Software Development with Automatic Code Generation: Observations from Novice Developer Viewpoint

Farahzad Behi
Embry Riddle Aeronautical University
Computer & Software Engineering
Daytona Beach, FL 32114
Phone: 1-386-226-6454
Fax: 1-386-226-6678
behif@erau.edu

Andrew J. Kornecki
Embry Riddle Aeronautical University
Computer & Software Engineering
Daytona Beach, FL 32114
Phone: 1-386-226-6888
Fax: 1-386-226-6678
kornecka@erau.edu

Abstract— *Modern software development methodologies use Model Based Development (MBD) in design and verification practices. A number of software design tools support the use of modeling throughout the development lifecycle. Using appropriate notation the system model is build and verified. Subsequently the software source code can be automatically generated from the design artifacts. The variety of tools on the market and deceptive vendor claims about tool applicability and ease of use leaves industry confused. To explore this, four students of ERAU Master of Software Engineering program were given an assignment: to learn how to use specific software development tool and subsequently develop small project while collecting observations. In addition to give the student opportunity to explore modern tools and technologies, the objective of this exercise was to collect observation on the leading software development tools usability from the perspective of a young software developer.*

Keywords: Software Engineering, Software Tools, Model Base Development, Automatic Code Generation, Software Process.

1.0 Introduction

Software development tools have growing impact on the effective and efficient development of software-intensive systems. Modeling can be used throughout the entire system development lifecycle. Modern software development methodologies rely on building conceptual models of the software system and analyzing the models (via simulation, animation, and semi-formal or formal model checking) before translation to conventional programming language format. Recently, the concept of using Model Based Development (MBD) has become central to design and verification practices. Supporting the approach software design tools provide additional Automatic Code Generation (ACG) capability. ACG can be treated at as the next logical step in specifying and describing a software system in the progression: from machine language to assembly language to algorithmic high-level language to object-oriented high-level language to graphic modeling language. The modeling approach provides higher level of abstraction through use of visual notation as well as formalized modeling languages. The higher levels of abstraction enable developers to focus on important features and behavior of the system, allowing the tool to handle the implementation details.

There are several categorization methods for design tools with ACG functionality. Their input representation may be textual or graphical. A tool may be capable of producing either a framework of the code that needs to be filled with code in specific language or a fully functional program. A tool may also restrict the format of the generated product, or it may provide a wide variety of output options, such as formal style or language. The relative importance of such criteria may be a factor to determine the relative value of a tool for the specific project and organization. The objective of this study was to collect observation on the leading software development tools usability from the perspective of a novice software developer. The tools selected for the presented study, can be categorized into two groups: (a) those using a function-based, block-oriented approach, and (b) those using structure-based, object-oriented approach.

With the function-based block-oriented approach, the initial design is initially specified in a form of diagrams representing the system functions (comparative and mathematical symbols, control blocks). The diagrams are then used to simulate the system behavior and evaluate its performance. Once the user is satisfied with the design, an automatic code generator translates the model and the resulting target source code is produced that reflects the rules specified in the diagrams. Typically, no code needs to be written and the tools of this category are popular between domain specialists (control, system, mechanical, aerospace, and civil engineers).

With the structure-based, object-oriented approach, the initial design structure and behavior are documented as a collection of models using object-oriented diagramming notations, such as class diagrams, sequence diagrams, state diagrams, etc. The specific behavior is represented in terms of events with actions defined in the supported computer language. Typically, computer scientists and computer/software engineers are primary users of such tool. As in the functional approach, the resulting diagrams are used to validate the system behavior through animation or simulation and then to generate the target source code. Recently, the standardized Unified Modeling Language (UML) notation is widely used.

Considering the above categorization and availability of tools, the following selection was made

- In the Software Engineering paradigm (structural: object-oriented):
  - o Rhapsody (iLogix) [1]
  - o Esterel Studio (Esterel Technologies)[2]
- In the Control Engineering paradigm (functional: block-oriented):
  - o MatLab (Simulink, Stateflow, Real Time Workshop) from MathWorks[3]
  - o Scade (Esterel Technologies)

After tool assignment, the students engaged in the project. The following sections present the observations collected after the project completion.

2.0 CASE STUDY

To reduce the bias related to lack of familiarity with the tool, the project was conducted in two phases: (a) learning phase and (b) execution phase. The learning phase included familiarization with the environment and the tool finalized with developing a simple system simulating an electric hairdryer defined by four simple requirements. The execution phase included building slightly more complex system simulating a microwave oven defined by ten requirements. In this phase the data and observations were collected. Six steps of the process used in the case study are presented in Figure 1.

Experiment Process:
Step One: Collect information about the specific tool from literature (dealing with the concept of operation, background, tool/vendor history, and industry feedback) while exploring the tool availability license currency, etc.
Step Two: Study the assigned tool and work with demo/tutorial for familiarization with the tool (which includes also porting of the resulting software to the VxWorks[4] target)
Step Three: Attempt to complete the throw-away example model (HAIR DRYER) to gain experience and learn what you need to know to attempt the project (collect the effort data)
Step Four: Only then start the actual project to build the evaluation model (MICROWAVE) while collecting data on effort, defects, code size, code performance, usability, and engineering observations
Step Five: Compile your observations and data into a brief final report
Step Six: Prepare brief presentation and a demo of your system.

Fig. 1: Experiment Process

3.0 RESULTS

The following sections describe four tools from the developer perspective. The description includes brief information about the tool, its operations and use, and general observations.

3.1 Esterel Studio

Esterel Studio is a development tool used to assist in building and verifying design control software for embedded systems. The foundation of the Esterel studio is synchronous Esterel language using a finite state machine approach to represent the control mechanism. This control mechanism can be implemented in hardware or software using the same specification. This hardware-software equivalency is the trademark of the approach [2]. More advanced users have the option of developing systems using more flexible, but more difficult internal language, while novice users can use the graphical constructs only. For greater flexibility, the two notations can be mixed. In either case, the starting point is creation of a workspace and a project, creation of a default or new model, creation of inputs and outputs, user-defined types, functions and procedures. Then the model can be populated using a combination of the three state views: (a) a graphical state in which the developer can draw pure state diagrams, (b) a macro state this is where the developer can mix text and graphical notation, and (c) a pure textual state block with inline Esterel code. Figure 2 presents a screen shot of the project using graphical state view.

The completed model can be checked with the model checker, which translates the model into its underlying notation amenable for formal analysis. The results of checking are displayed in the tool.

After this process is finished the model can now be simulated and/or translated into C- source code. A graphical representation of the model allows developer to observe the simulation progress step-by-step.

The tool provides documentation on interfacing with the generated C-source code. According to our interpretation of the documentation, Esterel has a graphical construct for separating state machines and allowing them to run concurrently. But examination of the generated code does not show concurrency.

After finishing the microwave project the generated source file were 537 LOC with the executable size 12kB.

The tool documentation is easy to understand but hard to find. The documentation for the installed, new software version was abridged and lacking detail. The installed system also had an earlier version of documentation, which was more detailed but often not consistent with the used software version. It would be nice to see more examples on how to pass data with events.

One of the problems with the tool has been its inability to recover gracefully from errors. The Esterel Studio software crashed and closed unexpectedly several times during operation. During the model checking there are error warnings that require more explanation from a usability perspective. One such error/warning is a cyclic warning showing where the cycles exist and how they are related but no explanation of how to fix them. Another issue is the software file/platform compatibility issues. This occurs when developing a model on one system and then trying to open and run the model as a simulation on another machine – it gives a “time not correct” error.

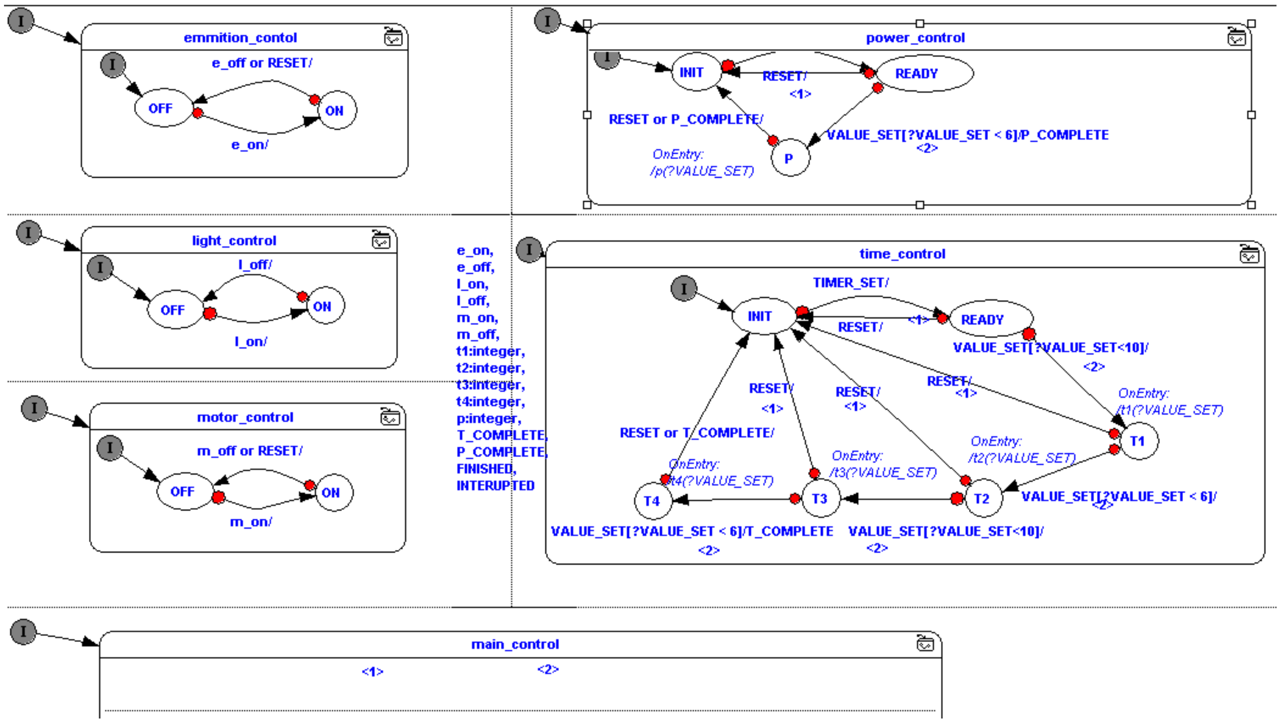


Fig. 2: Esterel Studio State Diagrams

3.2 *Rhapsody*

iLogix Rhapsody is a CASE - Tool for embedded systems software development. Rhapsody is claimed to be "The industry's leading Model-Driven Development environment based on UML 2.0 that allows full application generation for embedded systems and software developers." [1]. Through Rhapsody’s MDD approach, a developer can rapidly target the platform independent application model to a real time embedded operating system, reducing development and integration time. Rhapsody naturally lends itself to an iterative design approach where the software can be constantly executed, simulated and validated in a native environment, then downloaded to the embedded target.

The tool provides support for all UML 2.0 constructs. In addition, the tool is capable of auto-generating code in C/C++, Java, and Ada, which requires appropriate compilers to be installed on the system. The latest version allows developer to use both functional and object oriented design methodologies in one environment.

Rhapsody comes with an extensive set of tutorials and manuals however complexity of the tool exceeds capability of documentation to explain the required details. Documentation may, for example, show a sequence of steps necessary to accomplish certain functionality or feature. However, long learning period is required to reach mastery over Rhapsody complex features. The case study required that the model be developed in the Rhapsody for C. Figure 3 presents two elements of Rhapsody notation: File Diagram and Statechart.

The main power of the software is its flexibility to use any or all of these UML notations to develop the model at any level of granularity and from different but consistent viewpoints. While this is extremely useful, accommodating the different development tastes among developers and organizations, it is also a major challenge for novice developers or those with limited UML exposure. The challenge increases when attempting to use object-oriented modeling approach for functional project development leading to target C program. The recent Rhapsody version helps by supporting procedural development, where files replace objects and classes. A continuing practice with the tool is necessary to respond to such challenge.

The tool is an excellent platform for requirements analysis. Use case can be developed showing they trace back to requirements. Block Diagram and a Software Realization Diagram represent the software structure. These two diagrams were selected for this project following the tutorial guidance. Certainly other diagrams could have been constructed. Rhapsody analysis models provide minute details of the system/software operations. The choice of which diagrams to use or how far to go with the details in modeling ultimately depends on the nature of the software and developers’ experience.

Another good feature of Rhapsody is that while model views can be constructed independently of each other, the tool allows for their inter-linking and realization of changes among the views. Object Model Diagrams were used in the case study to model the system component at the design level to realize the files that make up the system. The tool's modeling flexibility allows the developer to either build upon structural design detail, such as object, by implementing functions and variables, etc. or to switch to behavioral view such as sequence diagrams. It also allows the developer to start with any model view or diagram and proceed later on to refine the views as the model matures in development. A component is a unit of executable code, and must be created in Rhapsody before the software is able to generate code for the model. Each component has a configuration defining the module views (diagrams, charts) and elements (events, operations, triggers) to be used in generating the code. The views or the elements must be associated with an object at the module level (file, class, object). Among many possible configurations in a project, the active configuration is one used during code generation. The tool generates code of substantial size. The Case Study code was over 4KLOC of the code.

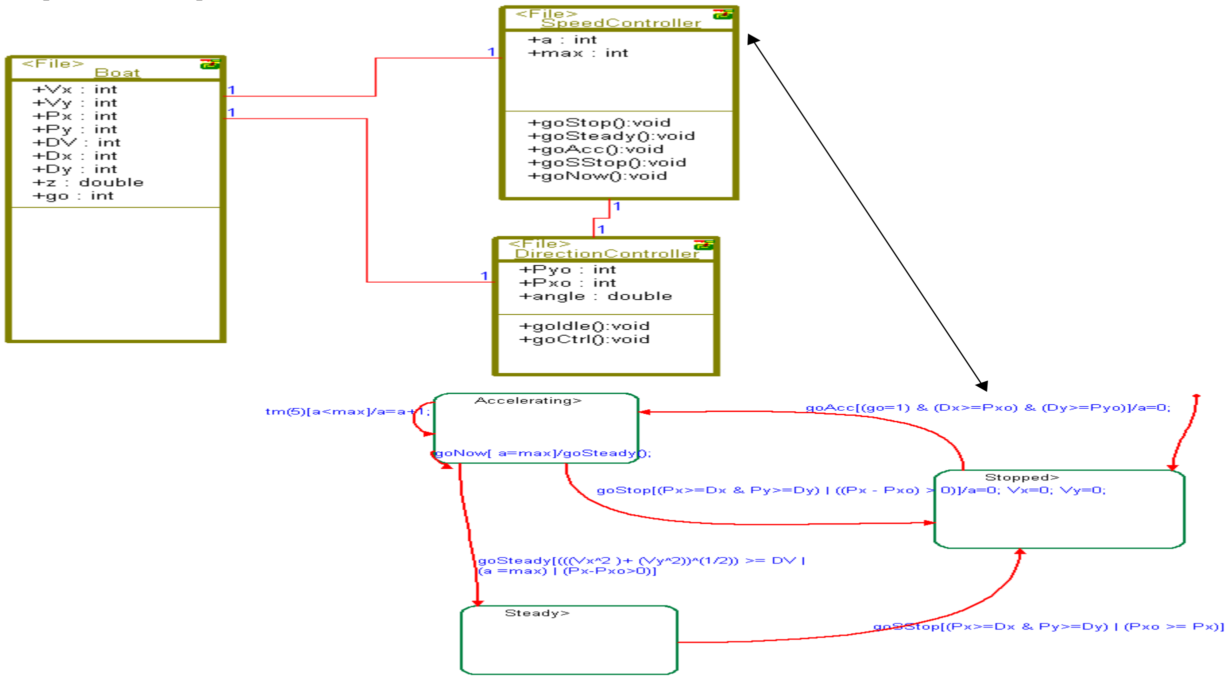


Fig. 3: Rhapsody in C File and Statechart Diagrams

The disadvantage is larger software that may be more cumbersome to debug, and may have some influence on performance in certain applications. However, Rhapsody also provide excellent debugging tools, where an error can be double clicked in the output screen of the software and the developer is instantly taken to the specific location of the error in the particular file in which it was reported.

Rhapsody is a great tool for modeling real-time embedded systems with the focus on ensuring safe design concepts. The tool offers incredible flexibility in how a model may be developed. The drawback to the above is the substantial learning curve that precedes a level of mastery where such gains may be realized. Rhapsody is a very complex tool - the software may be integrated to work with various platforms and other third party tools, such as CORBA, VxWorks, RapidRMA and many others. It seems to be impossible to master all of its capabilities in the course of 2-3 moths. Practice however is the best method at heading in that direction.

3.3 Scade

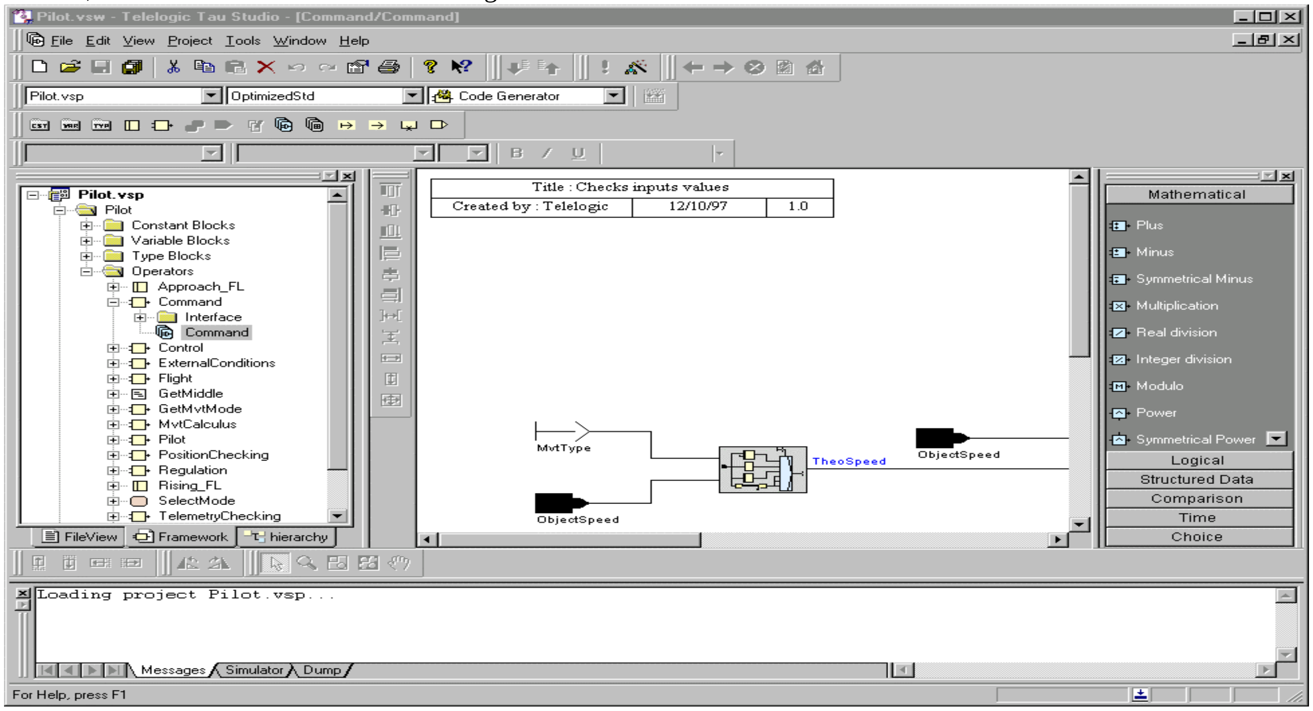
Esterel Technologies has commercialized SCADE to implement a correct-by-construction methodology with a sophisticated tool suite. The vendor literature reads: "SCADE Suite implements a unified conceptual model of embedded computation backed by three strong technical cores: the use of specific high-level rigorous graphical and textual language, compiling algorithms for correct-by-construction implementation, and formal testing and verification techniques." [2].

The tool is complicated but powerful. It is capable of handling different types of control logic. Graphic modeling of the designs in form of state diagrams, combinational and sequential logic constructs allows the designers the flexibility to describe the nature of the system and the events that control its behavior.

Behind the graphic notation, resides a formal programming language Lustre, a formally defined synchronous language for the development of safety critical control software. SCADE incorporates many other features such as the model-checking, abstract interpretations, testing capabilities, simulation, and debugging. Using these tools or SCADE capabilities in conjunction with Lustre's constraints, the software designed and generated by SCADE may be verified against any desired standard.

The entry point to SCADE is the Editor with main sections: the hierarchy window containing the files and components within the model, the editor window itself where the model is created and separate entities are connected, and the build window and message center for displaying all information to the developer (Fig. 4)

Systems are designed for completeness and lack of ambiguity using block diagrams to represent functional behavior and safe state machines for event-driven behavior. In either methodology, SCADE provides the designers with model checkers for syntax and semantic errors. Additionally, the SCADE Editor may perform methodological checks to maintain correctness during development .



Title : Checks inputs values		
Created by : Telelogic	12/10/97	1.0

```
Loading project Pilot.vsp...
```

Fig. 4: SCADE Hierarchy, Editor, and Build Windows

The state machine taking inputs and producing outputs provides the user with a means of describing the behavior of the system. The tool provides the developer with a graphical debugging and simulation feature. This feature is the virtual prototype of a system. It is a pre-build, non-physical prototype that has the same functionality of the intended system; the dream of a software engineer, not needing to rely on hardware performing properly during development. During the design phase of the system, the tool provides the ability to validate any algorithms in the system. This stage also allows the user to capture and verify any system-level safety properties producing counter-examples if the property does not hold.

The learning curve for the tool was relatively long due to marginal and inconsistent tutorial documentation. Once the learning process was complete, the actual building the model was significantly short as tools nicely support the type of design required by the data acquisition and control systems. The created model could be verified by a visual inspection via special function depicting the relation between various model components. The code generation produces several warnings attributed to variables that were long removed from the model.

The tool can automatically generate documentation for created models listing all inputs and outputs. The simulation capability allows the design to be verified before code generation. The tool prohibited programming loops requiring the developer to use multiple nodes and several imported operations. The created design included manually written function dedicated to data reading that was re-used several times in the system. The total number lines of generated code exceeded 1.8K. The traceability was checked using the lowest model layers. Checking traceability was not an easy task since the tool automatically assigns variables names. However, there is an option to assign name to variables locally, which could take a lot of additional development time and was probably not the intention of the tool designers. Difficulty of traceability was extended due to relatively limited readability of the generated code. The code generator, after intermediate translation, creates one large source file. The format of the code is of limited readability due to lack of indentation and continuous alignment. The tool allows the developer to manually add source code (using an Imported Operator) during modeling phases. However, if the developer did not pay attention, the code generator would overwrite the working file thus destroying the code added by the developer.

3.4 Matlab/Simulink/Stateflow/Real-Time-Workshop

MATLAB and Simulink are high-performance development environments used within engineering professions that can be applied on a wide range of projects. For example, complex control systems, can be modeled either textually or graphically and the resulting data can be analyzed against a variety of metrics. Simulink allows designers to do modeling with the additional toolboxes such as xPC Target, Stateflow, and Control Systems. These toolboxes interact with Simulink, allowing a designer to combine a state machine with a control system and communicate the data across an RS232 communication port. Another ability of MATLAB is to generate C source code from a model through use of Real-Time Workshop (RTW). The generated code can run on an embedded operating system or on Windows or Unix. Stateflow provides the ability to create and model systems using the UML State Machine notation with states and transitions (Figure 5). The state machine object within the Simulink model accepts two types of inputs: data and events. There can also be internal data within the state machine.

Transitions between states can have a variety of properties. There are conditions, events on transition, and more. Further, the states can have action properties such as entry, exit, are more. These states and transitions allow the designer to model the system behavior accurately and then combined with MATLAB as stated earlier, to allow the developer to auto generate source code.

The power of these applications is staggering when compared to other engineering tools. Not only can they be applied to specific engineering disciplines, but they can also incorporate more than one engineering discipline into a model. By combining objects from different toolboxes, MATLAB and Simulink provide the ability to model, for example, the flight dynamics of a 747 and the flight control software used to control it. MATLAB and Simulink have excellent and extensive reference materials. However, they are far from simple to learn and understand and not too helpful for a novice developer resulting in a steep learning curve. A part of the problem was also that the documentation version did not match the installed tool version.

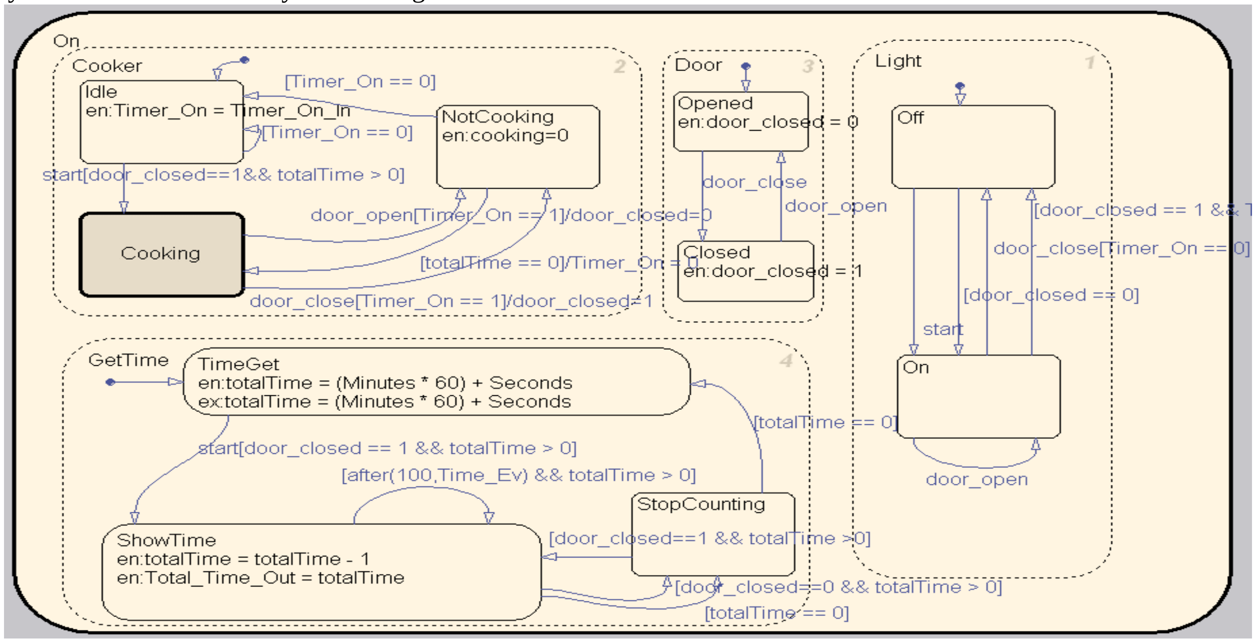

On
Cooker
2
Idle
en:Timer_On = Timer_On_In
[Timer_On == 0]
[Timer_On == 0]
NotCooking
en:cooking=0
start[door_closed==1&& totalTime > 0]
Cooking
door_open[Timer_On == 1]/door_closed=0
[totalTime == 0]/Timer_On = 0
door_close[Timer_On == 1]/door_closed=1
Door
3
Opened
en:door_closed = 0
door_close
door_open
Closed
en:door_closed = 1
Light
1
Off
[door_closed == 1 && T
door_close[Timer_On == 0]
[door_closed == 0]
start
On
door_open
GetTime
4
TimeGet
en:totalTime = (Minutes * 60) + Seconds
ex:totalTime = (Minutes * 60) + Seconds
[totalTime == 0]
start[door_closed == 1 && totalTime > 0]
[after(100,Time_Ev) && totalTime > 0]
StopCounting
ShowTime
en:totalTime = totalTime - 1
en:Total_Time_Out = totalTime
[door_closed==1 && totalTime >0]
[door_closed==0 && totalTime > 0]
[totalTime == 0]

Fig. 5: Matlab Stateflow Microwave Model

After starting Simulink toolbox browser a new model could be easily created. The inclusion of a Stateflow in a Simulink model was easy – just drag and drop. However understanding how to create Stateflow inputs was not. There are two inputs into the Stateflow: data and events. Events are passed into the Stateflow model block when they are generated within the Simulink model. The total size of the generated code was about 2 KLOC (with source files was 833 LOC of sources and 1,176 of header files). Once getting to a comfortable spot on the steep learning curve, implementing a model in MATLAB and Simulink are fairly easy. There were few issues with the simulation of the Simulink model, which would execute differently each time. For example, the ShowTime state with a self-loop set to an external timer event would transition at different times even if no variables or properties were changed. The guard was set to execute the transition after 1,000 executions of the timer event. The first time the model was simulated, this worked. The second time, instead of executing every second it executed every 10 seconds. This timing unpredictability was unresolved through the project.

Code generation within MATLAB is not very easy either. There are multitude of options available for both setting up the model and generation of the code. The documentation for this and for download to the target is not readily available for a novice user. The build resulted in generation of at least 50 or more object *.o files from the original three source files. Further, there documentation on what file to download to the target, and what task to execute to run the project on the target was difficult to find.

The scope of the behavioral aspects of the RTW code generation focus on the Stateflow diagram. The generated code does provide variables for the inputs, outputs, and local variables as defined in the Stateflow, however to access them from the target machine, a separate source file would need to be created.

Other issues experienced during this exercise included limited documentation on Stateflow transition conditions, specifically on how to include logic conditions and lack of guidance on how to include custom MATLAB code within the Stateflow object. The Simulink documentation was out of date reflecting earlier version of the product and some of the errors occurring in MATLAB and Simulink were not documented. During simulation, Simulink did not always execute the same way and occasionally run-time Simulink errors resulted in tool crashing.

What makes learning the application very hard is that there are very few simple models explaining how the toolboxes interact within Simulink. There is no explanation of why implementing x-object in such a way has a changing impact on the model; or why the model must have x-object implemented another way. The amount of the documentation is abundant often confusing unexperienced user. Lack of help for on how to use the tool was an issue. Certainly, MATLAB/Simulink examples (like an F-14 or a nuclear power plant models) detailed how inputs and outputs work through the system, but they were so complicated that even looking at them may intimidate new user. For this project, the use of MATLAB was limited to generate C source code from a Simulink model reflecting the behavior of the system specified in the requirements. After understanding how the applications worked, it was fairly easy to create the model. The obscurity of information made it difficult to implement more than the default input, output, and local data and event variables, and code generation.

4.0 Conclusions

The general observations, which were made during this experiment, were mostly common between the different tools. Lack of consistent, easy to navigate, and complete documentation was one of the major obstacles for novice users of these tools. Often, the documentation version lags the actual tool version; tutorials were inadequate and sometimes inconsistent with the behavior of the tool and they did not have enough simple examples. However, most documentation presented good reference material.

Other issues with most of the tools were: inability to clearly explain, present possible solutions, and recover gracefully from errors. At time the tool crashed when it should not have. The tools were complex and learning curves were very steep and time consuming for beginners. The code generation process was not as easy as it might seem. The generated code was large, not easy to read, and hard to debug. Verification of the integrity of the generated code is a different subject. Compilation and downloading of the code to the target platform has been a significant effort and required good knowledge of low-level computer skills such as compiling, linking, loading, *makefile* editing, location of libraries and executables, connection between host and target, etc. These skills are typically not strong side of the domain application developers. On the other hand, these are very powerful tools for software development. Each tool offers some level of power and flexibility during the process of software development. This flexibility and complexity has both good and bad side. Once the user is one level above a novice user, it is obvious that these are excellent tools for designing and implementing software. However, they are still evolving and need many improvements.

References

[1] www.ilogix.com . Rhapsody user guide, tutorial and reference document

[2] www.esterel-technologies.com . Esterel Studio user guide, tutorial and reference document

[3] www.mathworks.com . MatLab user guide, tutorial and reference document

[4] www.windriver.com . VxWorks and Tornado user guide, tutorial and reference document