Deriving Model-to-Code Transformation Rules at the Meta-Model Level

Lei Liu

Emanuel S. Grant

Department of Computer Science
University of North Dakota
liu@cs.und.edu
grante@cs.und.edu

Abstract:

The Unified Modeling Language (UML) has been widely accepted in the software development industry because of the benefits it brings to the task of specifying system requirements and design decision. However, models are not the final product of a software development cycle. Application code, which is compiled and executed, has to be derived from these models. Work on deriving transformation techniques at the meta-model level of abstraction is a necessary and valuable research area for the successful automatic generation of programming language code from graphical design models of software systems. By working at the meta-model level of abstraction it is possible to state with a high degree of certainty that all constructs at the model level of abstraction will be taken into consideration.

Introduction:

By using graphical and module design, the UML (Unified Modeling Language) models present systems' requirement in a graphical manner. Compared with other system design tools, UML models make it much easier for software engineers to understand software system requirements. Although the UML models are widely used in system design, they are not the final products for software engineering. Application code, which is complied and executed, should be generated from the UML models. Tools for supporting the generation of application code from UML models are not developed to the point where they can generate complete code from the design models. One possible cause of this problem is the difficulty of precisely stating the relationships between the model elements and the code, in the form of a program that has to be produced. In our work, transformation rules to go from UML activity diagram to C programming code will be explored at the meta-model level of both activity diagram and C language program.

The reason we choose the UML activity diagram for our research is because of its capability to express action flows and object interaction for an application system. C's relative simple program structure and small set of statements constructs, is the reason chosen as the target language for this research. A number of previous works on model transformation to executable program code have focused on the transformation at the model level. The techniques of these previous works have limited applications as they only address the transformation of specific instances of a model, and not all possible models of a particular type, into the chosen programming language code. The model level transformation rules limit reusability of the code generation technique, as the technique may not take into consideration all possible aspects of the model being transformed. For that reason, we focus on setting up the transformation rules at the meta-model level of both activity diagram and C programming language. By working at the meta-model level of abstraction it is possible to state with a high degree of certainty that all constructs at the model level of abstraction will be taken into consideration.

After organizing the common C programming statement into a meta model description (class diagram in UML), transformation rules from the meta model of activity diagram to the meta model of C programming language will be explored, e.g. we will give the rules from activity diagram to corresponding C program code. As a demonstration of the benefits of our work, we will present examples of the application of the transformation rules on a set of activity diagram models, which have been taken from different application domains. Code for these demonstration samples will be generated manually based on the rules generated on the meta models of activity diagram and C programming language. We will compile and execute generated code and see whether execution result may satisfy system requirement from the beginning.

Background:

The Unified Modeling Language (UML) is used to specify and document software entities at the analysis and design levels in software development. The UML defines several graphical models: class diagram, collaboration diagram, activity diagram, use case, sequence diagram and implementation diagrams. Class diagram captures static relation between different classes in a software system and expresses it in a graphical

view. An activity diagram shows the activity sequence of instances from different classes. All states in an activity diagram are actions and triggered by completion of the actions in the source state (i.e activity sequence). Usually, an activity diagram is attached with a classifier, such as a use case diagram and class diagram. It focuses on flows driven by internal processing.

There are four different layers for UML models: meta-meta model, meta model, model, user objects. A meta-meta model defines the language for specifying meta models. Meta-model is an instance of a meta-meta model and it defines the language for specifying models. A model is an instance of a meta model and it focuses on describing an information domain such as specified operations and attributes. User object (also called user data) is an instance of a model, which gives the values in the domains. Among these four layers, the meta-model describes all models for that category. That is the reason why we will work at the meta-model level of both activity diagram and C programming language.

Activity diagram

An activity diagram is a special case of a state diagram, which is a sequence of operations. A state in the sequence is triggered by the completion of the operation from a previous state. An example of an activity diagram is shown in Figure 1.

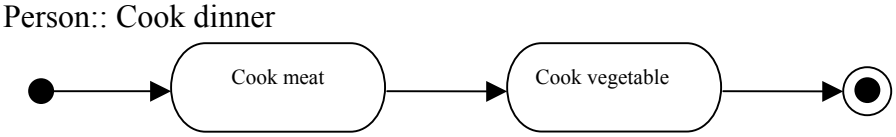

Figure 1: Example of activity diagram

There are several different states in this activity diagram.

Action state:

An action state is a state with an entry action and one or more outgoing transition to implicit event after completing the entry action. One example of action state is shown in Figure 2, where an action state is shown as a shape with straight top and bottom and with convex arcs on the two sides. Expression of the action state is placed in the middle of the graph and it need not to be unique within the diagram.

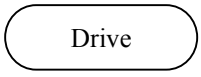

Figure 2: Action state

Sub activity state

A sub activity state is a group of nested action states, which cannot be exited until the last state has been reached. Sub activity state has the same symbol like that of an action state

with an additional icon in the right lower corner to identify it is a sub activity state. An example of sub activity state is shown in Figure 3.

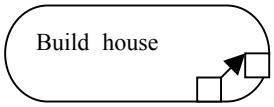

Figure 3: Sub activity state

Decisions

In order to indicate different transitions, which depends on Boolean conditions of the host object, in an activity diagram a guard condition is chosen to decide which transition will be taken. A diamond shape is used as the icon for guard condition and an example of guard condition is shown in Figure 4. [1]

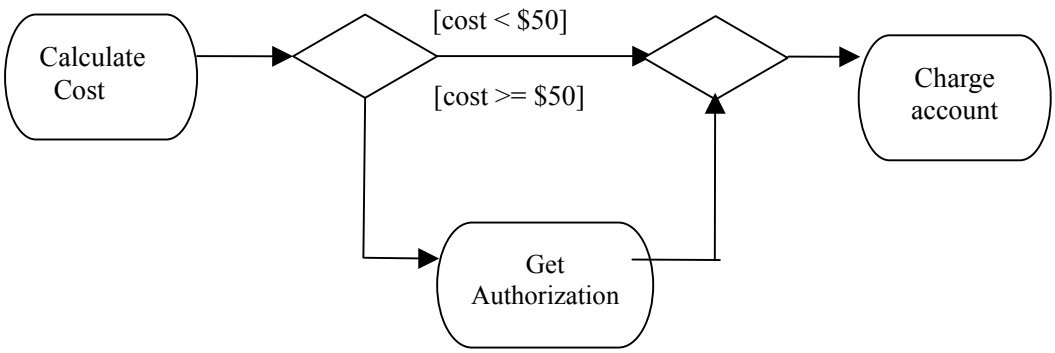

Figure 4: Decision and merge [1]

Call states

Call state is associated with a classifier and calls a single operation. A call state is the same as an action state along with the name of the classifier that hosts the operation in the parenthesis under it. Figure 5 gives an example of a call state. [2]

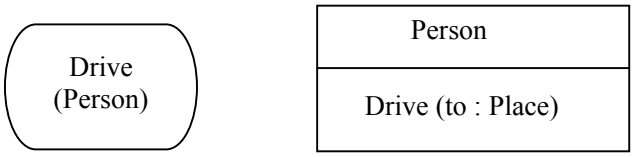

Figure 5: Call state and the operation it invokes

Addition to the action state, sub activity state, decision and call state, there are several other symbols/tools, which are used in activity diagram. Swim lanes organize action states and sub activity diagram into different categories. A solid lane separates each category and each action state and sub activity diagram is signed into one category. Action and object flow relation combines both action state and object (action related) in one diagram. Control icon indicates signal sending and signal receipt.

C programming language

As a general-purpose programming language, C "features economy of expression, modern control flow and data structures, and a rich set of operators". [3] Additionally, C program has small set of grammar and simple structure. It is a targeted programming language for this code generation research.

Meta model of C programming language

The work for this research work is at an early stage and only the foundation structures will be presented in this paper. The outline of the C program illustrated in this report based on work presented in *PURE C PROGRAMMING*. [4]

/*	*/	comment line
# include directives		
# define directives		preprocessor command
declaration statements		preprocessor command
prototype statements		
int main()		global variable declarations
{		main head
declaration statements		
prototype statements		variable declarations
initialization statements		function declarations
statements		variables initializations
return 0;		statements including function calls
}		return statement
/* user-defined functions */		
function name (parameters)		functions definitions
{		function header
statements		
}		function body

Table 1: C Program language structure

Transforming from activity diagram to C programming code will be based on the outline of C program listed above. Since we are working at the meta model of C programming language. The UML class diagram will be used to present the meta level model of the C program. (The UML class diagram is used to present the meta-model of all UML models.)

Rules for transformation

After developing up the meta-model of the C program, transformation rules for going from activity diagram to a C program will be explored. The rules are defined for the outline framework of a C programming code and not the complete code.

Related work

There are several related code generation tool development works. Kevin McLeish's work: transfers UML class and collaboration diagram to Java code by using XMI intermediate. [5] This work is at the model level instead of meta-model level of UML model and programming language. As mentioned previously, the meta-model level is the higher-level model for both UML and programming language, which will cover more aspects of UML model and programming language model comparing with specific models. Bart Verheecke has published a paper: specifying and implementing the operational use of constraints in Object-Oriented Application. [6] In that paper, constraint for transforming from model to specification other than generating code is emphasized.

Methodology

Transformation rules from activity diagram to C programming code will be done at the meta-model level. The meta-model for activity diagram exists in the UML specification. All necessary components for a C program will be organized into a class diagram of the UML model, which represents for the meta-model of C program structure frame.

Meta mode of activity diagram

Figure 6 shows the meta model of state machine diagram from which the activity diagram meta-model is extended [1].

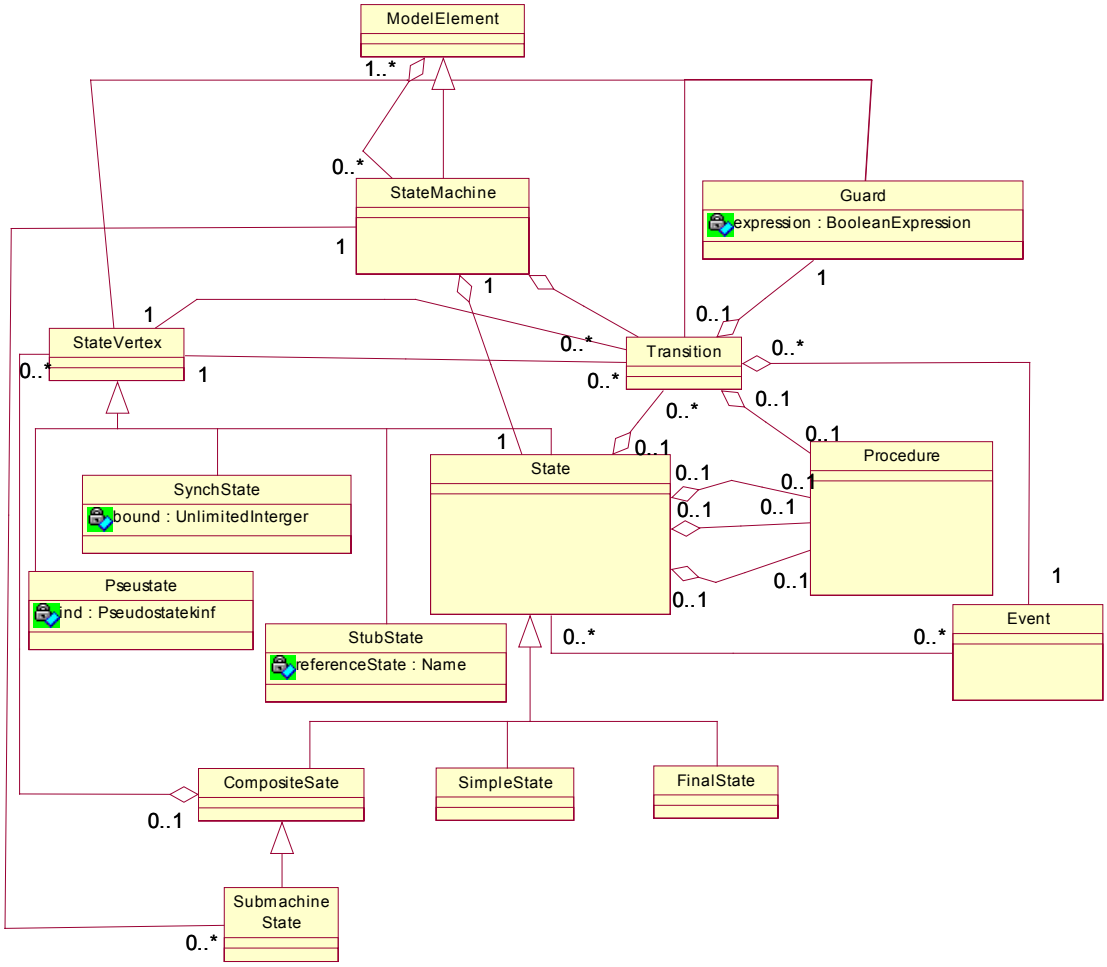

Figure 6: Meta model of the state machine

Figure 7 shows the meta-model of activity diagrams, which is an extension of the state machines of Figure 6.

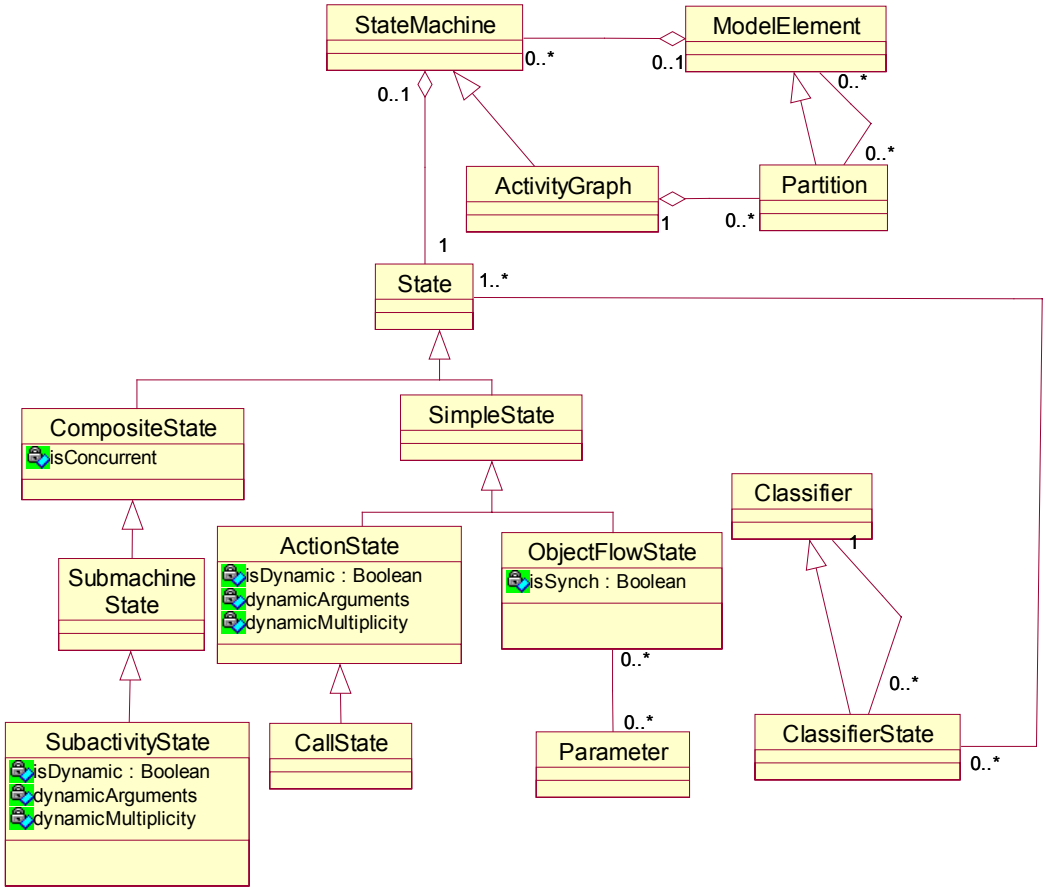

Figure 7: Meta-model of the activity diagram

All elements in activity diagrams are covered in this meta-model of activity diagram listed above. If we can transform each class instance in this graph to corresponding C programming code, we can transform all activity diagrams to C code. (e.g, each state in an activity diagram is an instance of one class in this graph)

Meta model of C programming language

Since this transformation work is focused on simple C program structure other than one hundred percent code, only those elementary C grammars are chosen for this work.

Basic elements of a C program include Directive Description, Statement, Main Function and User Functions. C programs process data by executing statements such as expression statement, selection statement, jump statement, compound statement, label statement and iteration statement. The relationships between those elements are shown in Figure 8.

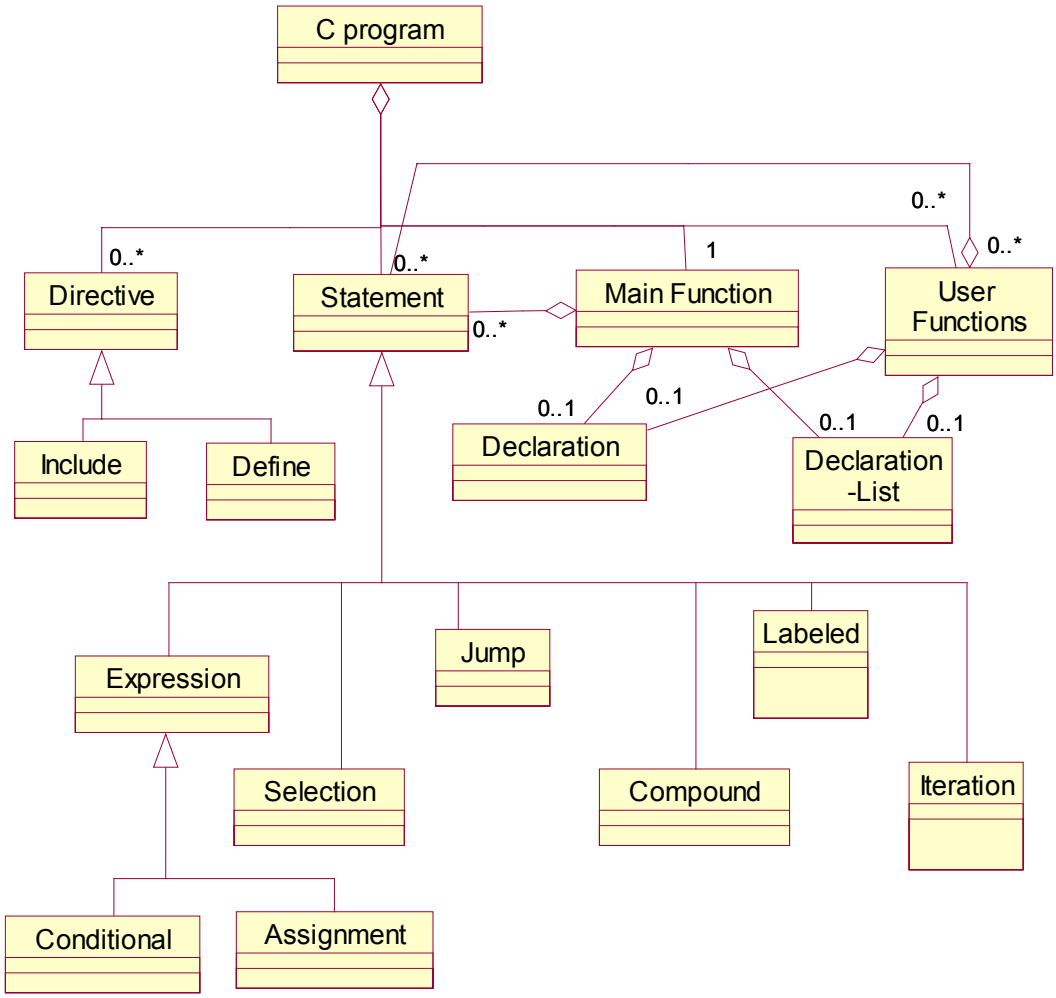

Figure 8: Meta model of activity diagram

From this graph we can determine that C program is composed of directive description, statement, main-function and user functions. # include and # define statements are generalized from the Directive (description). Expression, Jump, Label, Selection, Compound and Iteration statement are generalized from Statement. Combined and Assignment statements are generalized from Expression statement. Main Function and user functions are composed of different Statements and Declaration, and Declaration List. [3]

Transformation rules

Transformation rules will match elements in meta-model of the activity diagram to elements in the meta-model of the C program. This research work will explain only key elements in the meta-model of activity diagrams and C programs.

Each activity diagram instance can be transformed to a single C program function or several C program functions.

An example of an activity diagram is chosen from the UML specification [7] to help in explaining the transformation rules. Figure 8 presents the example activity diagram used

to explain the transformation rules.

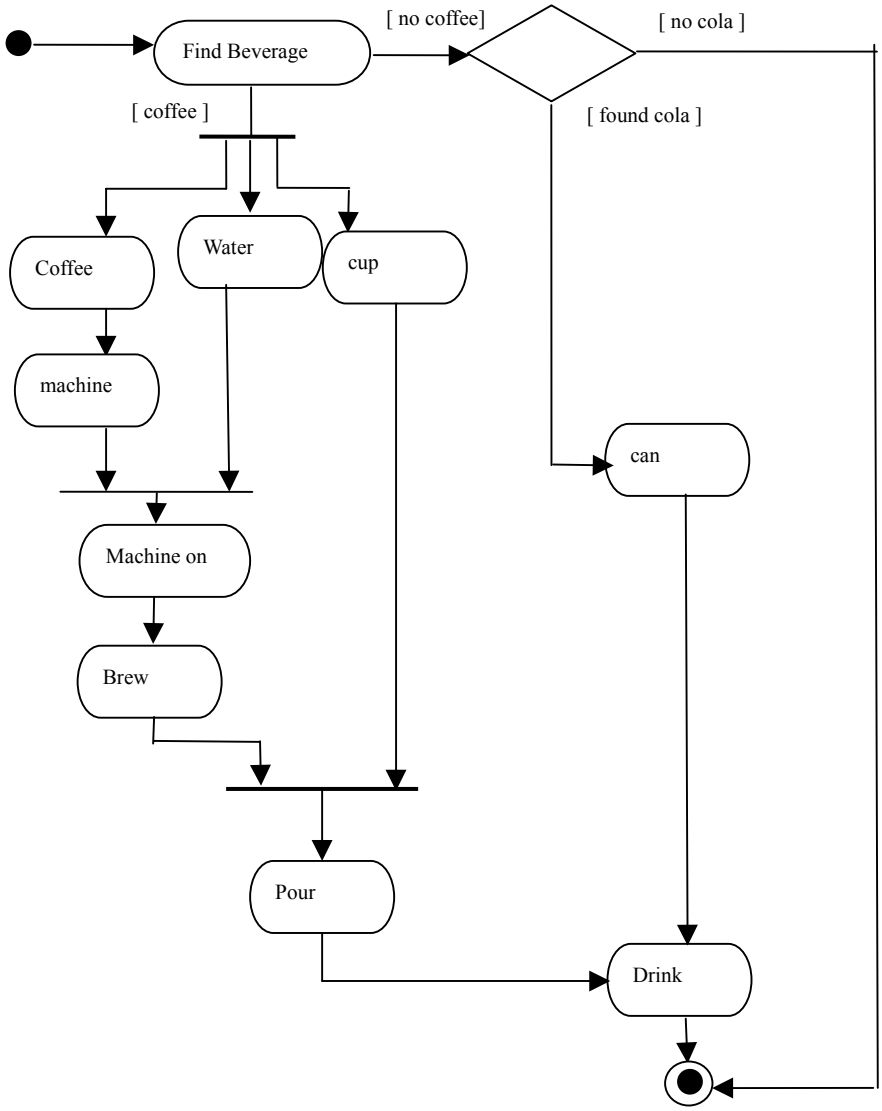

Figure 9: Example of activity diagram

This is the graph for a person to prepare beverage, and this sequence of activities will be attached with the classifier person.

The guard in the meta-model of the activity diagram can be transformed to a single expression statement in C program, since guard will decide path is to be taken. In the example above, the guard decides whether we have cola or not can be changed to *if (hascola)* hascola is the boolean to decide whether we have cola or not.

A state in each activity diagram is designed to finish one task, which can be transformed to one function. If the function is simple and can be expressed as a single one statement, the function can be transformed to one expression statement later. If the function is complicated and can't be expressed as a single expression statement, then a set of

statements is used in the transformation. The number of outgoing transitions determines how complicated the function may be. For example, two outgoing transitions can be transformed into *if then else* block. Outgoing transitions number more than two can be transformed into a switch statement block. After a function has been executed, the state of that classifier may be changed. In the example above, state Find Beverage can be transformed into function *findbeverage()*. The function *findbeverage* can be transformed to *if (hascoffee){} else{}* block. A *hascoffee* is one variable to indicate whether coffee is available or not.

A synchstate is followed by a set of activity states. We can transform this by following sequential functions, where each function is transformed from one state. States between two synchronization bars are executed from top to bottom and left to right. In the example above, between the first two synchronization bars, functions are executed on the order of: *pufcoffeeinfilter() putfillterinmachine() addwatertoreservoir()* then *getcupforbrew()*.

Unlike a simple action state, a sub activity state contains more than one state. This aspect can be transformed to user-defined function, which calls other functions inside the function body. At the end of each sub activity state, control is returned to the initial action state that called the composite state.

Applying transformation rules above, the example could be transferred to C programming code like:

```
Findbeverage{
  If (hascoffee){
     Putcoffeeinfilter()
     Putfilterinmachine()
     Addwatertoreservoir()
     Turnonmachine()
     Brewcoffee()
     Getcups()
     Pourcoffee()
     Drink()
  }
  Else{
    If(hascola){
      Getcansofcola()
       Drink()
    }
    Else{
  }
 }
}
```

Conclusion

The expected result for this research work is to define an algorithm for applying the transformation rules. With such an algorithm, it will be possible to develop an application to automatically produce a C program frame work. This C program frame work will then

be used to produce a working application.

By generating partial program, the time for software development is shortened greatly. At the same time, because the frame work is generated from UML visual model, compared with traditional writing code manually, the code generated from UML model is more reliable. Since adapting model to a new domain is much easier than rewriting programming code. Compared with programming code, models have more reusability.

The final goal of software engineer is to generate the complete code. By transforming frame code from UML model, we are one step closer to produce complete code.

Reference:

1. OMG UML specification P. 3-160.
2. OMG UML specification P. 3-161.
3. Programming language in C. Brian *W. Kernighan and Dennis M. Ritchie*. P.Xi
4. PURE C Programming. *Amir Afzal*. P.506 appendix E
5. Code Generation from UML Models using XMI. *Kevin McLeish* Master thesis, Colorado state university, Colorado, may 2002.
6. Specifying and implementing the Operational Use of Constrains in Object-Oriented Applications. *Bart Verheecke – Ragnhild Van Der Straeten.* 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Austrilia.
7. OMG UML specification P. 3-84.