Journal of Scientific and Practical Computing
Vol. 1, No. 2 (2007) 41–50

AUTOMATIC CODE GENERATION: MODEL–CODE SEMANTIC CONSISTENCY

ANDREW J. KORNECKI[1] and SONA JOHRI[2]

[1]*Computer & Software Engineering Department, Embry Riddle Aeronautical University, Daytona Beach, FL 32114, USA, kornecka@erau.edu*

[2]*BSC – Guidant Corporation, Saint Paul, MN 55112, USA, sona.johri@guidant.com*

Abstract:

Automatic translation or code generation of software models to code may help alleviate problems associated with manual coding effort. This paper emphasizes the importance of attaining a high level of assurance that the process of automatically translating model to code is correct. It illustrates modeling experiments performed using *Statemate* (*iLogix*) to establish a correspondence between model elements and code constructs. The research is a step towards achieving assurance of semantic consistency between model and generated code.

Keywords: Software Tools, Automatic Code Generation, Validation & Verification, COTS, Testing.

1. Introduction

Automatic code generation (ACG) helps to increase effectiveness of complex software production by reducing the cost and time associated with the coding effort. However, it is extremely important that the generated code faithfully reflects the model from which it is produced. The presented paper is an attempt to examine means to assess the semantic consistency between model and code so that sufficient reliance can be placed on ACG.

As syntax defines the structure of legal constructs of a modeling or programming language, semantics gives the meaning of these constructs. Both model and the resulting program can be represented as functions mapping inputs satisfying some properties into results satisfying other properties. The denotational semantics is based on constructing formal mathematical object expressing the meaning of the system. The operational semantics describe how a valid model (or a program) is interpreted as sequences of computational steps. These sequences then represent the meaning of the model (or program) [1].

In the case of automatic code generation, semantics refer to the meaning of modeling elements (symbols) and how they are interpreted. Dion in his paper [2] writes: "The semantics of textual or graphical formalism define the meaning of program written in it."

Software development tools represent the specification/design of a system in form of models using variety of graphical notations (statecharts, class diagrams, sequence diagrams, flowcharts, etc.). These models use specific symbols to define programming and run-time constructs. However, the interpretation of their behavior depends on the particular tool implementation. The same model implemented in three different tools may exhibit slightly varying behavior as shown in [3]. To make an argument that the model and code are semantically consistent, all the constructs of the model must be shown to have a corresponding construct in code (and vice versa). Also, the behavior of the model is the same as the behavior of the code when executed. This can be shown by either manual or automatic analysis of both: the model and the code from perspective of fulfilling all pre- and post conditions, invariants and assertions.

On the University of York exchange forum [4] Crocker writes: "if the model from which code is generated completely describes the required behaviors and is written in a notation with well-defined semantics, then there is no fundamental difference between this form of code generation and a compiler. Therefore, immature automatic code generators should not be trusted, just as in case of immature compilers". The semantic consistency is an important aspect in demonstrating that the code generated faithfully reproduces the intent of the model representing system design.

Several commercial tools are available to provide support for translation of formal or semi formal specification to executable code. Typical examples are: *Statemate* and *Rhapsody* from *iLogix*, *Tau* from *Telelogics*, *Matlab/Simulink/Stateflow/RTW* from *Mathworks*, *RoseRT* from *Rational/IBM*, *SCADE* from *Esterel Technologies* and many others. However, there is reason to distrust an automatically generated code: (a) the specification language and/or target language may lack formal specification, (b) the translation may not be formally defined, and (c) the translation tool may be incorrectly implemented. Therefore, the generated code cannot be blindly trusted unless there is an appropriate argument to the contrary. The paper outlines the methodology used to evaluate correctness of the translation from model to code. This research is another step in making code generation more acceptable for safety critical systems in regulated industry.

2. Background

It is acknowledged that the traditional approach of hand coding is not ideal and as software increases in complexity, this method and testing may be inadequate for embedded systems. Model-Based Development (MBD) methodologies and automatic code generation have emerged, shifting the focus of the software development process. MBD allows verification of the software specification at the model level and reduces manual coding. ACG also gives an engineer the opportunity to focus on the high-level design issues and on better understanding of the problem.

According to O'Halloran [5], "automatic code generation is viewed with suspicion, to some extent unfairly, partly because the 'development process' can be easily changed with little or no visibility to an auditor. Another reason is that if the automatic code

generator is a black box commercial tool then there is no human understanding of what is being generated. If the commercial tool changes every six months then generated code could be subtly different for the same input to the code generator." For ACG tool used for safety critical software, the modeling and target language must be simple and suitable for safety critical systems. Simplicity reduces the risk of developing a program whose meaning is different from its modeled specification. Besides simplicity the languages must also support predictability, security and boundedness.

The Whalen and Heimdahl paper [6] outlines a set of requirements for code generation to obtain higher level of confidence in the correctness of the translation process. They describe Requirements State Machine Language (RSML) and identify five principles of "trustworthiness": (a) formal definition of both model and code syntax and semantics, (b) translation to maintain the meaning of the specification (c) formal verification of the translator implementation (d) rigorous testing, (e) well structured generated code traceable to the original specification. The translator uses a simplified imperative target language Safety-critical iMPerative Language (SMPL), which does not support statements such as goto, break, continue and the pointers. This makes it more reliable for safety critical systems as compared to C and C++. Since the constructs in RSML have an exact mathematical meaning, it is possible to directly perform equivalence proofs between these constructs in RSML specification and their equivalents in SMPL [6].

One of the ways to perform automatic code generation is through code generation templates. For example, *Codagen Architect* tool enables the transformation of UML models directly into code (C#, Java, C++). The tool allows the user to create code generation templates using a template editor. The templates represent the building blocks of code generation. In the tool, the code generation logic (transformation and validation rules) is composed of XML tokens that are inserted into the template by selection from a dynamic menu that displays tokens that are appropriate to the current context [8].

Another approach is used by *SCADE* [2,9] (Safety-Critical Application Development Environment), a modeling and ACG tool by the *Esterel Technologies*. *SCADE* uses high-level graphical notations for modeling: data flow blocks and state machines. The model operations are based on declarative, synchronous paradigm, where system output depend on inputs and state for each of the time steps. Internally, *SCADE* blocks are represented as textual synchronous language *LUSTRE*. Subsequently, *SCADE/KCG* code generator automatically produces simple *C* code that does not support loops, recursion, jumps and dynamic variables. The translation process from *LUSTRE* to *C* is based upon scientifically proven algorithms that the code generator directly implements.

ACG has found applications in some industries. *SCADE* has been used for the development of critical software embedded in the Airbus aircraft. "The *SCADE* Code Generator (KCG) is qualified for DO-178B Level A, the highest level of assurance, enabling this technology to become a de-facto standard in civilian avionics" [2]. The qualification enables developers to eliminate low-level testing of *SCADE*-generated code and thus helps in 30-60% reduction of the project costs. The *SCADE* code generation tool

has also been recently certified as a product under IEC 61508 (SIL4) for use in automotive industries [7]. *SCADE* enables the immediate creation of production-quality, embeddable code.

3. Model Analysis

3.1. *Approach*

To verify that the behavior of model constructs is the same as the behavior of the corresponding code constructs, we identify the relevant model and code constructs/primitives. This information would provide the framework for proposed work to establish semantics consistency between model and code. An experiment was conducted, using *Statemate*, to illustrate the proposed approach. The tool's modeling notation allows developer to represent both static structure and dynamic behavior of the application. The tool was used to create a set of models reflecting the supported constructs. *C* code for these models was generated using the ACG feature of the tool. The code components were analyzed manually to determine their correspondence to the model. The objective has been to find answers to the following questions:

1. How are states, data, events and activities represented in the code?
2. How the names of model components correspond to the names in the code?
3. How are transitions between states represented in the code?
4. How is concurrency in the model (between statecharts) addressed in the code?

Each of the model constructs supported by the tool has been modeled in isolation (except the cases where the complete isolation was not feasible: e.g. a top-level state must be associated with an activity). A relation table between the model and code elements is presented below. Table 1 presents all the model constructs identified in the selected tool, the way how each construct is translated in terms of the generated program constructs, and examples of the code snippets representing specific model constructs.

Another experiment was conducted to verify the correspondence of the behavior of the model and the generated code. The model was supplied with inputs during simulation to observe the sequence of transitions from one state to another (and the related actions). A control flow graph was created for the code generated from the model. This graph was traced for the same inputs and the sequence of transitions was analyzed. This limited experiment has been designed to show behavior equivalency between the model and the generated code.

Table 1: Relation between Model and Code Artifacts

Model Element	Code Element	Example
Internal Activity represents a logical view or organization of a system; it interacts with the external environment receiving input stimuli and producing signals consumed by the environment	Unique identifier declared as an extern activity	**Model:** Activity KEEP_TIME **Code:** *extern activity KEEP_TIME*
Event is a trigger and/or condition that defines the criteria for a change in system state	Unique identifier declared as an extern event	**Model:** Event INCREMENT_HOURS **Code:** *extern event INCREMENT_HOURS*
Data is an integer, real, string, bit, bit-array, record, union or user-defined type	Unique identifier declared as an extern variable	**Model:** Data-item HOURS **Code:** *extern int HOURS*
Control Activity represents a link between an internal activity and a statechart **Basic** – one variable/procedure **Non-Basic** – more than one variable/procedure	One or more Variables of type Enumeration One or more Procedures	**Model:** Basic Control Activity CLOCK_CMTL **Code:** *typedef enum {notaChart_CLOCK_CMTL, CLOCK_CMTL} tpChart_CLOCK_CMTL_states;* *void exec_Chart_CLOCK_CMTL()*
State is a condition (or mode of operation) of the system in particular point in time **Basic** – a state that has no children or sub-states **Non Basic** – a state that has sub-states	One Constant (basic) One Constant One Variable of type Enumeration One Procedure (non-basic)	**Model:** Basic State SET Non Basic State ON with two substates SET and RUN **Code:** *#define conSETst 0* *#define conONst 0* *typedef enum {notaONst, SET, RUN} tpONst_states;* *void exec_ONst()*
Basic Transition (Transition with Action) where **Action** specifies what to do as a consequence of an event occurring	Three Statements (two notify and one new state assignment; plus additional action statements)	**Model:** Basic Transition from source state SET to target state RUN **Code:** *notify(scope_id, conSET,FALSE);* *notify(scope_id, conRUN,TRUE);* *ONst_isin = RUN;*
Conditional Transition represents the transition between states based on specific guard condition	*If* block containing three statements (two notify and one new state assignment)	**Model:** Transition from source state SET to target state RUN only when condition POWER is true. **Code:** *if (POWER) {* *notify(scope_id, conOFF,FALSE);* *notify(scope_id, conONst,TRUE);* *CLOCK_CNTLst2_isin = ONst;* *}*

3.2. *Static Consistency*

The section elaborates on modeling constructs provided by the tool. The mapping between the model and code elements indicate that there is traceability between model and code and that the constructs represented in the model have corresponding constructs in the code. However, *Statemate* does not support concurrency. Two states that are defined as concurrent in the model will be executed in a sequential manner. The tool does not implement two concurrent states as different threads in the code generated.

3.2.1. *Internal activities and events*

Internal activities and events are defined external to the file in which they are used. They use keyword 'extern' for declaration. For example, the activity *KEEP_TIME* and event *INCREMENT_HOURS* are defined as follows in the code.

```
extern activity KEEP_TIME;
extern event INCREMENT_HOURS;
```

The event is defined as a Boolean and activity is defined as a structure in types.h file.

3.2.2. *Data-items*

Data is represented as external variable. It is defined using keyword extern. For example data items *HOURS* and *MINUTES* are expresses as following in the code.

```
extern int HOURS;
extern int MINUTES;
```

3.2.3. *State*

A *Basic State* is defined by a constant reflecting its name (con<start state name>). A *Non-Basic State* is a state that is further decomposed into sub states. Every non-basic state has an *EXEC* procedure that activates all the state-logic within a single execution cycle. The procedure takes care of in state transitions, static reactions, and activation of sub-states. Also every non-basic state has a status variable that indicates what sub-state is currently active. The status type is an enumerated type. The variable is in the form <*statename*>*isin*. Every state (basic or non basic) has a constant associated with it in the form *con*<*state name*>. For an example, see the code above *exec_CLOCK_CNTL*.

3.2.4. *Transitions*

In a transition, the parent state status variable is changed to indicate the activation of the target sub-state. The constant associated with source state (*con*<*start state name*>) is made false and the constant associated with target state (*con*<*target state name*>) is made true. A *Basic Transition* (within state *ON*) from source state *SET* to target state

RUN will produce the following code. A *Conditional Transition* would have the code embedded within *if* block.

```
notify(scope_id,conSET,FALSE);
notify(scope_id,conRUN,TRUE);
ONst_isin = RUN;
```

In a *Transition with Action*, in addition to statements for a basic transition, the statements specified as actions in the model are also executed. The Conditional Transition contains a condition that must be satisfied for the transition to occur from the source state to the target state.

3.2.5. *Control Activity*

A control activity represents a link between an internal activity and a statechart. The statechart represents behavior of the internal activity. The following example model shows the statechart associated with *CLOCK_CNTL Non-Basic Control Activity*.

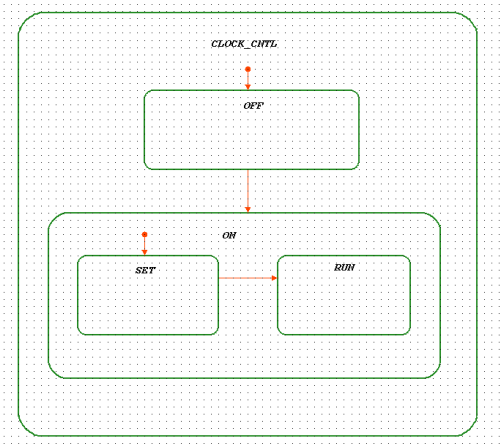

Fig. 1: Statechart of Non-Basic Control Activity *clock_cntl*

A control activity has a corresponding procedure in the code called *exec_Chart_<control activity name>*. It also has an associated variable that indicates whether the control activity is in active state or not. The variable type is enumerated type. Variable name has form, *Chart_<control activity name>_isin*, in code. The procedure *exec_Chart_<control activity name>* describes what happens when an activity is executed. It controls the activation of the statechart associated with the control activity.

A non-basic control activity is a control activity that has more than one state in the statechart it encompasses. A non-basic control activity has an additional procedure associated with it in the code. The module name is of the form *exec_<control activity name>*. This procedure implements the behavioral logic as described in the statechart encompassed within the control activity. It also has an additional variable of the form *<control activity name>_isin*. It indicates what state in the statechart is currently active. The variable type is enumerated type. The enumeration constants for this enumerated

data type are the names of the states in the statechart and default constant indicating the statechart is not active.

```
typedef enum {notaCLOCK_CNTL, OFF, ONst} tpCLOCK_CNTL_states;

typedef enum {notaChart_CLOCK_CNTL, CLOCK_CNTL}tpChart_CLOCK_CNTL_states;

tpCLOCK_CNTL_states  CLOCK_CNTL_isin = notaCLOCK_CNTL;
tpChart_CLOCK_CNTL_states
Chart_CLOCK_CNTL_isin = notaChart_CLOCK_CNTL;

void exec_CLOCK_CNTL()
{
                if(CLOCK_CNTL_isin ==  OFF ) {
      notify(scope_id,conOFF,FALSE);
      notify(scope_id,conONst,TRUE);
      CLOCK_CNTL_isin  =  ONst;
                        }
} /* exec_CLOCK_CNTL */

void exec_Chart_CLOCK_CNTL()
{
      switch (Chart_CLOCK_CNTL_isin) {
                case  notaChart_CLOCK_CNTL:
      notify(scope_id,conCLOCK_CNTL,TRUE);
      Chart_CLOCK_CNTL_isin  =  CLOCK_CNTL;
      notify(scope_id,conOFF,TRUE);

    CLOCK_CNTL_isin  =  OFF;
                break;
                case  CLOCK_CNTL:
      exec_CLOCK_CNTL();
      break;
                default:
      break;
      }
} /* exec_Chart_CLOCK_CNTL */
```

3.3. *Dynamic Consistency*

To provide illustration of dynamic behavioral consistency, another experiment was conducted. A model was created with a statechart representing simple up-down counter consisting of three states (*CHECKING, UP_COUNTER* and *DOWN_COUNTER*). Transitions occur from one state to another occur depending on the value of the input variable provided by the user during execution.

The control flow graph of the generated program shows the sequence of execution of the code. Suppose the value of input variable C is 2. The code executes statements for transitioning from *CHECKING* state to *UP_COUNTER* state. It increases the value of variable N to 10 and then executes code for transitioning to *CHECKING* state from *UP_COUNTER* state. During this transition it sets the value of C to -1. Next, code for transitioning from *CHECKING* state to *DOWN_COUNTER* state is executed. The value of N is decreased until it is zero and then transition from *DOWN_COUNTER* to *CHECKING* is executed. During this transition the value of C is set to 1. Again, the

transition from *CHECKING* to *UP_COUNTER* is made and the same set of statements is executed in the code. The model is provided with the same input (C=2) during simulation. It is observed that the same sequence of transitions occur between *CHECKING, UP_COUNTER* and *DOWN_COUNTER* states.

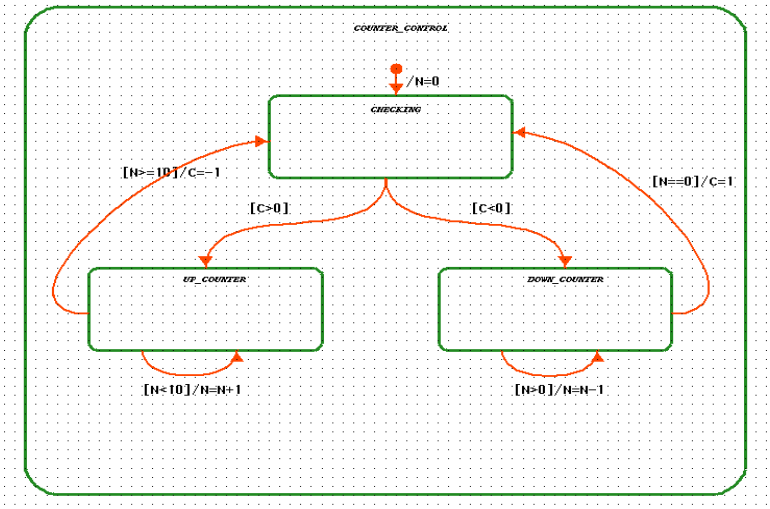

Fig. 2: Statechart showing up-down counter

The same experiment is conducted using the value of input variable C set to –1. It is observed that the sequence of transitions in the model and code are the same. This implies that there is behavioral equivalence between model and code.

The issue of non-determinism has huge impact on safety of the software product. In Statemate, non-determinism occurs when two transitions are triggered from a common state at the same time. In this case, the model simulation informs the user that non-determinism was detected and will let him or her select the transition to be executed. On the other hand, the generated code will select a transition arbitrarily and upon specific request will also issue a message that non-determinism was encountered.

The described experiment is based on a very simple model. It is proposed that during the future research work more complex models are created and used for experiments.

4. Future Work

The objective of the research is to verify semantic consistency between model and code generated using ACG tools. The mapping identified between model constructs and code segments explains how a specific construct in model is reflected in code. This information is important when analyzing whether the behavior and meaning of a model construct is the same as its corresponding code segment. The semantic consistency can be confirmed if the sequence of execution of transitions between states in the model is the

same as that represented in the code. Models in different configurations and more formal evaluation of pre/post-conditions and invariants can be used to address this concern.

The future work also includes identifying several popular software development tools, with ACG based on various principles (formal, semi-formal, frames, full translation) and creating a map between model and code elements as shown in the paper. This information can then be used to perform a range of experiments to check semantic consistency. The purpose of using more than one ACG tool for future research is to ensure that the results are valid for ACG technique in general and not a particular tool.

The research has been exploring the use of COTS testing tools to identify the resulting code coverage in terms of the segments/lines of code executed for a specific test case i.e. defined combination of input. The same input data is applied to the model to verify the sequence of model transitions and their consistency with those observed during the code execution.

Acknowledgments. Acknowledgements are due to Boston Scientific/CRM Guidant, Inc. St. Paul, MN, for their support of this research.

References

1. L. Allison, A Practical Introduction to Denotational Semantics, Cambridge University Press, 1986
2. B. Dion, Correct-By-Construction Methods for the Development of Safety Critical Applications, SAE World Congress, Detroit, MI, March 2004
3. A. Kornecki, J. Erwin, Characteristics of Safety Critical Software, 22nd International System Safety Conference, System Safety Society, Providence, RI, August 2004
4. D. Crocker, posting on Automatic Code Generation in Safety Critical Software Development. 23 Jan. 2004. <http://www.cs.york.ac.uk/hise/safety-critical-archive/2004/0027.html> retrieved 2 Oct. 2005
5. C. O'Halloran. Issues for the Automatic Generation of Safety Critical Software, 15th IEEE International Conference on Automated Software Engineering, 2000, p. 277,
6. M. Whalen, M. Heimdahl, An Approach to Automatic Code Generation for Safety-Critical Systems, 14th IEEE International Conference on Automated Software Engineering, 1999, p. 315
7. J. Gärtner, Code Generator Schemes Aid Safety-Critical Code Development. COTS Journal, April 2005.
8. Codagen Architect, <http://www.codagen.com/products/architect/faq.htm> retrieved 27 Nov. 2005
9. W. Hohman, Supporting Model-Based Development with Unambiguous Specifications, Formal Verification and Correct-By-Construction Embedded Software, SAE World Congress, Detroit, MI, March 2004