raise.

Rule 3: Data Type

Both Yakindu and UPPAAL support boolean value and integers, but UPPAAL does not support real number and string. To represent a real number in UPPAAL we use two integers to store its integer part and fraction part, respectively. We also provide real number's comparisons (such as $>$, $\leq$, etc.) and operations (such as $+$, $-$, etc.). For a string, we represent it by an integer variable with a dictionary which maps integer values to string values.

Rule 4: Event

Yakindu statechart has three types of events: *in event*, *out event*, and *internal event*. Both *in event* (such as push a button) and *out event* are interfaces with human beings, while *internal event* is for the communication among automata in the statechart model. We use the event stack described in Section II-B4 to implement the event raise and event acceptance. For the in/out event, we simulate its occurrence (human action) by an auxiliary event automaton defined as follows. The event automaton only contains one state and has a self-loop to push the event into the event stack. Fig. 2 depicts an example screen shot of the in event transformation.

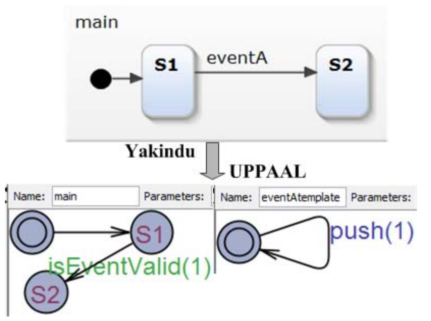


Fig. 2. Event Transformation

Rule 5: Timing Trigger

Yakindu has two types of timers: *every* and *after*. The transformation rules for both types of the timers are similar. We use the `every` $5s$ example shown in Fig. 3 to illustrate the transformation process of *every* timer. We declare a channel (`chan every5s`) and a clock (`clock` t) for the timer. The timer is represented by an input *synchronization* of the channel (`every5s?`). We use an auxiliary automaton to simulate the timer. The timer automaton only contains one state with *invariant* ($t <= 5$) to guarantee that the timer automaton is not allowed to stay in the state for more than 5 time units. The timer automaton also has a self-loop guarded by $t == 5$, which resets the clock ($t := 0$) and synchronizes on the channel `every5s!` with the main automaton.

The transformation of *after* timer is the same with the *every* timer except that the auxiliary timer automaton is different. Instead of a self-loop transition, the *after* timer automaton transits to another state with the same transition settings (guard, synchronization, and update), as shown in Fig. 4.

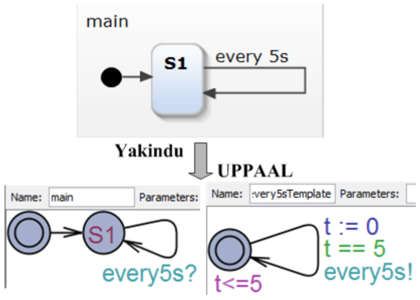


Fig. 3. *Every* Timer Transformation

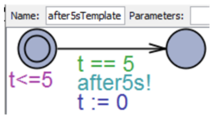


Fig. 4. *After* Timer Automaton

Yakindu supports different time units, such as second, millisecond, microsecond, and nanosecond. To simplify the time values in UPPAAL the Y2U tool supports time unit adaption, rather than always uses nanosecond as the time unit. For example, if the Yakindu model only uses second, then the time unit of UPPAAL model is also second; while if the Yakindu model uses both second and millisecond, then the UPPAAL model uses millisecond as its time unit.

Rule 6: State Action

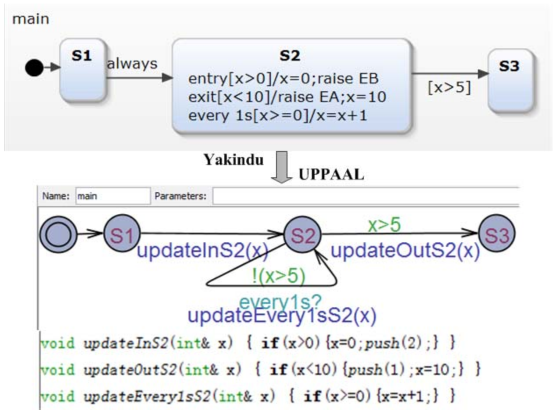
Yakindu has two types of state actions: *entry/exit* actions and *timer* actions, and Yakindu state actions also have guard. We transform state actions into *update* of corresponding transitions and represent the *update* by UPPAAL *functions* to avoid the interference of state action guards and the transition guards. The function `push(int event)` defined in Section II-B4 is called to simulate the *event* raise in state actions. Fig. 5 shows the screen shot of a state action transformation example.

Rule 6.1: *Entry/Exit* Action

The *entry/exit* actions are carried out on entering or exiting a state. They are transformed into *update* on all incoming/outgoing transitions of the state. For the state `S2` in Fig. 5, its *entry* action `entry[x > 0]/x = 0;raise EB` means that when entering `S2`, if x is larger than 0 then x is assigned to 0 and the event `EB` is raised. We transform the *entry* action to the update of transition S1 $\rightarrow$ S2, and use function updateInS2(int &x) to assign x value and call push(2) to push event EB onto the event stack. The *exit* action is transformed similarly.

Rule 6.2: Timer Action

For each timer action of a state, we add a self-loop transition for state. The self-loop transition is guarded by the negations of guards on all existing outgoing transitions, synchronized with the timer automaton (**Rule 5**), and updated with the timer action. For example, the state S2 in Fig. 5 has a *timer* action every 1s[x >= 0]/x = x + 1 which means if S2 is active and $x \geq 0$, x will be increased by 1 for every second. To transform the *timer* action, we add a self-loop transition for S2. Because the *timer* action is taken only if S2 is active, hence we set the guard of the added transition as $!(x > 5)$ which is the negation of S2's outgoing transition guard. The added transition is also synchronized with an auxiliary timer automaton defined by **Rule 5** through channel every 1s. In the mean time, the update function updateEvery1sS2(int &x) is called to increase x.



```
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;} }
```

Fig. 5. State Action Transformation

2) *Flatten Hierarchical Structure:*

Yakindu allows composite states, while UPPAAL only support flat structure. Such disparity requires the Y2U tool to flat the hierarchical structure in Yakindu model, i.e., separate the sub-automata contained in the composited state from the main automaton containing the composite state, and implement the interactions between a sub-automaton and the main automaton through synchronizations between them. If a composite state contains multiple sub-automata, we transform each sub-automaton using **Rule 7** below. If a statechart model contains nested composite states, we flat it recursively starting from the out most composite state.

Rule 7: Composite State

First, a composite state is represented by a simple state in UPPAAL model. The actions in a composite state are also transformed by **Rule 6**. To maintain the interaction between the main automaton and sub-automata in a composite state, we declare two channels: active for activating/entering the composite state and deactive for deactivating/exiting sub-automata in the composite state. In the main automaton, we add two output synchronizations on active and deactive channels for incoming and outgoing transitions of the composite state, respectively. The sub-automata in a composite state are hence separated from the main automaton. For each sub-automaton, as all its transitions have lower priorities than each outgoing transition of the composite state, we adjust the transition guards according to the transition priority rule, i.e., **Rule 8** below. In addition, we add an input synchronization on the active channel for the initial state's outgoing transition, and add a transition, which outgoes to the initial state and synchronizes on the input deactive channel, for each state except the initial one. We set the execution priorities (**Rule 9**) of sub-automata to be only lower than the main automaton. In addition, such sub-automata priorities setting schema also prevents the main automaton to accept events raised by sub-automata, which is not allowed in Yakindu execution semantics. Fig. 6 shows an example of composite state transformation. In Fig. 6, the sub-automaton is activated/deactivated when the main automaton enters/exists state S2. The activation/deactivation of the sub-automaton is implemented by synchronization with the main automaton through channel active/deactive. As the sub-autamaton transition SS1 $\rightarrow$ SS2 has lower priority than the main automaton transition S2 $\rightarrow$ S3, based on **Rule 8**, the guard of SS1 $\rightarrow$ SS2 is changed to $y == 0\&\&!(x == 0)$.

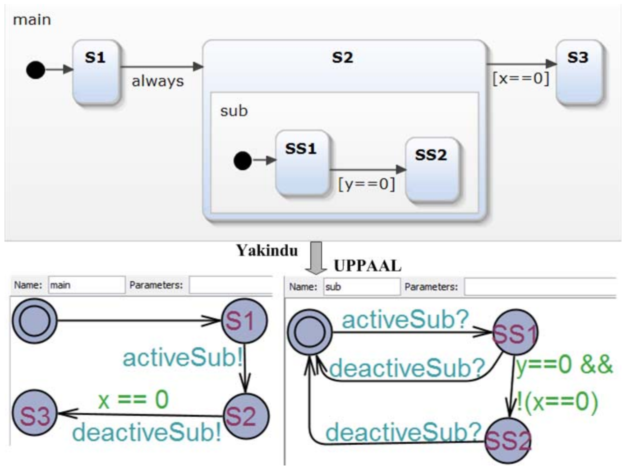


Fig. 6. Composite State Transformation

3) *Modeling Determinism and Synchrony with UPPAAL:*

The Yakindu statecharts have deterministic and synchronous execution semantics [2], while UPPAAL timed automata's execution semantics is non-deterministic and asynchronous [4]. To maintain the semantics equivalence between the Yakindu statechart and the transformed UPPAAL timed automata, we need to model determinism and synchrony in UPPAAL.

To implement the determinism within an automaton, the Yakindu statecharts assign a priority to each transition and select the transition with the highest priority from all enabled outgoing transitions of the same state to perform. The default priority is set as the order transitions are added when the model is built. We implement the transition priorities in UPPAAL by **Rule 8** given below.

Rule 8: Transition Priority

Assume a state has n outgoing transitions $\{T_1, T_2, \ldots, T_n\}$ sorted in non-increasing priority order, and the guard of transition T_i is denoted as G_i, to consider transition priorities, we change the transition guard for T_i to be G_i && $!G_1$ && $!G_2$ && $\ldots$ && $!G_{i-1}$ to enforce that higher priority transitions take place before lower priority transitions.

The Yakindu statecharts implement the determinism among different automata by automaton priority and synchronous execution. According to the decreasing automation priority, each automaton executes one step in one time cycle if the transition is enabled, otherwise the automaton stays in current state. In UPPAAL timed automata, we use the lockstep method [16], shown in **Rule 9**, to force synchronous execution based on automaton priorities. We ignore the added event automata (**Rule 2**) and timer automata (**Rule 3**) when modeling synchrony in UPPAAL as the added automata does not affect the model's execution behavior.

Rule 9: Automaton Priority and Synchrony

Suppose a model contains n automata $\{A_1, A_2, \ldots, A_n\}$ that are sorted by its execution priority in decreasing order. The lockstep method to model synchrony is as follows. Each automaton A_j is associated with an integer I_j to indicate how many steps A_j has executed. If none of outgoing transitions is enabled, the automaton stays in current state, which is equivalent to that the automaton executes a self loop step in current state. To ensure all automata execute the same number of steps, 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. If a state does not have outgoing transitions, then the added self-loop transition is guarded by `true`. 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. I_j is increased by 1 when A_j executes one step. More specifically, as A_1 has the highest priority, to ensure A_1 gets executed first, we add an additional guard $I_1 == I_2$ && $I_2 == I_3$ && $\ldots$ && $I_{n-1} == I_n$ on its each transition. For other automata A_j $(j > 1)$, to ensure its execution order, we add $I_j < I_{j-1}$ as a transition guard for A_j. To solve possible out-of-range problem of step indicators I_j, I_j is reset to 0 when it reaches its maximal value $(2^{15}-1)$. As Yakindu does not store events, hence when the automaton with the lowest priority executes a transition, it calls `empty()` defined in Section II-B4 to clear all events for current time cycle. We use an example to explain the transformation rule.

Assume we have three automata `T1`, `T2`, and `T3` as shown in Fig. 7, where the execution priority is `T1` $>$ `T2` $>$ `T3`. We define the step indicators for `T1`, `T2`, and `T3` as `synT1`, `synT2`, and `synT3`, respectively. For each state in the three automata, we add a self-loop transition that disables original outgoing transitions of the state. For instance, the guard of added self-loop transition for state `A1` is $!(x > 0)$. For each transition, additional guards are added to force lockstep execution, such as guard function `checkHighest()` for `T1` and `checkLower()` for `T2` and `T3`. Each transition also calls function `update()` to increase step indicator by 1 and reset the indicator of higher priority automaton to 0 if its maximal value is reached. However, the step indicator reset of the lowest priority automaton `T3` is not performed. To solve the problem, for each state in the highest priority automaton `T1`, we add another self-loop transition which checks if `synT3` reaches maximal value by function `checkLowestMAX()` and reset `synT3` by function `updateLowestMAX()`.

Note that in addition to maintaining the execution semantics equivalence, restricting UPPAAL model to have deterministic and synchronous execution semantics will simplify the process of tracing back to the Yakindu model when UPPAAL finds an unsatisfied property.

4) Support Simultaneous Events in UPPAAL:

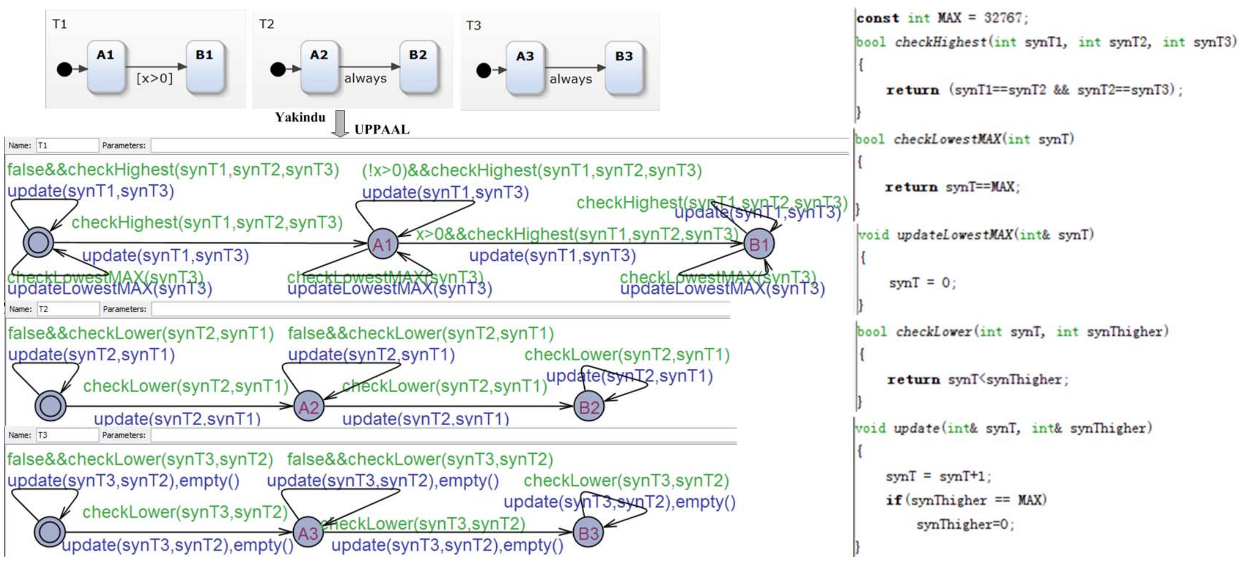
We use synchronizations provided in UPPAAL to simulate the *events* in Yakindu. The challenge is that Yakindu supports simultaneous events, while UPPAAL does not. A simple solution to simulate simultaneous events in UPPAAL seems to be adding series *committed* states and represent an event by a synchronization on a transition between two *committed* states. However, such solution can not be directly applied. First, we have to ensure the simultaneous events maintain a consistent order, or can lead to a deadlock. Second, the added synchronizations may interfere with other transformation rules that involve synchronizations (such as **Rule 5**, **Rule 6.2**, and **Rule 7**).

To solve the problem, we design an event stack to store all raised/valid events during one execution time cycle. The event stack includes the current valid event number and all valid events each of which is represented by a unique integer. The constant integer `TotalEventNumber` is adjusted based on the model. A dictionary that maps each event name to its corresponding integer is provided. The event stack provides three functions: (1) `empty()` empties the event stack at the end of an execution time cycle and is called by the automaton with lowest execution priority (**Rule 9**); (2) `isEventValid(int event)` checks if the input `event` is valid and is called when a transition guard contains event; (3) `push(int event)` pushes an input `event` onto the stack and is called when an event is raised. The implementation of the event stack is given in Fig. 8.

With the support of event stack, we obtain the following properties: (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; and (4) an event can only be accepted in the same execution time cycle in which it is raised.

C. Correctness of the Transformation

As Yakindu statecharts lack formal semantics, it is difficult to prove the equivalence between a Yakindu model and its associated transformed UPPAAL model unless a concrete equivalence criteria is given. In this paper, we define two models 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, and (2) all variable values at each



```
const int MAX = 32767;
bool checkHighest(int synT1, int synT2, int synT3)
{
    return (synT1==synT2 && synT2==synT3);
}
bool checkLowestMAX(int synT)
{
    return synT==MAX;
}
void updateLowestMAX(int& synT)
{
    synT = 0;
}
bool checkLower(int synT, int synThigher)
{
    return synT<synThigher;
}
void update(int& synT, int& synThigher)
{
    synT = synT+1;
    if(synThigher == MAX)
        synThigher=0;
}
```

Fig. 7. Synchrony Modeling in UPPAAL

```
const int TotalEventNumber = 10;
typedef struct {
    int validEvents[TotalEventNumber];
    int validEventNumber;
} EventStack;
EventStack eventStack;
void empty(){eventStack.validEventNumber = 0;}
bool isEventValid(int event) {
    int i = 0;
    for(i=0; i< eventStack.validEventNumber; i++)
    { if(eventStack.validEvents[i] == event){return true;} }
    return false;
}
void push(int event) {
    if(! isEventValid(event))
    { eventStack.validEvents[eventStack.validEventNumber] = event;
      eventStack.validEventNumber++; }
}
```

Fig. 8. Event Stack

execution step of the two models are equal. The following theorem proves that the transformation is correct.

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

Proof. We prove it by induction on the number N of iteration the rule is applied.
Base Case: When $N = 1$, the transformation maintains the execution equivalence. To prove the base case, we need to prove each rule maintains the execution equivalence. We take **Rule 6** as an example and simulate the models in Fig. 5 using both Yakindu and UPPAAL. For both the Yakindu model and the transformed UPPAAL model, the execution paths are $S1 \to S2 \to S3$ and the value change traces of variable x are $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 10$. The tracing results show that the two models have the same execution behaviors. Other transformation rules can be proven similarly.
Induction Step: Assume the statement is true when $N = k$ and prove it also holds when $N = k + 1$. According to the transformation rules, each rule does not interfere the execution behaviors of other rules. The base case has proven each transformation rule maintains the execution equivalence. Hence, if the statement holds when $N = k$, then it is also true when $N = k + 1$. ■

D. Transformation Procedure

The transformation procedure of the Y2U tool is shown in Algorithm 1. First, we define a data structure M to represent automata structure (Line 1). The data structure M contains variable declarations, automata information, states information, transitions information, etc. Then the Y2U tool parses the input Yakindu statechart Y according to Yakindu statecharts' XML format and stores all information in M (Line 2). For each automaton A stored in M, the Y2U tool transforms each element in A based on **Rule 1** to **Rule 9** defined in Section II-B and updates corresponding information in M (Line 3-5). Lastly, the Y2U tool creates the transformed UPPAAL timed automata U, which is corresponding to the input Y, from the updated M according to UPPAAL XML format (Line 6).

III. Trace Failed UPPAAL Properties back to Yakindu Statecharts

In the process of proving a desired property on an executable medical best practice guideline, UPPAAL may find a counter example, i.e., an execution path that fails the desired property. In this case, we need to trace the failed UPPAAL property back to the Yakindu model. However, as the transformation from Yakindu model to UPPAAL model may add new states and transitions in the UPPAAL model. The trace back of a failed execution path from UPPAAL model to Yakindu model

Algorithm 1 TRANSFORM(Y)

```
Input: A Yakindu statechart $Y$.
Output: The transformed UPPAAL timed automata $U$ corresponding to the input $Y$.
1: Define a data structure $M$ to represent automata
2: Parse $Y$ to $M$
3: for each automaton $A$ in $M$ do
4:     Transform each element in automaton $A$ based on Rule 1 to Rule 9 and update $M$
5: end for
6: Create $U$ from $M$
7: return $U$
```

becomes a challenge task. In this section, we first identify state and transition relationships between Yakindu and transformed UPPAAL models firstly, and then analyze the execution paths in UPPAAL and present the trace back procedure.

A. State and Transition Relationships between Yakindu and Transformed UPPAAL Models

For the nine transformation rules given in Section II-B, **Rule 4** and **Rule 5** add auxiliary event automata and timer automata to the UPPAAL model to simulate the event raise and timing trigger, respectively. As proven by Theorem 1, 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.

For a given Yakindu model Y, let $\mathcal{S}_Y$ and $\mathcal{T}_Y$ denote the state set and transition set in Y, respectively. Let $\mathcal{S}_U$ and $\mathcal{T}_U$ denote the state and transition sets transformed from the Yakindu model Y using transformation rules (**Rule 1** to **Rule 9**) with auxiliary event and timer automata being removed. We have the following theorems.

Theorem 2. *Given a Yakindu model Y and its transformed UPPAAL model U, the mapping from UPPAAL state set $\mathcal{S}_U$ to Yakindu state set $\mathcal{S}_Y$ is bijective.* □

Proof. According to **Rule 1**, each state in Yakindu model is transformed to a unique state in UPPAAL model, hence the mapping from $\mathcal{S}_U$ to $\mathcal{S}_Y$ is surjective. Based on all transformation rules (**Rule 1** to **Rule 9**) without considering auxiliary event and timer automata, no additional states are added into $\mathcal{S}_U$, so the mapping from $\mathcal{S}_U$ to $\mathcal{S}_Y$ is also injective. Therefore, the mapping from $\mathcal{S}_U$ to $\mathcal{S}_Y$ is bijective. ■

Theorem 3. *Given a Yakindu model Y and its transformed UPPAAL model U, the mapping from UPPAAL transition set $\mathcal{T}_U$ to Yakindu transition set $\mathcal{T}_Y$ is surjective, but not injective.* □

Proof. According to **Rule 2**, each transition in Yakindu model is transformed to a unique transition in UPPAAL model, hence the mapping from $\mathcal{T}_U$ to $\mathcal{T}_Y$ is surjective. However, **Rule 6**, **Rule 7**, and **Rule 9** add additional transitions into $\mathcal{T}_U$, so the mapping from $\mathcal{T}_U$ to $\mathcal{T}_Y$ is not injective. ■

B. Execution Paths in UPPAAL

If a desired property is not satisfied, UPPAAL often gives a counter example. Depending on different verification strategies (such as searching orders) that a user chooses. counter examples may be different. However, UPPAAL may not be able to provide all possible counter examples through limited number of verifications, because the verification stops once a counter example is detected.

Example 1. *Given a model containing three automata* `T1`*,* `T2`*, and* `T3`*, as shown in Fig. 9, the property we want to verify is that x is always larger than 0 when* `T2` *reaches state* `D2`*, i.e., $A[\ ]$* `T2.D2 imply` *$x > 0$. For the given model, the property does not hold, and UPPAAL gives following counter execution paths: (1)* $\texttt{Path1}: (\texttt{A1},\texttt{A2},\texttt{A3}) \rightarrow (\texttt{A1},\texttt{B2},\texttt{A3}) \xrightarrow{\texttt{true}} (\texttt{A1},\texttt{C2},\texttt{A3}) \rightarrow (\texttt{A1},\texttt{D2},\texttt{A3})$*, (2)* $\texttt{Path2}: (\texttt{A1},\texttt{A2},\texttt{A3}) \rightarrow (\texttt{A1},\texttt{A2},\texttt{B3}) \rightarrow (\texttt{B1},\texttt{A2},\texttt{B3}) \rightarrow (\texttt{B1},\texttt{B2},\texttt{B3}) \xrightarrow{\texttt{syn}} (\texttt{C1},\texttt{C2},\texttt{B3}) \rightarrow (\texttt{C1},\texttt{D2},\texttt{B3})$*. For each path, if there are multiple transitions between two states, we put the transition information (such as guards, synchronizations, updates) over the arrow to uniquely identify the transition. For instance, there are two transitions between state* `B2` *and* `C2`*, we use* `syn` *on the path to identify the transition with the synchronization between* `B2` *and* `C2`*.* □

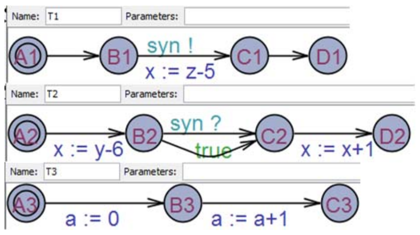


Fig. 9. Trace Back Example

In a UPPAAL execution path, each state is the combination of the active states of all automata. Each step in a path only takes one transition in an automaton, except the step that synchronizes two automata, in which case both synchronized automata takes one transition.

Note that we prefer breadth first search to depth first search when finding a counter example by UPPAAL. There are two reasons for the select preference: (1) the breadth first search can find more potential individual paths for different automata; and (2) the breadth first search can avoid individual paths of automata that do not interfere with the desired property, such as the path of automaton `T3` in Example 1.

C. Trace Back Procedure

Based on the analysis given in Section III-A, the auxiliary event automata added by **Rule 4** and timer automata added by **Rule 5** can be ignored from back tracing perspective. For the other automata, we need to trace states and transitions in the execution path from UPPAAL back to Yakindu. Based on Theorem 2, the state mapping from Yakindu model to UPPAAL model is bijective. The trace back of states is simple and we only need to find the corresponding states. According to Theorem 3, the transition mapping from UPPAAL model to Yakindu model is surjective but not injective. Hence, for the transitions in UPPAAL model which have corresponding transitions in Yakindu model, the trace back is also simple. However, additional analysis is needed for tracing back the added transitions in UPPAAL model.

Based on the proof of Theorem 3, the added transitions can be divided into three cases as follows.

Case 1: transitions added by **Rule 6**.
If a state contains timer actions, then self-loop transitions are added to the state. If the added transitions fail a desired property, it should be traced back to the corresponding timer actions in the state. However, the state is already contained in the path to be traced back, otherwise the added transitions can not be in the trace back path. Hence, the added transitions can be ignored in this case.

Case 2: transitions added by **Rule 7**.
For a composite state, transitions are added into sub-automata to simulate the exit execution of the sub-automata. Based on Theorem 1, these additional transitions does not change the model's execution behaviors. Hence, the added transitions can be ignored in this case as well.

Case 3: transitions added by **Rule 9**.
To model synchrony in UPPAAL, self-loop transitions are added to each state to simulate that the automaton stays in current state. Based on Theorem 1, these added transitions does not change the model's execution behaviors. Hence, the added transitions can also be ignored.
According to the above analysis, we can ignore the back trace of transitions added into UPPAAL model by **Rule 6**, **Rule 7**, and **Rule 9**.

Based on above analysis, the trace back procedure is as follows: (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 Yakindu model; (3) for each transition in the path, find its corresponding transition in Yakindu model; if the corresponding transition is not found in the Yakindu model, ignore the transition.

There may be multiple execution paths that cause a property to fail, but UPPAAL can only provide one counter example at a time. We take iterative approach to fix one counter example at a time, re-transform the modified Yakindu model for further verification with UPPAAL. We use Example 2 to show the iterative process.

Example 2. *Given a Yakindu model whose transformed UPPAAL model is the same as the one given in Example 1 (shown in Fig. 9), verify the property* $A[\,]$ `T2.D2 imply` $x > 0$.

We transform the Yakindu model using the Y2U tool and verify the property. The property is not satisfied, and a counter example given by UPPAAL is $\texttt{Path1}: (\texttt{A1},\texttt{A2},\texttt{A3}) \rightarrow (\texttt{A1},\texttt{B2},\texttt{A3}) \xrightarrow{\texttt{true}} (\texttt{A1},\texttt{C2},\texttt{A3}) \rightarrow (\texttt{A1},\texttt{D2},\texttt{A3})$. *The* `Path1` *is traced back to the Yakindu model, and the corresponding path in Yakindu is the same with* `Path1`. *We analyze the path in Yakindu model and fix the action of transition* $\texttt{A2} \rightarrow \texttt{B2}$ *to* $y := 7, x := y - 6$ *in Yakindu model.*

We re-transform the modified Yakindu model and verify the property again. The property still does not hold, and a counter example given by UPPAAL is $\texttt{Path2}: (\texttt{A1},\texttt{A2},\texttt{A3}) \rightarrow (\texttt{A1},\texttt{A2},\texttt{B3}) \rightarrow (\texttt{B1},\texttt{A2},\texttt{B3}) \rightarrow (\texttt{B1},\texttt{B2},\texttt{B3}) \xrightarrow{\texttt{syn}} (\texttt{C1},\texttt{C2},\texttt{B3}) \rightarrow (\texttt{C1},\texttt{D2},\texttt{B3})$. *Using back trace process, we fix the action of transition* $\texttt{B1} \rightarrow \texttt{C1}$ *to* $z := 6, x := z - 5$ *in Yakindu model.*

We re-transform the modified Yakindu model and verify the property again. The property now is satisfied. In this example, two iterations are taken to fix the errors in the model. □

IV. Cardiac Arrest Case Study

In this section, we perform a case study on a simplified medical scenario. The models and detailed information of the cardiac arrest case study are available on our website www.cs.iit.edu/∼code/software/Y2U/case-study/cardiac-arrest.html.

A. Cardiac Arrest Scenario

Cardiac arrest is the abrupt loss of heart function and can lead to death within minutes. American Heart Association (AHA) provided resuscitation guidelines for the urgent treatment of cardiac arrest [6].

In a cardiac arrest scenario, 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[1]. Suppose the patient's airway is open and breathing is under control. However, 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. Giving 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 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.

In medical treatment procedure, the side effects of a treatment may invalidate the previously satisfied preconditions. For example, one potential side effect of sodium bicarbonate is suppressed respiratory drive[5], which adversely affect patient breathing that is one precondition of activating a defibrillator. In this case, the preconditions of activating a defibrillator need to be re-checked. If the breathing precondition is invalidated, assisted ventilation should be provided In addition, a treatment may not be effective and additional treatments should be provided. For example, increasing IV fluid volume may not

[1]The shockable rhythms are ventricular fibrillation and ventricular tachycardia [6].

[2]Non-shockable rhythms are asystole and pulseless electrical activity [6].

[3]Severe acidosis, which is an increased acidity in the blood and other body tissue, will significantly reduce the effectiveness of epinephrine [6].

[4]If a patient suffers from kidney insufficiency, giving epinephrine may worsen the kidney function and cause acute renal failure [6].

[5]Respiratory drive is the control of respiration, which involves the exchange of oxygen and carbon dioxide [6].

successfully improve patient's urine flow rate. In this case, diuretics, such as Lasix, should be given. Fig. 10 shows a simplified cardiac arrest treatment procedure.

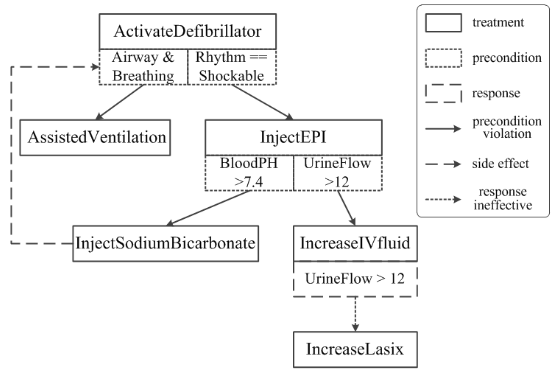


Fig. 10. Cardiac Arrest Treatment Procudure

There are two medical properties needed to be verified in the cardiac arrest treatment validation procedure: **P1**: Defibrillator is activated only if the EKG rhythm is shockable and airway and breathing is normal; and **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. Executable and Verifiable Model Design

In our previous work, we 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 [17], [15]. In this paper, we use Yakindu to model the simplified cardiac arrest treatment procedure with the validation protocol. The model consists of the following automata: Treatment, Ventilator, EPIpump, SodiumBicarbonatePump, IVpump, and LasixPump. The automata communicate using Yakindu events and shared variables. The *Treatment* automaton implements preconditions validation, side effects monitoring, and expected responses checking specified in Fig. 10. The other automata implement treatment actions (such as medicine injection) and communicate with the *Treatment* automaton. Due to the space limit, we focus on the *Treatment* automaton. The *Treatment* automaton is built with Yakindu statechart tool, as shown in Fig. 11.

We transform the model built with Yakindu to UPPAAL model with the Y2U tool. The transformed *Treatment* automaton in UPPAAL is given in Fig. 12. For the transformed model, we need to verify the medical properties **P1** and **P2** given by the medical staff always hold. In addition, we need to verify the UPPAAL model is deadlock free.

The property **P1** can be checked by two formulas: (1) $E <> \ Treatment.ActivateDefibrillaotr$ and (2) $A[\,]\ Treatment.ActivateDefibrillaotr\ imply\ Breath == 0\ \&\&\ Rhythm == 0$. The first formula verifies that if the $ActivateDefibrillaotr$ state is eventually reached, and the second one verifies that if the two preconditions are always satisfied when the defibrillaotr is activated. The property **P2** is represented as $E <> \ Treatment.InjectEPI$ and $A[\,]\ Treatment.InjectEPI\ imply\ BloodPH\_int >= 7\ \&\&\ BloodPH\_frac > 4\ \&\&\ UrineFlow\_int > 12$. It can be verified similarly. formulas. The deadlock free property is checked by the formula $A[\,]\ not\ deadlock$. In the transformed cardiac arrest treatment UPPAAL model, all the properties, i.e., **P1**, **P2**, and deadlock free, are satisfied.

C. Trace Back with Injected Error

To show the trace back procedure, we inject an error into the Yakindu model as follows: change the guard from state `InjectEPIPre` to state `InjectEPI` to `BloodPH` $>$ 7.4 && `UrineFlow` > 9.

We transform the Yakindu model with injected error to UPPAAL model and verify property **P2**. The verification result is failure and a counter example given by UPPAAL is `CP` : `ActivateDefibrillaotrPre` $\rightarrow$ `InjectEPIPre` $\rightarrow$ `InjectEPI`. Then we trace `CP` back to the Yakindu model, which is the same with `CP`. We correct the error by modifying the guard from state `InjectEPIPre` to state `InjectEPI` to `BloodPH` > 7.4 && `UrineFlow` > 12. The modified Yakindu model is re-transformed, re-verified property **P2**, which is satisfied.

V. Conclusion

The existing medical best practice guidelines in hospital handbooks are often lengthy and difficult for medical staff to remember and apply clinically. The paper present an approach to transform medical best practice guidelines to executable and verifiable statechart models. In particular, we present a tool, called Y2U, to transform a statechart model to a verifiable formal model, so that formal verification can be exercised. The transformation rules are designed in such a way that not only the transformed formal model maintains the execution equivalence with the original statechart model, they also allow easy trace back to the statechart model in case verification finds a counter example. The case study of a simplified cardiac arrest treatment scenario clearly demonstrated: (1) statechart model provides a more efficient and accurate way to communicate among medical experts and computer scientists; (2) the Y2U tool accurately transform the Yakindu model to the UPPAAL model; and (3) the back trace functionality supported by the Y2U tool provides good feed back to medical personals for quick identifying errors in the statechart model.

The tool is available on our website www.cs.iit.edu/∼code/software/Y2U.

Acknowledgement

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.

References

[1] Autonomous driving with the yakindu smart car. http://blog.statecharts.org/2015/02/yakindu-smart-car.html.
[2] Yakindu statechart tools. https://www.itemis.com/en/yakindu/statechart-tools/.
[3] Yakindu statecharts enter lego mindstorms. http://blog.statecharts.org/2014/11/yakindu-statecharts-enter-lego.html.

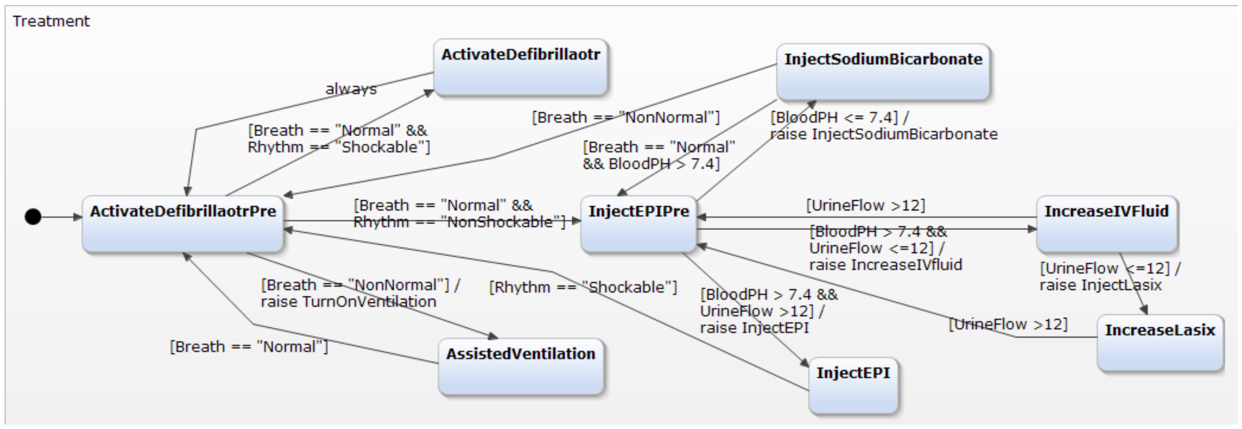

Treatment
ActivateDefibrillaotr
InjectSodiumBicarbonate
always
[Breath == "NonNormal"]
[BloodPH <= 7.4] / raise InjectSodiumBicarbonate
[Breath == "Normal" && Rhythm == "Shockable"]
[Breath == "Normal" && BloodPH > 7.4]
ActivateDefibrillaotrPre
[Breath == "Normal" && Rhythm == "NonShockable"]
InjectEPIPre
[UrineFlow >12]
IncreaseIVFluid
[BloodPH > 7.4 && UrineFlow <=12] / raise IncreaseIVfluid
[UrineFlow <=12] / raise InjectLasix
[Breath == "NonNormal"] / raise TurnOnVentilation
[Rhythm == "Shockable"]
[BloodPH > 7.4 && UrineFlow >12] / raise InjectEPI
[UrineFlow >12]
IncreaseLasix
[Breath == "Normal"]
AssistedVentilation
InjectEPI

Fig. 11. Cardiac Arrest Treatment Yakindu Model

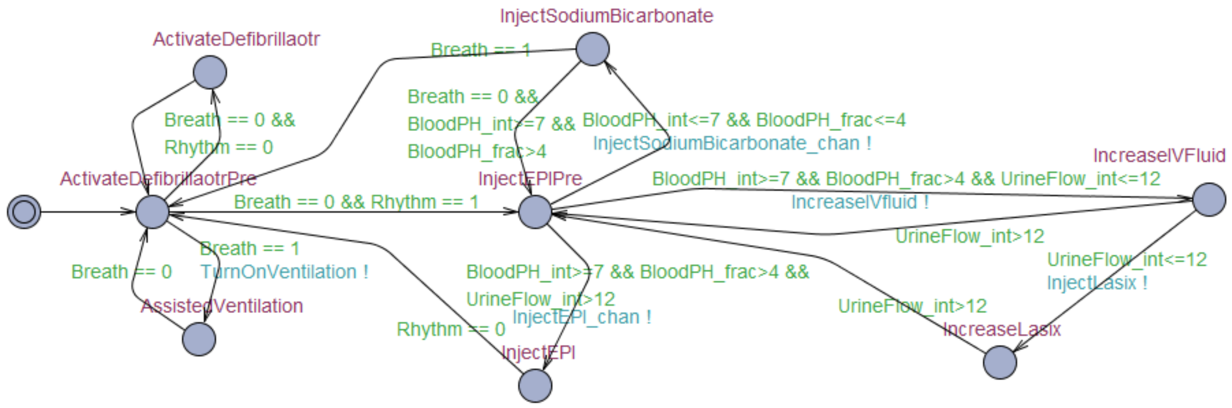


Fig. 12. Cardiac Arrest Treatment UPPAAL Model

[4] G. Behrmann, A. David, and K. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236. Springer, 2004.

[5] A. David, M. Möller, and W. Yi. Formal verification of uml statecharts with real-time extensions. In *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. 2002.

[6] J. M. Field, M. F. Hazinski, M. R. Sayre, L. Chameides, S. M. Schexnayder, R. Hemphill, R. A. Samson, J. Kattwinkel, R. A. Berg, F. Bhanji, et al. Part 1: executive summary 2010 american heart association guidelines for cardiopulmonary resuscitation and emergency cardiovascular care. *Circulation*, 122(18 suppl 3):S640–S656, 2010.

[7] J. Fox, N. Johns, and A. Rahmanzadeh. Disseminating medical knowledge: the proforma approach. *Artificial Intelligence in Medicine*, 14(1-2):157 – 182, 1998.

[8] A. Furfaro and L. Nigro. A development methodology for embedded systems based on rt-devs. *Innovations in Systems and Software Engineering*, 5(2):117–127, 2009.

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

[10] G. Hripcsak, P. D. Clayton, T. A. Pryor, P. Haug, O. Wigertz, and J. Van der Lei. The arden syntax for medical logic modules. In *Proceedings/the... Annual Symposium on Computer Application [sic] in Medical Care. Symposium on Computer Applications in Medical Care*, pages 200–204. American Medical Informatics Association, 1990.

[11] M. Jersak, K. Richter, R. Ernst, et al. Formal methods for integration of automotive software. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 45–50, 2003.

[12] O. Laurent. Using formal methods and testability concepts in the avionics systems validation and verification (v&v) process. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 1–10, April 2010.

[13] M. Nobakht and D. Truscan. An approach for validation, verification, and model-based testing of uml-based real-time systems. In *the Eighth International Conference on Software Engineering Advances (ICSEA 2013)*, pages 79–85, 2013.

[14] V. L. Patel, V. G. Allen, J. F. Arocha, and E. H. Shortliffe. Representing clinical guidelines in glif. *Journal of the American Medical Informatics Association*, 5(5):467–483, 1998.

[15] B. R. Schatz and R. B. Berlin Jr. *Healthcare infrastructure: Health systems for individuals and populations*. Springer Science & Business Media, 2011.

[16] J. Smed and H. Hakonen. *Algorithms and Networking for Computer Games*. Wiley, 2006.

[17] P. Wu, D. Raguraman, L. Sha, R. Berlin, and J. 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.

[18] J. Xing, B. Theelen, R. Langerak, et al. From poosl to uppaal: Transformation and quantitative analysis. In *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, pages 47–56, June 2010.

[19] O. Young and Y. Shahar. The spock system: developing a runtime application engine for hybrid-asbru guidelines. In *Artificial Intelligence in Medicine*, pages 166–170. Springer, 2005.

[20] D. Zorin and V. Podymov. Translation of uml statecharts to uppaal automata for verification of real-time systems. *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering*, 6, 2012.