Formalization of UML Statechart Models Using Concurrent Regular Expressions

S. Jansamak **A. Surarerks**

ELITE (Engineering Laboratory in Theoretical Enumerable System)
Faculty of Engineering, Chulalongkorn University,
Pathumwan, Bangkok 10330, Thailand,
Email: sirichai.ja@student.chula.ac.th, athasit@cp.eng.chula.ac

Abstract

The Unified Modeling Language (UML) is widely used in the software development process for specification of system based on the object-oriented paradigm. Thought the current version of UML is semi-formal, it is difficult to determine whether the model is consistent, unambiguous, or complete. This paper proposes the transformation rules for formalizing UML statechart diagrams. The target language for the transformation is Concurrent Regular Expressions (CREs) which are extensions of regular expression. The transformation result will be represented in mathematical form and suitable for applying verification. We also show that this formalization can be used to check the simple inconsistency of the system.

Keywords: UML, Statechart, Formal, Concurrent Regular Expressions, Model Checking.

1 Introduction

The Unified Modeling Language (UML) is widely used in the software development process for the specification of system based on the object-oriented paradigm. Though the current version of UML proposed by OMG (Object Management Group 2003) is semi-formal, it is difficult to determine whether the model is consistent, unambiguous, and complete.

To overcome this problem, we introduce a method for formalizing UML models. There have been many attempts to formalize UML models, especially statecharts. However, formalizing of concurrent properties has not been described exactly. This paper provides an alternative approach of formalizing UML Statecharts. The behavior of system modeled by statechart is transformed to Concurrent Regular Expressions (CREs), which proposed by Vijay K. Garg *et al.* (Vijay K. Garg & M.T. Ragunath 1992).

A regular expression is a technique which describes a set of strings. The described string consists of adjacent alphabets such that the order is significant. In our approach each alphabet is replaced by event, conditions, or actions in statechart. It can be viewed that the described string represents the occurrence of events, conditions, and actions in sequential. CREs are extensions of regular expression. Four operators are added for modeling a concurrent system. Vijay K. Garg *et al.* (Vijay K. Gard et al. 1992) proposed this to describe Petri Nets in the algebra based form. Mitsutaka *et al.* (Mitsutaka Okazaki, Toshiaki Aoki & Takuya Katayama 2002) used CREs to extract threads from concurrent objects for the design of embedded systems.

The transformation between statecharts and CREs uses our proposed rules which cover the basic modeling form of statechart. Since the result is a formal language, it can be checked for consistent of the model in the mathematical way.

The structure of the paper is as follows: Section 2 presents background material: UML Statecharts and CREs. Section 3 introduces transformation rules between two languages. Section 4 applies the concept of transformation rules to the computer example. Section 5 introduces the simple inconsistency checking on system modeled by CREs. Section 6 ends with conclusion and future work.

2 Background

In this section we introduce UML Statecharts and CREs which are used in this work.

2.1 Statecharts

Statechart is a graph of states connected by transitions, which represent state machine. Typically, statecharts describe behavior of instance of class to event which it receives, but statechart may also describe the behaviors of other entities such as use-cases, actors, subsystems, operations and methods.

Statechart SM is a finite set of state transition models, where state transition model is defined as follows:

$$ST = (S, T, E, C, A, i, f)$$

where

- S is a finite set of states
- E is a finite set of events
- C is a finite set of conditions
- A is a finite set of actions
- $T \subseteq S \times E \times C \times 2^A \times S$ is a finite set of transitions where 2^A denotes the power set of A
- $i \in S$ is a initial state
- $f \in S$ is a final state.

2.2 Concurrent Regular Expressions

Concurrent Regular Expressions (CREs) are extension of regular expressions which we use to model concurrent system. Regular expressions consist of alphabet Σ, null string ϵ, and operators. The three operators using in regular expressions is as follows.

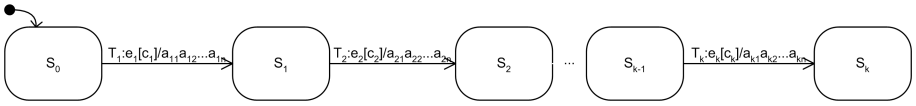

Figure 1: Sequential transitions

- (concatenation) $A.B$ denotes the set $\{a.b|a \in A \text{ and } b \in B\}$. For example, $\{a, b\}.\{c, d\} = \{a.c, a.d, b.c, b.d\}$
- (set union) $A + B$ denotes the union set of A and B. For example, $\{a\} + \{b\} = \{a, b\}$
- (Kleene-Star) A^* denoting the set of all strings that can be made by concatenating zero or more strings in A. It is defined as $A^* = \cup_{i=0,1,...} A^i$. For example, $\{ab\}^* = \{\epsilon, ab, abab, ababab, \ldots\}$.

Note that the concatenation operator denoted by dot (.) can be discarded from the expression, *i.e.* $a.b = ab$.

In this section the four extension operators: interleaving, alpha-closure, synchronous composition, and renaming will be introduced.

2.2.1 Interleaving

Interleaving operator is used to define the concurrent between multiple systems, which execute independently. Interleaving, denoted by $||$, is defined as follows.

$$a||\epsilon = \epsilon||a = a \qquad \forall a \in \Sigma$$
$$a.s||b.t = a.(s||b.t) \cup b.(a.s||t) \qquad \forall a, b \in \Sigma, s, t \in \Sigma^*$$

This definition can be extended to interleaving between two sets in a natural way, *i.e.*

$$A||B = \{w|\exists s \in A \text{ and } t \in B, w \in s||t\}.$$

For example, consider two sets A and B where $A = \{ab\}$ and $B = \{cd\}$ the $A||B = \{abcd, acbd, acdb, cabd, cadb, cdab\}$. Similar to $A||B$, we also get a set $A||A = \{aabb, abab\}$. The $A||A$ is denoted by $A^{(2)}$ to distinguish from the traditional use of the exponent *i.e.* $A^2 = A.A$.

If we want to model the system of two automatic vending machines. The first sells chocolates and the second sells drinks. The behavior of the first machine can be modeled by using CREs as $(Coin.Chocolate)^*$. The second machine can also be modeled by using CREs as $(Coin.Drink)^*$. As the two vending machines are independent working, we model the system as $(Coin.Chocolate)^*||(Coin.Drink)^*$.

2.2.2 Alpha-Closure

Alpha-Closure operator is defined as an analogue of a Kleene-Closure for the interleaving operator. Alpha-closure ,denoted by $^\alpha$, is defined as follows:

$$A^\alpha = \cup_{i=0,1,...} A^{(i)}$$

where $A^{(i)} = A||A \ldots i$ times.

For example, consider set $A = \{ab\}$ then $A^\alpha = \{\epsilon, ab, abab, aabb, ababab, aabbab, abaabb, \ldots\}$.

We can use the alpha-closure operator to describe a system which contains buffer. If we consider the producer and consumer problem. The consumer can consume products after the producer produce products. We model the system by using alpha-closure operator as $(Produce.Consume)^\alpha$. The expression describes the behavior of system as $\{\epsilon, Produce.Consume, Produce.Consume.Produce.Consume, Produce.Produce.Consume.Consume, \ldots\}$. As the result of alpha-closure operator, the number of $Produce$ in any prefix of set members is not less than the number of $Consume$.

2.2.3 Synchronous Composition

Synchronous composition operation is used to define the synchronization between multiple systems. For example, consider two sets A and B where $A = ab$ and $B = \{ac\}$ then $A[]B = \{abc, acb\}$. Synchronous composition denoted by [] is defined as follows:

$$A[]B = \{w|w/\Sigma_A \in A, w/\Sigma_B \in B\}$$

where w/S means the restriction of the string w to the symbols in S. For example, $\{aadcb\}/\{a, c\} = \{aac\}$ and $\{aadcb\}/\{b, d\} = \{db\}$.

The example of system which using this operator is a man and a vending machine. The behavior of customer is $(Thirsty.Coin)^*$. The behavior of vending machine is $(Coin.Drink)^*$. The man has a communication with the vending machine on the Coin event. Then the system will be modeled using synchronous composition operator as $(Thirsty.Coin)^*[](Coin.Drink)^*$.

In this paper we also use an operator $[S]$ which is an extension proposed by Mitsutaka Okazaki, Toshiaki Aoki, Takuya Katayama (2002). The set S contains actions that never be interleaved. The synchronous composition with S extension is defined as follows:

$$A[S]B = \{w|w \in (\Sigma_A + \Sigma_B)^*, w/(\Sigma_A \cup S) \in A, w/(\Sigma_B \cup S) \in B\}.$$

Then the distributive property always holds. For example, consider $(\{a\} + \{b\})[]\{b\} = \{b\}$, but $(\{a\}[]\{b\}) + (\{b\}[]\{b\}) = \{ab, ba, b\}$. If we use the operator $[S]$, $(\{a\} + \{b\})[]\{b\} = (\{a\}[S]\{b\}) + (\{b\}[S]\{b\}) = \{b\}$ where $S = \{b\}$.

2.2.4 Renaming

Renaming operator is used to rename the event symbols of a process. Let L_1 be a language defined over Σ_1. Let σ represent a function from Σ_1 to $\Sigma_2 \cup \{\epsilon\}$. Then $\sigma(L_1)$ is a Language defined over $\sigma(\Sigma_1)$ defined as follows:

$$\sigma(L_1) = \{(s)|s \in L_1\}$$

3 Formalization

In statecharts, transition arrows are labeled with events, conditions and actions. The occurrence of an event fires a transition if the machine is in the source state of the transition, the occurrence of the event matches the event of the transition, and the condition of the transition holds. In our approach, we simply use CREs to model event, condition, and actions as $e.g(c).a_1.a_2 \ldots a_n$ which means that actions a_1, a_2, ..., a_n will be performed if event e occurs and condition c holds.

The definition of transition function T is as follows:

$$T(p, e, c) = (q, a_1, a_2, \ldots, a_n)$$

where p, e, c, q, and a_1, a_2, ..., a_n is source state, event, condition, target state, and sequence of actions respective1y.

The CREs represent this transition as follows.

$$e.g(c).a_1.a_2a_n$$

We next show our rules for the transformation from UML Statecharts to CREs.

3.1 Sequential Transitions

A sequence of transition functions in statechart represents the order of state changing as in Figure 1. We can model the sequence of transition functions using concatenation operator (.) as follows:

$$T_1.T_2 \ldots T_k$$

where

- T_1 is $e_1.g(c_1).a_{11}.a_{12} \ldots a_{1n}$
- T_2 is $e_2.g(c_2).a_{21}.a_{22} \ldots a_{2n}$
- T_k is $e_k.g(c_k).a_{k1}.a_{k2} \ldots a_{kn}$

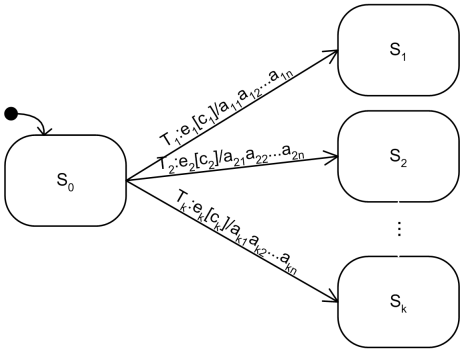

Figure 2: Choice transitions

3.2 Choice Transitions

Source state has multiple transitions function to target states as in Figure 2. The source state can change to each target state up to event trigger which received. We can model the transition functions using choice operator as follows:

$$T_1 + T_2 + \ldots + T_k$$

where

- T_1 is $e_1.g(c_1).a_{11}.a_{12} \ldots a_{1n}$
- T_2 is $e_2.g(c_2).a_{21}.a_{22} \ldots a_{2n}$
- T_k is $e_k.g(c_k).a_{k1}.a_{k2} \ldots a_{kn}$

3.3 Concurrent Substates

Composite state has concurrent substates, and each substate executes independently as in Figure 3. We can model the concurrent substates using synchronous operator as follows:

$$T_1[]T_2[] \ldots []T_k$$

where

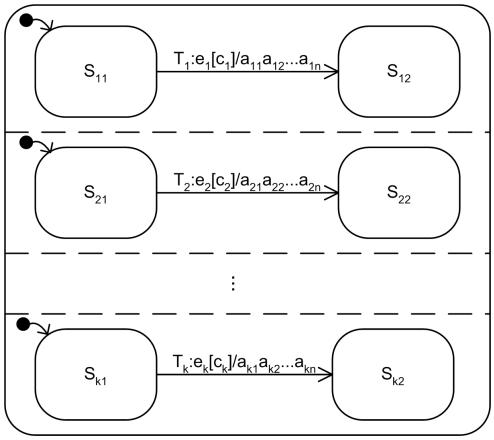

Figure 3: Concurrent substates

- T_1 is $e_1.g(c_1).a_{11}.a_{12} \ldots a_{1n}$
- T_2 is $e_2.g(c_2).a_{21}.a_{22} \ldots a_{2n}$
- T_k is $e_k.g(c_k).a_{k1}.a_{k2} \ldots a_{kn}$

Note that if there is no communication between concurrent substates ($\Sigma_{T_1} \cap \Sigma_{T_2} \ldots \Sigma_{T_k} = \emptyset$), we can replace the synchronous operator with interleaving operator such as $T_1||T_2|| \ldots ||T_k$.

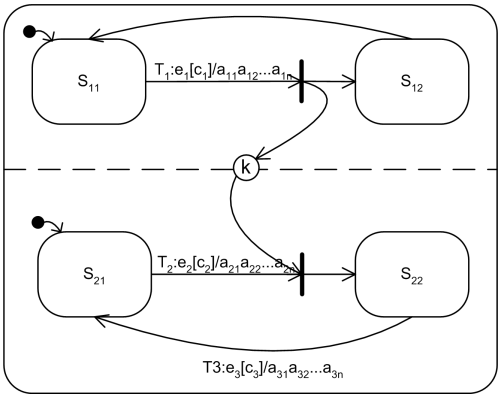

Figure 4: Synch state

3.4 Synch State

Composite state has two concurrent substates, and there is synch state between two concurrent states as in Figure 4. The synch state value can be any positive integer and star(*), which represents the number of tokens that can store in synch state. We can model this system as follows:

$$(T_1)^*[](T_2.T_3)^*[]Constraint^*$$

where

- T_1 is $e_1.g(c_1).a_{11}.a_{12} \ldots a_{1n}$
- T_2 is $e_2.g(c_2).a_{21}.a_{22} \ldots a_{2n}$
- T_3 is $e_3.g(c_3).a_{31}.a_{32} \ldots a_{3n}$
- $Constraint$ is $((T_1T_1^*T_2T_3)^*)^k||(T_2T_1T_1^*T_3)^*$ in the case that k is any positive integers, and in the case that k is $*$ constraint is $((T_1T_1^*T_2T_3)^*)^\alpha||(T_2T_1T_1^*T_3)^*$

The system is defined by using synchronous composition between 3 sub-CREs. The first and second are behavior of each substate. The third term is added constraint for synch state, which means T_1 and T_2 must occur before T_3 and synch state can store tokens when its value any positive integer or star(*).

4 Implementation

In this section we show the implementation of our rules to transform the Computer UML Statecharts to CREs.

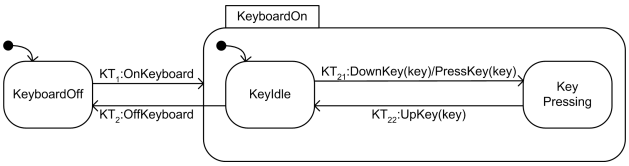

Figure 5: Keyboard

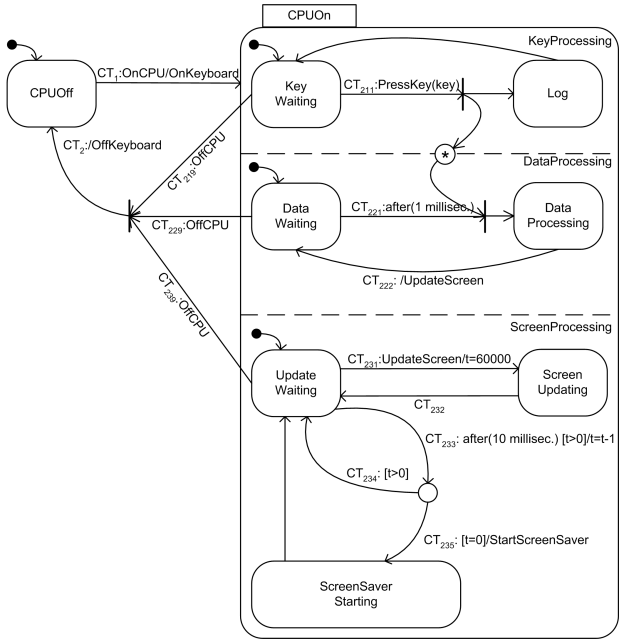

Figure 6: CPU

Computer in our example has two components: CPU and Keyboard. When the CPU is switched on, the Keyboard will be automatic switched on, and both waiting for input event. The Keyboard wait for user pressing a key, and the CPU wait for key input from the Keyboard. The CPU has three concurrent processing units. The first is Keyboard Processing, which waits for keyboard input. The second is Data Processing which checks for processing data every 1 millisecond. Data Processing unit will process data and update screen when input data is in queue. The last is Screen Processing which responses to update display screen. It updates screen when receives update event from Data Processing unit. If the screen is not updated in 10 minutes, the display screen will be changed to screen saver.

According to our computer specification, we can model the computer system which contains two objects: $Keyboard$ and CPU. $Keyboard$ is an object shown in Figure 5, which waiting for user input. It sends $PressKey$ event with parameter key to CPU when user presses key on keyboard. CPU is an object shown in Figure 6, which waiting for processing data. CPU detects for $PressKey$ sequence from $Keyboard$ every 1 millisecond, and processes the input keys one by one. Then the screen will be updated. If screen is not updated in 10 minutes, the screensaver will be activated.

The transitions of $OffCPU$ and $OffKeyboard$ are not connected to the boundary of $CPUOn$ and $KeyboardOff$ respectively because we want to reduce the complexity of model defined in CREs.

To define the computer system, we firstly define behavior of each object: $Keyboard$ and CPU. $Keyboard$ has two states: $KeyboardOff$ and $KeyboardOn$ state. The $KeyboardOn$ state is a composite state, which has $KeyIdle$ and $KeyPressing$ substate. We model $KeyboardOn$ state as $(KT_{21}.KT_{22})^*$ where KT_{21} is $DownKey(key).Press - Key(key)$ and KT_{22} is $UpKey(key)$. Then the behavior of $Keyboard$ is defined as follows:

$$Keyboard = (KT_1.KeyboardOn.KT_2)^*$$

where

- KT_1 is $OnKeyboard$
- KT_2 is $OffKeyboard$

CPU also has $CPUOff$ and $CPUOn$ state. The $CPUOn$ state is a composite state, which has three concurrent substates inside. The first substate $KeyProcessing$ is defined as CT_{211}^*, the second substate $DataProcessing$ is defined as $(CT_{221}.CT_{222})^*$, the third substate $ScreenProcessing$ is defined as $(CT_{231} + CT_{233}.(CT_{234} + CT_{235}))^*$, and $Constraint$ for synch state between $KeyProcessing$ and $DataProcessing$ substate is defined as $((CT_{211}CT_{211}^*CT_{221}CT_{222})^*)^\alpha$ $||$ $(CT_{221}CT_{211}CT_{211}^*CT_{222})^*$ where CT_{211}, CT_{221}, CT_{222}, CT_{231}, CT_{233}, CT_{234}, and CT_{235} is $PressKey(key)$, $after(1millisec.)$, $UpdateScreen$, $UpdateScreen.t = 60000$, $after(10millisec.).g(t > 0).t = t - 1$, $g(t > 0)$, and $g(t = 0).ChangeToScreensaver$. The $CPUOn$ is defined as $KeyProcessing$ [] $DataProcessing$ [] $ScreeenProcessing$ [] $Constraint$. Then the behavior of CPU is defined as follows:

$$CPU = (CT_1.CPUOn.CT_2)^*$$

where

- CT_1 is $OnCPU.OnKeyboard$
- CT_2 is $OffCPU.OffKeyboard$

The CPU and $Keyboard$ execute concurrently. Then the behavior of $Computer$ is defined as follows.

$$Computer = CPU[]Keyboard$$

5 Simple Inconsistency Checking

As the result of the transformation from UML Statecharts to CREs, we found that we can simply check the inconsistency of model. The checking concept base on our transformation rule 4, which we model the objects which execute concurrently by using synchronous composition operator. Each concurrent object has own possible behavior. The behavior of each object must hold when it executes concurrently with others objects.

Definition. *The system is consistent if all possible behaviors of all objects are included in the set of all behaviors of the system, otherwise it is inconsistent.*

Theorem. *If there is at least one object such that*

$$L \neq S/\Sigma_L$$

where

- *L is a CREs of object*
- *S is the set of synchronous composition between all objects which execute concurrently in the system*

- Σ_L *is an alphabet of* L.

Then the system is inconsistent.

Proof. Given a system with n objects, O_1, O_2, ..., O_n. Let L_i be a CREs of O_i, $1 \leq i \leq n$. The system $S = L_1[]L_2[]\ldots[]L_n$ is the set of all possible behaviors of the system. By definition of S, we have that $S/\Sigma_i \subseteq L_i$ for any i, Σ_i is an alphabet of L_i. Assume that there exists an O_j, $\exists win L_j$ and $w \notin S/\Sigma_j$. Since w is in L_j, w is a possible behavior of O_j. Because of $w \notin S/\Sigma_j$, w does not exist in the system. It follows that the system is inconsistent. □

We next show the example for applying our simple consistency checking rule to the vending machine problem as shown in Figure 7 and Figure 8.

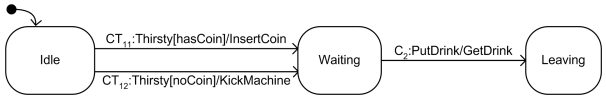

Figure 7: Thirsty man

Figure 8: Vending machine

The example is a system of a thirsty man and vending machine. The first statechart is modeled for a thirsty man. There are two ways to get a drink from vending machine. First is inserting a coin into the machine, and getting the drink. The other is kicking the machine and getting the drink because he does not have a coin. We can define the behavior of the thirsty man Man as follows:

$$Man = (CT_{11} + CT_{12}).CT_2$$

where

- CT_{11} is $Thirsty.g(hasCoin).InsertCoin$
- CT_{12} is $Thirsty.g(noCoin).KickMachine$
- CT_2 is $PutDrink.GetDrink$.

Then, the possible behavior of Man is as follows:

$$\begin{aligned} Man = \{&Thirsty.g(hasCoin).InsertCoin.\\ &PutDrink.GetDrink, Thirsty.\\ &g(noCoin).KickMachine.\\ &PutDrink.GetDrink\} \end{aligned}$$

The second statechart in Figure 8 is model of a vending machine. The vending machine waits for coin from a man. It puts the drink when it only gets a coin. We can define the behavior of the vending machine VM as follows.

$$VM = InsertCoin.PutDrink$$

Then, the possible behavior of VM is as follows.

$$VM = \{InsertCoin.PutDrink\}$$

As the system is a concurrent execution with communication between Man and VM, so we will use the synchronous composition operator to connect between them as follows:

$$System = Man[]VM.$$

To find the possible behavior of the system, we apply the distributive law to separate the expression into two as follow:

$$\begin{aligned} System &= Man[]VM\\ &= (CT_{11} + CT_{12}).CT_2[]VM\\ &= (CT_{11}.CT_2 + CT_{12}.CT_2)[]VM\\ &= (CT_{11}.CT_2[S]VM) + (CT_{12}.CT_2[S]VM), \end{aligned}$$

where $S = \Sigma_{Man} \cap \Sigma_{VM} = \{InsertCoin, PutDrink\}$. The possible behavior of $CT_{11}.CT_2[S]VM$,

$$\begin{aligned} CT_{11}.CT_2[S]VM = &Thirsty.g(hasCoin).InsertCoin.\\ &PutDrink.GetDrink[S]\\ &InsertCoin.PutDrink\\ = &\{Thirsty.g(hasCoin).InsertCoin.\\ &PutDrink.GetDrink\}. \end{aligned}$$

The possible behavior of $CT_{12}.CT_2[S]VM$,

$$\begin{aligned} CT_{12}.CT_2[S]VM = &Thirsty.g(noCoin).Kick\\ &Machine.PutDrink.GetDrink\\ &[S]InsertCoin.PutDrink\\ = &\{\}. \end{aligned}$$

Then the behavior of System is

$$\begin{aligned} System = &\{Thirsty.g(hasCoin).InsertCoin.\\ &PutDrink.GetDrink\} + \{\}\\ = &\{Thirsty.g(hasCoin).InsertCoin.\\ &PutDrink.GetDrink\}. \end{aligned}$$

Finally, we check inconsistency of the system by applying the restriction of the $System$ to the actions in Man and VM,

$$\begin{aligned} Result_1 &= System/Man\\ &= \{Thirsty.g(hasCoin).InsertCoin.\\ &\quad PutDrink.GetDrink\}/\{Thirsty,\\ &\quad g(hasCoin), InsertCoin, g(noCoin),\\ &\quad KickMachine, PutDrink, GetDrink\}\\ &= \{Thirsty.g(hasCoin).InsertCoin.\\ &\quad PutDrink.GetDrink\}\\ Result_2 &= System/VM\\ &= \{Thirsty.g(hasCoin).InsertCoin.\\ &\quad PutDrink.GetDrink\}/\{PutDrink,\\ &\quad GetDrink\}\\ &= \{PutDrink.GetDrink\}. \end{aligned}$$

As the applying of our inconsistency checking rules,

$$Result_2 = VM,$$

but

$$Result_1 \neq Man.$$

Though the behavior of $Result_1$ is not equal to Man, we can say that the model of this system is inconsistent. The man cannot get a drink from the vending machine by kicking it, because the vending machine does not have behavior to support kicking action. In the case of removing the inconsistency from the system, we can remove the transition with label $KickMachine$ from the Man as in Figure 9, or add behavior to support $KickMachine$ action to the VM as in Figure 10.

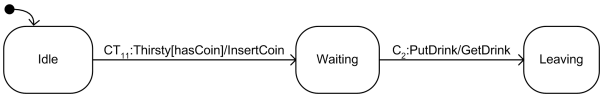

Figure 9: New thirsty man

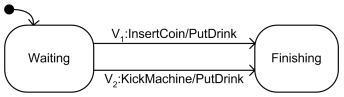

Figure 10: New vending machine

6 Conclusion

In this paper, we have presented the transformation rules from UML Statecharts to CREs. The transformation result represents in mathematical form which is suitable for applying model checking. We applied our transformation rules to the computer system, and show the simple inconsistency checking on the man and vending machine example. In the future, the other diagrams will be formalized in the same language.

References

Object Management Group (2003), Unified Modeling Language Specification Version 1.5. http://www.omg.org/uml. Accessed March 2003.

Vijay K. Garg & M.T. Ragunath (1992), Concurrent regular expressions and their relationship to Petri nets. Theoretical Computer Science 96:285-304.

Mitsutaka Okazaki, Toshiaki Aoki & Takuya Katayama (2002), Extracting threads from concurrent objects for the design of embedded systems. Ninth Asia-Pacific Software Engineering Conference (APSEC'02).

Bruce Power Douglass (2000), Real-Time UML developing efficient objects for embedded systems. Addison-Wesley.

J. Rumbaugh, I. Jacobson & G. Booch (1999), The Unified Modeling Language Reference Manual. Addison-Wesley.