MODEL-BASED VERIFICATION TOOLCHAIN FOR INCREASING TRUST ON AUTOMATED CODE-GENERATORS

By

Akshay Agrawal

Thesis

Submitted to the Faculty of the Graduate School of Vanderbilt University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE in

COMPUTER SCIENCE

December, 2013

Nashville, TN

Approved:

Dr. Sandeep Neema

Dr. Joseph Porter
ACKNOWLEDGMENTS

The research presented in this thesis was funded by DARPA Research contracts #FA8650-10-C-7075, "Multi-Modeling Language for Cyber Physical Systems" and #FA8650-10-C-7082, "META Design Flow for Cyber Physical Systems".

I am very grateful to my research advisor, Sandeep Neema, for providing me appropriate guidance during my research endeavors. He has always inspired me with his apt research aptitude and helped me hone my research and development skills by challenging the results and progress of my work at every phase. His support and motivation has helped me presenting my research work to the research community who deal with similar ideas with great confidence.

A good part of the research work was to develop tools using software development skills. During these development phases, a staff engineer at ISIS VU, Harmon Nine, helped me a lot and I appreciate him for all the time that he willingly and happily spent with me.

Last, but not the least, I thank Joseph Porter, research scientist at ISIS VU, for sharing his research experience in the domain of verification of Cyber-Physical Systems. He remained available for me whenever I got stuck in my research. He took out time always to listen to my solutions and validate them. Moreover, he took extra steps for me by pointing me to important research papers in the field of my research and sharing examples to help me understand the research issues.

This research would not have been possible without any of the people mentioned above and I will always be indebted to them.
# ACKNOWLEDGMENTS


# LIST OF TABLES


# LIST OF FIGURES


# I INTRODUCTION

1. Motivation
2. Research Problem and Proposed Solution
3. Organization of Thesis

# II RELATED WORK

# III BACKGROUND

1. Cyber-Physical Systems (CPS) and CPS research challenges
2. Model-Based Design of CPS
3. Generic Modeling Environment (GME)
4. CyPhyML: An MIC paradigm for CPS
5. Cyber components in CyPhyML
6. Temporal Logics and their Automaton equivalents
   - Linear Temporal Logic (LTL)
   - Büchi Automaton - An Automaton equivalent for LTL
6. Transition-based Generalized Büchi Automata and its C code equivalent
7. Model Checking using LTL
8. CBMC - A Symbolic C-code Bounded Model Checker
9. NuSMV model checking tool
10. SPOT Library
11. Google CTemplate

# IV DESCRIPTION OF THE VERIFICATION TOOL CHAIN

1. Conceptual Overview
2. Architectural Overview
3. VTC meta-model
4. Pattern-based property to equivalent TGBA to C code monitor translation
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>III.1 Overview of MetaGME Classes</td>
<td>23</td>
</tr>
<tr>
<td>III.2 List of Temporal Operators</td>
<td>32</td>
</tr>
<tr>
<td>IV.1 Semantics of Behavior Patterns for Temporal Property modeling in VTC</td>
<td>51</td>
</tr>
<tr>
<td>IV.2 Semantics of Scope Patterns for Temporal Property modeling in VTC</td>
<td>52</td>
</tr>
<tr>
<td>IV.3 Mappings between VTC Property Patterns and LTL formulae</td>
<td>58</td>
</tr>
<tr>
<td>V.1 IO Signals description for Ignition Logic controller</td>
<td>66</td>
</tr>
<tr>
<td>V.2 Verification results for Ignition Logic controller</td>
<td>77</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>I.1</td>
<td>Conceptual Overview</td>
</tr>
<tr>
<td>I.2</td>
<td>Proposed Verification Workflow for CyPhyML based CPS development</td>
</tr>
<tr>
<td>I.3</td>
<td>Proposed implementation of VTC toolset for CyPhyML based CPS development in GME</td>
</tr>
<tr>
<td>III.1</td>
<td>CPS generic model</td>
</tr>
<tr>
<td>III.2</td>
<td>CPS development using Model-Based Designs</td>
</tr>
<tr>
<td>III.3</td>
<td>Steps in Model-Based Design. (Obtained from [65])</td>
</tr>
<tr>
<td>III.4</td>
<td>UML-based Meta-model Example using MetaGME</td>
</tr>
<tr>
<td>III.5</td>
<td>Abstract CPS design in CyPhyML</td>
</tr>
<tr>
<td>III.6</td>
<td>Subsets of the CyPhyML meta-model</td>
</tr>
<tr>
<td>III.7</td>
<td>Heterogeneous composition of components in a Vehicle using CyPhyML</td>
</tr>
<tr>
<td>III.8</td>
<td>Tools for synthesis and code generation for CyPhyML cyber components</td>
</tr>
<tr>
<td>III.9</td>
<td>Two processes writing to a memory</td>
</tr>
<tr>
<td>III.10</td>
<td>Büchi Automaton for LTL formula - $G F p$</td>
</tr>
<tr>
<td>III.11</td>
<td>Transition-based Generalized Büchi Automaton for LTL formula - $G F p$</td>
</tr>
<tr>
<td>III.12</td>
<td>C code for Transition-based Generalized Büchi Automaton for LTL formula - $G (p \rightarrow X q)$</td>
</tr>
<tr>
<td>III.13</td>
<td>C code for Transition-based Generalized Büchi Automaton for LTL formula - $G (p \rightarrow X q)$ with Assert statements</td>
</tr>
<tr>
<td>III.14</td>
<td>Model Checking Approach</td>
</tr>
<tr>
<td>III.15</td>
<td>CBMC Loop Unwinding Approach</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>III.16</td>
<td></td>
</tr>
<tr>
<td>IV.1</td>
<td></td>
</tr>
<tr>
<td>IV.2</td>
<td></td>
</tr>
<tr>
<td>IV.3</td>
<td></td>
</tr>
<tr>
<td>IV.4</td>
<td></td>
</tr>
<tr>
<td>IV.5</td>
<td></td>
</tr>
<tr>
<td>IV.6</td>
<td></td>
</tr>
<tr>
<td>IV.7</td>
<td></td>
</tr>
<tr>
<td>V.1</td>
<td></td>
</tr>
<tr>
<td>V.2</td>
<td></td>
</tr>
<tr>
<td>V.3</td>
<td></td>
</tr>
<tr>
<td>V.4</td>
<td></td>
</tr>
<tr>
<td>V.5</td>
<td></td>
</tr>
<tr>
<td>VI.1</td>
<td></td>
</tr>
<tr>
<td>VI.2</td>
<td></td>
</tr>
<tr>
<td>VII.1</td>
<td></td>
</tr>
<tr>
<td>VII.2</td>
<td></td>
</tr>
<tr>
<td>VII.3</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

Cyber-Physical Systems (CPS) are composed of physical and software components. Embedded controllers and logic switches control a physical process using feedback loops and actuators. Controllers sense the status of physical components through sensors and then further drive the physical components according to their implementation logic. The economic and technological potential of CPS have lead to their prevalence in the modern era. Some examples of such systems are - modern medical devices, assisted living systems, automobiles, aircrafts, automated manufacturing units etc. Emergence of Model-Based Design (MBD) methodologies and tools has provided accelerated and effective ways of modeling complex CPS. Therefore, MBD has become predominant in CPS development. MBD Engineers develop graphical domain models of CPS using Domain-Specific Modeling Languages that are rich in their knowledge of a specific domain of concern. Model translators and code generators are the workhorses of MBD which facilitate automated transformations of developed models resulting in cost effective and less time consuming development life cycle of CPS. Such translators can generate target models for simulations or target program to run on actual embedded controllers in CPS.

As discussed by Lee [55], CPS like automobiles, medical devices, etc... are required to strictly adhere to their safety specifications to avoid hazards, and hence, are safety-critical. Failures of such systems can lead to physical inconvenience and monetary losses on one hand and even catastrophic disasters on the other. Thus, mitigation of bugs in systems is desired and hence, verification of developed system designs has recently been emphasized by research communities and industry both.

The verification of systems during development cycles incurs major costs and time but currently lacks formal approaches. Therefore, formal methods for verification are recently
sought by researchers. Model checking is a formal approach to verification where the behavioral state space of a system is exhaustively explored to find undesired states in the behavior. This leads to full-coverage analysis of the behavior of a system.

In MBD, the controller models can be verified using formal verification methods during the initial modeling phase, but finally the end products (for example, generated code) remain the artifacts from these models that get deployed and executed in safety-critical CPS. Application of the untrusted code-generators or translators components to generate the deployable artifacts remain the points of concern. Their use may lead to injection of errors in the translated target models or programs (for example, a faulty translator might introduce new states in the behavioral state-space of the translated models or programs), thus, invalidating their equivalence or consistency with the source models. Such errors may lead to undesired behavior in the translated artifacts and their deployment to a compromise with safety. Therefore, it remains beneficial to develop trust on the translation using automated tools and checking whether they preserve the functional behavior during the translations. The complexity of the translator tools are analogous to compliers and makes their verification a hard problem. Moreover, the generated code is not generally intended to be human-readable, but only optimized for execution and easy for automated deployment, which makes their manual verification more tedious and avoids the advantages of MBD. This thesis addresses this issue in safety-critical CPS and focuses on implementing and integrating extensions to MBD tools to provide an automated workflow to develop trust on translation by code-generators and their generated software.

I.1 Motivation

A Domain Specific Modeling Language (DSML) named "Cyber-Physical Systems Modeling Language" (CyPhyML) [53], developed using a MBD tool GME (Generic Modeling Environment), is an element of the DARPA AVM META project which is targeted at design and development of heterogeneous CPS that are particularly military ground vehicles. The
Cyber components in CyPhyML, representing digital controllers implemented as embedded software, are based upon ESMoL DSML [52], [59]. The Cyber components can be modeled in Simulink (MATLAB) and imported for use in CyPhyML. Model translators are the workhorse of MBD tools, and play a central role in synthesizing transformed models and code for deployment. Stateflow model transformers and interpreter tools are used automatically import Simulink Stateflow models to CyPhyML domain and generate C code for deployment of the controller components, respectively. The transformed ESMoL models (Cyber models) and their generated deployment code must conform to certain behavioral specifications for the purpose of safety and desired functional behaviors.

Modeling, design and development of CPS are error-prone. Unavailability of important design details during the initial phases of development, usage of multiple third-party tools during the development life-cycle and utilization of automated code generators and translators remain few of the several sources responsible for introduction of errors into the target designs and code. Errors might get introduced in behavior of the generated code for components due to faulty code generation or faulty design of a component. The need to find potential errors in designs introduced during model transformations and code generation motivates this research.

High complexity of the code-generators that are involved in MBD of large CPS renders their manual verification impractical. Current development processes lack precise integration of state-of-the-art verification tools that are capable of formally verifying large systems. As a result, validation remains confined to procedures like peer review and testing by simulation [16]. Peer review has been really efficient in recognizing bugs during development where expertise of senior engineers aid a lot. Testing refers to the verification technique where data from a system is analyzed after actually executing the system with pre-defined test inputs and then comparing with the correct anticipated outputs. But these techniques do not provide the type of exhaustive coverage of functional behavior of a system which is necessary for analyzing safety-related properties. In the former approach, the
developed system models are manually checked for syntactical and logical errors without simulating the model designs, while in the latter approach verification remains as good as the test scenarios that the designs are subjected to. Although, the above procedures can be automated to certain extent, mostly they are performed manually by humans which makes it tedious. This arises the need for automated formal verification workflows.

With the development in computing technology and on-going research on verification tools (for example, CBMC [29], NuSMV [27], SPIN [46], Zing [14]), these tools can now better handle complex systems with large state spaces. Also, these tools provide counterexamples to understand the cause for errors. The need to leverage such tools and integrate them with development tool chains, further adds to our motivation for this research.

Verification of systems and tools require certain expertise in the domain of property specification (specifically temporal logic). This also motivates a need to provide a convenient way of modeling verification properties to encourage engineers, who lack expertise with complex temporal formalisms to include verification procedures during model-based development. Graphical tools for modeling verification properties in an intuitive manner seem to be a probable solution in these scenarios.

1.2 Research Problem and Proposed Solution

Verification of software in CPS systems is a challenging problem. Model-based design tools have simplified construction of such software for system designers using higher abstractions. However, they don’t obviate the need for verification. The verification obligation now includes not only correctness of the higher level abstraction, but also the correctness of the translator that generates the executable software. Verification of complex model translators in a general sense remains an intractable problem. This thesis purports to develop a pragmatic approach which attempts to examine the correctness of such untrusted tools from the limited but tractable perspective of property preservation. A set of verification properties (using LTL) are defined which are checked using a model checking tool
(NuSMV) to assert the correctness of the (Stateflow) model with respect to the stated properties. These verification properties are auto-translated to code modules that implement equivalent observer automata. These code modules are then compiled with auto-generated code for their associated component of the system. The compiled program remains suitable for verification. To verify the correctness of behavior of the generated code with respect to the same set of properties which are initially checked with NuSMV, the compiled program is then model-checked using a source code verification tool (CBMC). The approach is packaged and delivered as a model-based toolchain for CyPhyML DSML and is named Verification Tool Chain (VTC). A conceptual overview of the proposed solution is given in Fig. I.1. Further, Fig. I.2 presents an abstract version of the proposed verification workflow for verifying the Cyber components’ behavior and developing trust in the code-generator tools that are associated with the CyPhyML DSML.

Figure I.3 presents the proposed implementation needed to pragmatically realize the workflow shown in Fig. I.2. A VTC meta-model (or DSML) allows verification properties to be modeled and linked to relevant CPS components in a TestBench for applying verification procedure. A TestBench contains a bigger component assembly representing interconnected models of components. In addition system requirements and validation tests are defined in the TestBench. These features of the TestBench allow its contained models
to be exposed to different types of analyses. In accordance to the proposed solution, a verification TestBench ultimately contains components models under concern that are linked with models of verification properties that are developed using the VTC meta-model. The process of modeling the properties is made convenient for the modeling engineers by using a pattern-based modeling approach. The VTC meta-model captures natural language-like patterns for convenience in modeling. Further, the TestBench interpreter tools transform the pattern-based property models to C code in the context of the linked components. An existing automated code-generator, which is to be verified, generates C code for the component assembly in the TestBench. CBMC is used to apply model checking procedure on the generated set of C code files. In the case of negative verification results, counterexamples traces from source C code files are generated for the users.

It should be noted that the translation of models to the NuSMV language for verification at the model-level is not being done in an automated manner in this work. But, addition of such an automated tool to VTC in future will be helpful.
Figure I.3: Proposed implementation of VTC toolset for CyPhyML based CPS development in GME
I.3 Organization of Thesis

The chapters in this thesis are organized as follows. Chapter II contains succinct descriptions of work done by others that is similar to ours. Chapter III contains an introduction to Cyber-Physical Systems and how model-based design is exploited during their development. It contains description of a model-based design tool, named Generic Modeling Environment (GME). Further, it covers the description of CyPhyML, a modeling paradigm for Cyber-Physical Systems. Preliminary concepts related to Temporal Logic in addition to description of other verification tools CBMC and NuSMV, are discussed in the same chapter as well. Chapter IV presents detailed description of the implementation during the research. A case-study illustrating application of implemented tools under the research is presented in Chapter V. Chapter VI contains conclusions and future work.
CHAPTER II

RELATED WORK

Model interpreter tools are backbone of model-based development of systems as they help in achieving rapid development by automating the development process. But faulty interpreter tools can introduce errors in the final product of development. Thus, verification of untrusted interpreters becomes a major requirement. Owing to their complexity, their manual verification is intractable and hence different approaches for their verification are necessary. A fairly good amount of research is done in this direction. [70] describes a verification technique to prove correctness of a translator. The translator presented in the paper translates source code written in MicroGypsy (a descendant of Pascal that is used as both programming and specification language) to target code in Piton (an assembly language). Two interpreters are used by them, where each interpreter can interpret the semantics of MicroGypsy and Piton, respectively. The interpreters run the MicroGypsy and Piton programs to their final states. The information from these final states are compared for their equivalence. If the information from the final states are equivalent then correctness of translator is proved, else it is refuted. Though their approach is aimed at verifying correctness of a translator, they do not make use of formal verification methods or model checking tools. Hence, no counterexamples in cases of refutation can be generated for debugging purposes, which remains one of the main benefits of model checking tools.

Though the authors of the paper above performed verification of translator’s behavior mechanically, Cimatti et al. [28] discuss automated verification of translation tools. They aim at proving semantic equivalence between generic domain-independent source computation models (composed of boolean and arithmetic-like operations) and executable target programs by checking some syntactic properties on target programs. The verifier checks the target programs against certain precomputed data to prove syntactic property specifica-
tions. The verifier is embedded – it takes an input model and an output program to perform automated syntactic verification for the proof of correctness of program generation, and thus, the program generator. Their motivation comes from the fact that the verifier is embedded into a transportation software development framework where safety is critical.

Necula and Lee [58] present research efforts in the direction of checking correctness of compilers. This paper deals with compilers which either convert programs from one source language into another target language or perform optimizations on programs in a particular programming language. Such compilers can introduce extra faulty states during optimizations or translations. Therefore, for reliability reasons their proof of correctness is desired. The paper describes the design and implementation of an optimizing compiler that translates programs from a strictly typed programming language to an assembly language. The aim of their design is to ensure the type safety in generated assembly language code to prove the correctness of optimizing computations performed by the compiler. The paper points out the importance of such proving techniques by mentioning the application of these techniques in proving certain behavioral properties of the generated programs as an addition to their former goal of compiler correctness proofs. Though like us, they are interested in verifying translators, their focus remain on verifying only syntactical properties while we are trying to show equivalence between behavioral logic of source models and synthesized target code. Karsai and Narayanan [50] present a tractable verification approach to provide certificates based on certain desired properties for assurances on the transformed models generated by automated model transformers in model-based design. They show their Goal-directed Certification technique on transformations between source StateChart [43] models and target Extended Hierarchical Automata (EHA) [57] models. They use a variant of the Bisimulation technique [16] along with Semantic Anchoring [25] to check equivalence between functional behavior of the source and the target models. Reachability analysis on EHA models for the desired safety properties is used to validate the transformation. Semantic anchoring allows them to check models of even those formalisms which
include instantaneous state transitions (i.e. multiple transitions in a single macro time step). They do mention their approach to be not providing a complete proof of correctness of the transformation, but claim their technique useful for practical verification purposes to increase trust on the transformed models. Use of the Bisimulation technique as opposed to model checking approach in our work and their focus on validating transformed models rather than generated deployable code are the key points that differentiate our work from theirs.

Property prover tools for verification can have applications other than proving correctness of a translator. An interesting idea of an application of such verification tools is presented by Zaremski and Wing [72]. They introduce a specification matching approach for different components to facilitate automated retrieval and reuse of similar components. They essentially show their specification matching approach for function modules. They compare the signatures of two function modules to derive a syntactic match between them. In addition to syntactic matching, they include semantic matching using assertion-based specifications on pre-conditions and post-conditions of the function modules under test. These specifications are matched using a specification prover tool. The main difference between their work and our proposed solution is that they use verification approach to enable retrieval or reuse of desired function modules from a library of functions rather than to verify any translator tool. The other difference lies in the fact that we use LTL to model our specifications while they use assertion based refutation of properties on preconditions and postconditions of function modules.

LTL formulae can be represented by equivalent Büchi Automata (BA), a type of ω-automata that accept infinite input sequence. These automata can be used for integration with program code to be tested. The work presented by Giannakopoulou and Havelund [40] shows implementation of a tool named Trace Analyzer (TaZ) that converts LTL formula to a BA. They use the converted BA as observer automata to analyze Java programs by integrating them with Java PathExplorer (JPaX) [44] (a tool for monitoring Java programs).
JPaX is used by them to pass the output of Java programs as input sequence to BA for enabling on-the-fly verification of Java programs. In our proposed solution in this thesis, we also aim at using Transition-based Generalized Büchi Automata (TGBA) (which is a variant of BA) as observer automata that are integrated as monitoring automata with synthesized C code of CPS controller models. Hence, their application of LTL equivalent automata is closely related to our proposed implementation.

The approach discussed in the paper above of converting LTL formula to an equivalent automaton is adapted with a variation in the work done by Staats and Heimdahl [64]. They verify the correctness of untrusted translation tools used in model based development of systems. In their verification technique, initially, verification of modeled designs of systems is performed with the NuSMV [27] model checker before synthesizing code. Then verification on synthesized code with CBMC is performed to prove the correctness of translators that are used for code synthesis. They postulate that if some critical properties can be proved to hold true, before and after translations, then it can suggest increased trust in the translation tool. They use TGBA as the equivalent automaton for an LTL formula and translate it automatically to C code monitors. To enable verification, these monitors are integrated as observing monitors with the synthesized code from system models. The monitors are augmented as user-given assertions that can suggest violation of a temporal property and can be checked for violation by reachability analysis with CBMC. Results from verification with NuSMV and CBMC are compared to prove correctness of translators. They present experimental results to prove correctness of two code generation tools that are used to synthesize C code for Simulink models. One of the tools is the Real-Time Workshop C code generator and the other is a C code generator that was being developed by Rockwell Collins Inc. Their technique is conceptually very similar to ours, but unlike them we have integrated our verification approach with a model-based design tool GME (Generic Modeling Environment [54]) by developing domain-specific model interpreter tools for enabling convenient and automated verification procedures.
Now we discuss some recent related work where people have presented application
and usage of existing model checking tools within model based development techniques
for systems. The research work of Heitmeyer [45] discusses a Software Cost Reduction
(SCR) workflow for development of Software systems. They focus on verification of large
systems using the benefits of composition of systems and reuse of existing components
through model-based engineering. They have presented a specification analysis approach
for abstract system models and they use the TAME theorem prover for model checking the
abstract models. After every phase of their engineering development cycle they validate
the translation of models based on critical specification properties which are modeled using
temporal logic. Their ultimate goal is to achieve high trust in the model constructs so that
the code synthesized from them is ‘correct by construction’. They present examples of
avionics software, submarine’s monitoring software and a software to process data from
different memory partitions to support their research methodology. Though they suggest
that high-level trust on synthesized code can be obtained by formal validation of the abstract
model designs before code synthesis, they do not elaborate on the possibility of faulty code
generators.

The recent research presented by Wang [68] focuses on analyzing correctness of be-
havioral propagation in C code generated for Simulink models. They use an open-source C
code translator for Simulink models, named GENE-AUTO (GA). In their technique, behav-
ioral analysis is facilitated by augmenting stability proof annotations in the Simulink mod-
elns. These augmentations are implemented using blocks from Simulink that are provided
in a block library. The stability proofs represent the pre-conditions and post-conditions of
Simulink blocks. These proofs are converted to a library of annotation block backends for
GA model representations of Simulink models. These annotation block backends are then
converted to ACSL (a language to specify properties for C code formally) specification an-
notations in generated C code. Formal verification tools like Farma-C, that support ACSL,
enable verification of properties (specified using ACSL) on generated C code for Simulink
blocks. This approach appears similar to our proposed solution and utilizes model-based approach for automated behavioral verification of code generated from system models. But, they do not explicitly relate their work to correctness of model translators or code generators and they do not use LTL for specifying the verification properties.

Though increasing use of state-of-the-art model checking tools is making verification of models more conveniently approachable, however, verification properties still need to be specified by engineers which requires certain expertise with concepts like LTL. Lack of such expertise makes it inconvenient for the engineers to correctly specify the properties. For these reasons, convenient specification modeling approaches are presented in several research works. The work presented by Bryant [22] discusses conversion of natural language-like function specifications to object-oriented structures for their easy integration with development processes. The predicates of the functions can be specified in a natural language-like format with domain-specific information for the variables used by functions. The definitions of functions should comply with the rules of two context grammars. The context grammars are derived using a Two-Level Grammar (TLG) and they conform to the domain for development. The Specification Development Environment (SDE) tool facilitates specifying natural language-like definitions of functions and parses the definitions to convert them to object-oriented designs. The tool requires sufficient user-interaction for improvising the definitions which are partially defined. They do not use their technique to model temporal properties exactly, but we get inspired from their work to implement easy-to-use natural language-like approach for specifying verification properties.

The work by Dwyer et al. [36], [37] presents usage of English-like patterns to define temporal properties for a system and is explicitly focused on convenient modeling of temporal properties. Usage of such patterns frees engineers from needing great expertise with temporal logic formalisms. They have categorized the patterns into behavior patterns (eg. Occurrence, Absence, Response) which define the specific behavior that a property signifies about the system. To define the moments during the execution of a system where
the behavior defined by a temporal property holds true, they provide scope patterns (eg. After, Before, Until, Always). Boundaries for a scope pattern can be defined with events that represent state of a system. They have specified mappings between combinations of English-like behavior and scope patterns for a property and LTL formulae. Such mappings can be used for automated conversion of pattern-based properties to LTL formulae. They collectively call it Specification Pattern System (SPS) and claim its usefulness in modeling numerous critical temporal properties over systems that are being used in the industry. Salamah et al. [61] extend SPS to develop Property Specification Tool (Prospec) that allows composition of SPS patterns for defining sequential and concurrent behavior of a system.

PROPEL, for "PROPerty ELucidation", is another tool presented by Smith et al. [62] which makes use of templates to model properties. They present three types of useful templates - Finite State Automata Templates (FSA), Disciplined Natural Language Templates (DNL), Decision Tree Templates (DT). FSA and DNL provide an intuitive automata-based modeling approach to decide and model behavior for a temporal property over some set of constituent variables and events of a system. FSA and DNL can be used in parallel for modeling a property as they contain analogous options for intuitive modeling of properties. DT, on the other hand, uses SPS-like patterns to define the desired behavior for a property. For better assessment of the properties that are modeled in PROPEL, semantically equivalent DNL paragraphs and timeline graph representations are presented to the end-users that help them refine their models for the properties.

A recent work by Wagner [67] presents a modeling and verification tool CertaAMOR that combines a pattern-based approach for modeling of requirements specification for systems with automated iterative verification procedures. The author discusses a microwave system example that is developed using SpecDSL and constitutes temporal properties embedded in the system model. The author further mentions the Requirements Analysis Tool (RAT), developed at Fondazione Bruno Kessler in the EU, which facilitates automated verification of specification properties. The usage of a convenient pattern-based property mod-
eling tool is closely related to the property modeling approach in our proposed solution for verification. The model-augmented English language-like requirements can be propagated to later stages in development life-cycle in an automated fashion using the CertaAMOR tool, however, the approach doesn’t include translation of abstract models to deployable code.
CHAPTER III

BACKGROUND

In this chapter, explanations for concepts of verification, meta-modeling in model based designs and tools used in the work are discussed with their strengths, limitations and major features.

III.1 Cyber-Physical Systems (CPS) and CPS research challenges

CPS are complex systems with controllers monitoring and controlling physical processes. With the changes in the states of a physical process the controllers perform computations to change states and guide the execution of the physical process. Developments in networking, monitoring, computational systems etc... are leading to more complex CPS which have begun to solve major engineering issues in present world. Some complex engineering problems where CPS has helped are described in [42] and [26] which mention a CPS for tracking a bio-medical weapon and a medical CPS in intensive care units of hospitals, respectively. Figure III.1 shows how a generic CPS model appears. In the figure it can be seen that a controller takes output from a physical component as its input, which forms a feedback loop, and via a controlling link it provides input to the physical system which increasingly and dynamically might change the behavior of the physical component.

There are many design challenges currently in the field of CPS. Due to their use in safety-critical scenarios, verification and validation of CPS is a major requirement for quality assurance which is also realized in [42] and [26]. The other research challenges in CPS are - (1) Modeling logical compositions of heterogeneous components: CPS can be composed of multiple components having different behaviors and formalisms. This makes the composition of these different components in CPS [66] models a big concern due to the incompatibility in their semantics. (2) Distributed Architecture of CPS: Benefits like concurrency and distributed computation facilities development of CPS as a distributed ar-
chitecture. Solutions to the above challenges often include developing complex reactive and networking systems which further increases risk for infusion of faults in their models that can cause serious troubles during execution of such systems. These factors further emphasize need of rigorous verification and validation of CPS.

III.2 Model-Based Design of CPS

In this section we shall discuss the current practices in model based designing of CPS and why verification is required under these practices. Owing to the complexity of CPS and emergence of a variety of computer based design tools (AutoCAD [1], Simulink [8] etc.), model-based design of CPS has become an effective practice that helps in rapid development of the CPS and makes it more productive. Meta-models are paradigms in model based designs which capture the specifications of a domain for development - in this case CPS. They capture the properties of individual types of sub-components in a system belonging to a particular domain, pertaining to the communication between types, interconnections among them, data and control flow through them etc. Further they capture the containment and connection relationships among different types of components in a domain, thus,
Figure III.2: CPS development using Model-Based Designs
providing a hierarchical structure realization of the systems that need to be developed. By embedding constraints on relationship mappings in the meta-models it can be ensured that developed domain models strictly adhere to certain relational mapping specifications and comply with a set of domain specific rules. The graphical user interfaces provided with the model-based design tools help engineers to construct a model of a particular system in hierarchical manner and visualize it in terms of the relational rules of the domain that the system belongs to. The model-based tools not only provide graphical and logical visualization of systems, but also provide additional tools that may facilitate their analysis and deployment. Additional integrated tools use the advantage of accessing meta information of the domain that developed model belongs to, which allows them to understand a particular domain so as to perform analysis and deployment tasks according to it. Figure III.2 depicts the flow of model-based development of CPS. This approach of modeling of systems is referred to as Model Integrated Computing (MIC) (for more details refer to [65] and [49]). Hence, the steps in model-based design can be summarized (as in Fig. III.3) as follows:

1. **Domain Analysis and Meta Modeling:** During this step, the domain under consideration is analyzed so as to construct the meta-models that capture the rules and specifications of the domain to which the target domain models should adhere to. This is similar to requirements analysis in software engineering with a subtle difference that requirements here are set of specifications for a particular domain of interest.

2. **Target domain model development:** Using meta-models, target domain models of particular systems are developed which depict the physical or computational behavior of components within a system in addition to connections and relationships among them. The goal here is always to ensure the production of models that represent the mappings of components of a system analogous to the real system.

3. **Development of Model Interpreters:** During this step, additional tools are developed
which refer to the meta models developed in the first step and take the models generated in the second step as inputs to perform certain tasks on them. The tasks can include certain types of analysis, augmentation of specifications, transformations of models for simulations, verification, deployment etc.

Lee et al. [47] discuss a CPS - the Tunneling Ball Device [48] as a case study and provide detailed steps of model-based design of the device by realizing it as a centralized system (as opposed to distributed system) composed of continuous system components depicting kinematics of a free ball, rotating disc and a DC motor. They use PTIDES [34] as
the model of computation for the Tunelling Ball Device, which is derived by specifying
timing constraints on sensor networks and thus extending discrete-event DE [39] seman-
tics. They use simulation tools - Ptolemy [38] and LabView [4]. Ptolemy is a framework
for modeling heterogeneous CPS using a variety of Models of Computation (MoCs) where
the input and output behaviors of the components are defined by specifications of an MoC.
Brooks et al. [21] study advantages and pitfalls of using different MoCs in model-based
design using a Traffic Light example. Balasubramanian et al. [17] emphasize on the advan-
tages of mode-based design over ad-hoc methods for system integration and composition
using reusable component designs by discussing the application of model-based design to
develop Embedded Automotive Applications.

III.3 Generic Modeling Environment (GME)

GME [54] is a modeling tool to develop DSMLs (Domain Specific Modeling Languages)
and their respective target domain system models. The development method for system
modeling in GME follows the MIC approach. Post-requirements analysis of a domain,
UML (Unified Modeling Language) [10] - based meta-models are developed in GME that
represent a DSML for the domain. The graphical interface and UML-based approach to
construct meta-models make later revision phases of a DSML convenient. GME provides
a design language - MetaGME, which is a set of classes that are used to develop meta-
models for DSMLs. Figure III.4 and Table III.1 show a simple example of a meta-model
developed using MetaGME and semantics of the MetaGME classes, respectively. The de-
developed meta-models are used to then develop system models within the context of the
specified DSML. GME allows for checking syntactical correctness of models, by checking
them against constraints modeled in OCL (Object Constraint Language [6]). Often during
the final stages of development life cycle analysis and deployment facilities are necessary.
For these purposes additional toolsets including code generators and model transformers
are developed within the context of the specified DSML’s meta-model. Association of
tools with a particular DSML of interest makes them more intelligent for use within the context of the DSML. Development of such extension tools for GME requires generation of a domain-specific C++ or CSharp API for the class types that are defined in the DSML’s meta-model. GME facilitates automated generation of these APIs, which contain Object-oriented classes [11], using UDM (Unified Data Modeling Framework [56]). These APIs allow engineers to develop the extension tools that can operate on input models developed using a DSML meta-model, and traverse the input models’ hierarchy to further transform the input models for further analysis, simulation and code generation.
III.4  CyPhyML: An MIC paradigm for CPS

The Cyber-Physical Systems Modeling Language (CyPhyML) [53] is a DSML for modeling heterogeneous CPS. It is a unified modeling framework to model behaviors of heterogeneous components of a CPS while allowing logical interconnections among them to define designs. Owing to different behaviors of components in a heterogeneous CPS, it becomes hard to define semantics for the interconnections between different components. It is developed under the META project of the DARPA AVM research effort to aid in development of next generation autonomous military vehicles.

To realize complex heterogeneous CPS, their physical and controller component models need to be composed together. The physical components (continuous), which may be acausal [69], can be modeled as Modelica [5]-based models; the controllers, which are causal, can be modeled as Stateflows (discrete) using Simulink [8]. Acausal connections between components are consistent with the idea that the composition of physical systems implies a simultaneous and instantaneous sharing of the physical states between composed
components. Causal connections are consistent with the composition of digital systems - each causal signal interface is either an ‘input’ or ‘output’, and the interconnections imply dependency between components as well as a separation in their time of occurrence of events in each component. The state information of physical components is fed into controllers to drive their discrete logic which in turn control the continuous dynamic behavior of the physical components. But, composition of continuous and discrete components poses issues related to semantics of the composition as the components may share variables across boundaries. The composed systems are referred as Hybrid System [15] having both discrete and continuous states. CyPhyML facilitates meaningful composition of multi-domain CPS components in designs and the synthesis of simulation models to evaluate those designs. CyPhyML components are specified as wrappers encapsulating more specific model behavior. For example, the dynamic behavior of a vehicle component may be specified as a Modelica model. To use that model compositionally in a design, CyPhyML requires a reference to the detailed Modelica model and allows the modeler to specify an interface with strongly-typed ports that define how the component can be composed with compatible components to form a design. Careful attention to the semantics of those interfaces and connections allows the final design to be well-formed and ultimately represent a valid simulation. CyPhyML provides variety of interface ports for the wrapper components. Signal ports, power ports and mechanical ports are few among those types. These interface ports are internally connected to the Stateflow or Modelica model contained inside the wrapper component. To compose compatible components the interface ports are externally connected. These interconnection types must be specified in the CyPhyML meta-model. Certain complex context-specific rules for valid interconnections are also modeled as Object Constraint Language (OCL)-based constraints. Interconnection between signal ports are responsible for propagating data and control signals between physical and controller components. For mechanical and power-based interconnections mechanical and power ports can be used. The composed models can be simulated by associating them with Test
Bench [71] models in CyPhyML. The domain-specific model interpreter tools are applied on the TestBench models to generate code or transformed models for simulation and analysis. Figure III.5 shows a controller and a physical system modeled which are imported into the CyPhyML modeling environment. Both the models in the figure are shown to be encapsulated by Component wrapper blocks and are interconnected through different interface ports of the wrapper blocks. The controller sends and receives data and control signals to the physical system via signal ports labeled as 'S' on blocks 'A' and 'B', while the physical system is getting its power from block 'D' via power interface port of block 'B' and is maintaining a mechanical connection with block 'C' via mechanical interface port of block 'B'.

The CyPhyML meta-model is designed using GME. Figure III.6.1 shows that a 'ComponentType' object can contain a 'SignalFlowModel' and/or 'ModelicaModel' objects. In CyPhyML - a 'SignalPort' object can be an interface port contained inside a 'ComponentType' object, a 'IO_Signal' object can be contained as an interface port inside a 'SignalFlowModel' object, a 'ModelicaSignalPort' object can be contained as an interface port inside a 'ModelicaModel' object. Figure III.6.2 shows how 'SignalPort' type ports can be internally connected to 'IO_Signal' and 'ModelicaSignalPort' type ports via 'SignalPortMap' type connections to enable meaningful composition of heterogeneous 'ModelicaModel' and 'SignalFlowModel' type components. While composing heterogeneous components in CyPhyML, 'SignalPort' type ports of different Component wrapper blocks can be connected via 'InformationFlow' type connections, as can be deduced from Fig. III.6.3 and Fig. III.6.4. Figure III.7 shows composition of heterogeneous components in a vehicle using CyPhyML - a transmission assembly (named 'TransmissionExtendedGenericV2'), containing a Modelica model with continuous behavior and is controlled by the transmission controller (named 'TransmissionControllerV2'), containing a StateFlow model that follows discrete semantics. The controller determines the current gear state for the transmission assembly based upon its RPM (Rotation Per Minute) values. All sig-
Figure III.6: Subsets of the CyPhyML meta-model
nals for transmission gears from 'TransmissionExtendedGenericV2' to 'TransmissionCon-
trollerv2' are modeled as 'InformationFlow' type connections, thus, enabling heteroge-
neous composition of both components that have 'SignalPort' type (strongly typed) inter-
face ports. Figure III.7 also shows how the 'SignalPort' type ports ('Gear1' to 'Gear6') of
'TransmissionControllev2' (wrapper component), are internally connected to 'IO_Signal'
type ports of 'ControllerContainer' (a 'SignalFlowModel' following Stateflow semantics) to
enable heterogeneous composition.

III.5 Cyber components in CyPhyML

To model the behavior of the cyber components, ESMoL (Embedded Systems Modeling
Language) [52], [59] paradigm is adapted in CyPhyML. ESMoL is a DSML to enable mod-
eling of embedded systems and generating simulations for them. Its major purpose lies
in adding distributed deployment capability to Simulink models for platform-specific dis-
tributed architectures and enabling their scheduling analysis. 'MDL2MGA' is an ESMoL-
specific interpreter tool that converts Simulink models to CyPhyML-specific cyber-domain
models which. 'SignalFlowModel' in CyPhyML (refer to Fig. III.6 and Fig. III.7) are
the encapsulating wrapper interfaces for the converted models. Strongly typed interface
ports of 'SignalFlowModel' enable composition of the converted Simulink models with
other physical components in CyPhyML. For simulation purposes, a C code API is gener-
ated for the cyber components in CyPhyML using 'CyPhy2SLC_CodeGen' interpreter tool.
CyPhy2SLC_CodeGen is an upgraded and CyPhyML-compatible version of the interpreter
tool developed under the research work for ESMoL that was used to generate the C code
API for Simulink/Stateflow models in ESMoL. The steps for synthesis and code genera-
tion for CyPhyML cyber components are given in Fig. III.8. The two interpreter tools
mentioned in this section are used for performing experiments that are presented in this
thesis.
Figure III.7: Heterogeneous composition of components in a Vehicle using CyPhyML
Figure III.8: Tools for synthesis and code generation for CyPhyML cyber components
III.6 Temporal Logics and their Automaton equivalents

Requirement for verification of safety-critical CPS is paramount. For the purpose of verification of CPS, some sort of formalism to specify properties of a system is required. Numerous state-of-the-art verification tools use temporal logic to specify a system’s properties. Specific properties of a system over certain state-variables can be specified over time using temporal logic. Temporal Logic [16], [31], [18] can be of two types - Linear Temporal Logic (LTL) or Computational Tree Logic (CTL). In LTL, the time is assumed to be linearly increasing during the execution of a system leading to a linear path for time-based traces, while in CTL, the time can have branching paths leading to a tree structure of time-based traces. Though the nature of time for a system can be associated with real time, within the context of verification of systems the notion of time is generally abstract. Hence, each time step in abstract-time within the context of a specification can be related to, suppose, each sampling cycle of a sensor network, or each computation cycle of a toplevel system in a CPS etc. In the work presented in this thesis, the LTL formalism is used, and hence discussed in detail here rather than CTL. In the first subsection LTL is discussed, in the second subsection equivalent automata for LTL (Büchi Automata) is discussed and in the third subsection C code representations of LTL formulae are discussed.

III.6.1 Linear Temporal Logic (LTL)

LTL is used to specify Linear Time properties of a system. It extends propositional logic with temporal modalities [16] – that is, atomic propositions of predicate logics can be associated with temporal operators to convert predicate logic to LTL. The elementary temporal operators are given in Table III.2.

Examples: Suppose $a_1$ and $a_2$ are two atomic propositions over boolean variables $p$ and $q$, respectively, such that – $a_1: (p == \text{true}), a_2: (q == \text{true})$. Following are examples of a few properties written as LTL statements:

1. 'now and forever in the future' $a_2$ must occur 'in the next time step' after every
Operator Notation | Meaning
---|---
G (global) | now and forever in the future
F (future) | eventually in the future
X (next) | in the next time step in the future
U (strong until) | until some stop condition in the future
W(weak until) | forever or until some stop condition in the future

Table III.2: List of Temporal Operators

occurrence of $a_1$. Using temporal operators the LTL mathematical statement can be written as: "$G ( a_1 \rightarrow X a_2 )$", where ‘$\rightarrow$’ means 'implies'.

2. *if $a_1$ occurs in the very first time step, then in the second time step $a_2$ must occur.*
   The equivalent LTL statement will be: "$a_1 \rightarrow X a_2$".

3. *'In the future' if $a_1$ occurs, then in the next time step $a_2$ must occur.* The equivalent LTL statement will be: "$F (a_1 \rightarrow X a_2)$".

4. *$a_1$ occurs 'until' $a_2$. The equivalent LTL statement will be: "$(a_1 \ U a_2)$".*

5. *Occurrence of $a_1$ leads to occurrence of $a_2 'in the future'*. The equivalent LTL statement will be: "$(a_1 \rightarrow F a_2)$".

For the purpose of verification of systems the atomic propositions can be formulated over state variables in a system and specification properties can be written in temporal logic by associating temporal operators with those atomic propositions. An example of two processes $P1$ and $P2$ is given in Fig. III.9. Here both the processes are trying to obtain mutually exclusive write access to the memory using semaphores. The semaphore variables for both the processes are ‘writeLock’. For such a scenario there can be specified at least two temporal properties to ensure sound behavior of the whole system: (1) Both processes should not get write access to the memory at the same time. (2) After $P1$ gains access to write to the memory, $P2$ should also eventually gain write access, and vice-versa.
For Fig. III.9, let us assume two atomic propositions $A1$ and $A2$ such that - $A1$: $P1.writeLock == true$, $A2$: $P2.writeLock == true$. The above stated temporal properties can be written as follows after applying temporal operators on the conjunction of boolean atomic propositions $A1$ and $A2$:

1. $P1.writeLock$ and $P2.writeLock$ should never be true simultaneously:

   "$G ( !(A1 & A2 ) )$", where '!' and '&' are boolean operators for 'NOT' and 'AND' respectively.

2. $P1.writeLock$ becoming true should eventually lead to $P2.writeLock$ becoming true in the future:

   "$A1 \rightarrow F A2$".

There can be two types of temporal properties - safety and liveness [51], [16]. Safety properties depict that something bad never happens, while Liveness properties mean that something good eventually happens. In the above example of two processes writing to a memory, the first property stating that both processes should never simultaneously gain write access to memory is a good example of a safety property. Verification of this safety property for the system ensures that a major safety-critical behavior is ensured within the
system. The second property in the same example stating that one process eventually gains write access after the other is a good example of a liveness property. Though the second property doesn’t relate to safe behavior of the system, but it ensures that both the processes eventually are able to write to the memory.

III.6.2 Büchi Automaton - An Automaton equivalent for LTL

Once properties are available as LTL statements, they must be converted to C-code statements for C-code verification tools. LTL statements are not suitable for such conversions directly because of the denotational nature of LTL. Hence, using intermediate operational forms such as automata remain a good choice to facilitate generation of equivalent C-code statements for LTL statements. Automata traversal logic in interpreter tools may then easily automate the generation of LTL-equivalent C-code statements for verification purposes. Automaton equivalents of LTL properties for models that do not represent finite systems require the automaton to accept infinite input sequences.

A Büchi Automaton (BA) [60] is an infinite input sequence accepting automaton which can be used to represent LTL properties. Schnieder and Alpern [13] explain how LTL properties can be represented as a deterministic Büchi Automaton (DBA) and give examples of DBA representations of temporal specifications for a mutual exclusion protocol. They further explain that the states of a BA can present predicates in a temporal logic and hence an acceptable input sequence of the BA proves satisfiability of the temporal logic that it represents. A definition for BA is given in Definition 1.

**Definition 1**: A Büchi Automaton is a defined as a six-tuple automaton:

\[ A = (S, S_0, \delta, F, D, L) \]

where,

- \( S \) is finite set of states in the automaton,
- \( S_0 \subseteq S \) is a set of initial states,
- \( \delta \) is a set of transition relations depicting \( S \rightarrow 2^S \),
- \( F \subseteq S \) is a set of acceptance states,
D is a finite domain for inputs,
\[ L : S \rightarrow 2^D \] is labeling function.

In relation to Definition 1 an acceptable input sequence exists when for an infinite input sequence over domain D an acceptance state from F is reached infinitely often. The concept of acceptance states can be understood with an example as in Fig. III.10 which is a Büchi automaton representation of the LTL property \((G F p)\) depicting - 'p' occurs infinitely often. It is a DBA with initial state as '1' and an acceptance state as '2'. It can be intuitively understood that if '2' occurs infinitely often for some input sequence in the DBA then it satisfies the claimed property of \(p\) occurring infinitely often.

Lerda and Giannakopoulou [63] explain an approach to minimize the Büchi Automaton derived from an LTL formula. For verification purposes, the transition system representation of a system under test is combined with a BA-equivalent of the LTL property via Cartesian product. The equivalent BA representations of complex LTL formulae can have an exponential number of states, thus, increasing the state-space complexity of the final product automaton to be verified. Thus, minimization of a Büchi Automaton leading to fewer states in the final product automaton helps in verification owing to its reduced state-space. The authors claim to generate smaller product automata when minimization of BA is performed by rewriting LTL formulae according to a set of expansion rules (called tableau rules) and then applying boolean optimizations over the states generated for BA.
III.6.3 Transition-based Generalized Büchi Automata and its C code equivalent

In this we use another variant of BA, called Transition-based Generalized Büchi Automaton (TGBA) [41], where acceptance conditions are associated with transitions rather than states. They can be generated with the existing tableau-based methods and potentially generate much smaller automata than acceptance-state-based BA as suggested in [41]. Figure III.11 shows the TGBA equivalent of the BA in Fig. III.10. In the figure, the transitions with ‘{Acc}’ belong to accepting set of transitions. An infinite input sequence for an LTL formula proves to be satisfiable if infinitely often those transitions are traversed in the corresponding TGBA which belong to the accepting set of transitions. In Fig. III.11 if the accepting transition from state ‘1’ to state ‘2’ and the accepting self-transition of state ‘2’ are traversed infinitely for an infinite input sequence, then it means that ‘p’ occurs infinitely often, hence, satisfying the LTL formula (G F p).

We are interested in automated verification of systems, and hence generate code of verifiable programmatic wrappers from LTL-formulae-equivalent-TGBA in an automated manner. To discuss an example of code generation from a TGBA let us consider an LTL formula (G (p → X q)) denoting – always for every occurrence of ‘p’ in the next time step must occur ‘q’, where ‘p’ and ‘q’ are atomic propositions. Figure III.12 shows the TGBA and the C code representations of the considered LTL formula. All transitions in the shown TGBA are acceptance transitions, i.e. satisfiability of the LTL formula proves true if any of the transitions is traversed for an input sequence. The C code presented is a set of ‘if else’
Figure III.12: C code for Transition-based Generalized Büchi Automaton for LTL formula \((-G(p \rightarrow X q))\)

Statements with conditions encoding the current state status and proposition conditions for which transition conditions become true. States '1' and '2' are labeled as 'G(Xq | !p)' and 'q & G(Xq | !p)', respectively.

For verification purposes the generated C-code is utilized by code verification tools. For an input sequence to the TGBA, a violation of LTL formula is concluded if none of the accepting transitions in Fig. III.12 are traversed for the input sequence. 'Assert' statements are included to the set of 'if else' statements as given in Fig. III.13 to indicate violations of the property. If a verification tool reaches the 'assert' statement during its applied verification procedure then a violation is concluded. Features of TGBA, like smaller compact final automata and convenient code representations and generation, motivates their usage in this work.
III.7 Model Checking using LTL

For verification of safety-critical CPS, formal methods are currently recommended by researchers. Model Checking is a formal method of verification where exhaustive state-space explorations is performed automatically on the source CPS models to check whether they satisfy certain verification properties. To perform model checking, a transition system for the system under test is modeled and is multiplied with an automaton equivalent to the negated LTL formula of a property (property representing 'bad behavior'). The derived product automaton is subjected to model checking tools like CBMC, NuSMV which apply decision procedures to check if in the product automaton a bad state can be reached. In cases of unsatisfiability (when a bad state is reachable), a counterexample trace in the source model is provided for debugging purposes. The model checking approach is shown in Fig. III.14.

III.8 CBMC - A Symbolic C-code Bounded Model Checker

As explained in Section III.7, model checking for verification of systems requires generation of the product of a transition system representing the behavior of system under test and an automaton equivalent of LTL property that is needed to be checked. The generated product automaton is provided as an input to model checking tools to perform their verification. The state-of-the-art model checkers represent the state-spaces of provided input automata.
using *Binary Decision Diagrams (BDD)* [23] or *Boolean Encodings (SAT)* [20], [19], [24] to enable application of verification procedures on them. BDD uses explicit representations of states in a state-space. The explicit state-space representations make use of canonical forms which make the representations bulky. Additionally, requirements for computation of variable ordering in BDDs exacerbates the problem of state-space explosion. For this reason, model checking techniques using BDDs can not handle very large state-spaces (that include around $10^{20}$ states). As a result verification problems for BDD-represented state-spaces become intractable. To counter this, symbolic model checking tools make use of propositional formulae to represent states in the state-spaces using boolean encodings. This enables them to handle very large state-spaces too [24].

CBMC [29], [30], [9] is a symbolic model checker that can be used to verify ANSI-C programs. It facilitates checks on pointers, bounds on arrays, type safety in addition to user specified assertions. It converts a specification property and the behavioral state-space of a
while loop structure in a given C-program

```c
while ( condition )
{
    // computational body;
}
```

**With a bound of ‘3’ CBMC unwinds the above loop as follows:**

```c
if ( condition ) {
    // computational body;
    if ( condition ) {
        // computational body;
        if ( condition ) {
            // computational body;
        }
    }
}
```

Figure III.15: CBMC Loop Unwinding Approach

C-program, that is an implementation of a system in C language, into set of propositional formulae, that are in *Conjunctive Normal Form (CNF)*, to apply satisfiability checks on them. CBMC converts an input C-program into a *goto binary* file with extension ‘.exe’. The generated goto-binary is not executable, but essentially a control flow graph of the C-program comprising of *if, else and goto functions*. As suggested by CBMC’s name, it performs *bounded* model checking on a given C-program. Bounded model checking with bound of ‘k’ means that symbolic model checking is performed till an execution depth of ‘k’. CBMC achieves this by the approach of ‘loop unwinding’. For an example of loop unwinding, consider Fig. III.15 where a *while* loop is converted into a set of three *if else* statements for a bound of ‘k=3’ in CBMC. With a given bound ‘k’ CBMC unwinds the loop structures till the specified bound to generate the corresponding set of propositional formulae for the given C-program.

Suppose the set of propositional CNF formulae for a C-program to be tested and a
property are signified by ‘S’ and ‘P’, respectively. CBMC generates a conjunction of ‘S’ and negation of ‘P’ \((S \land \neg P)\) and checks for its satisfiability by passing it to an efficient SAT solver. If the generated conjunction formula proves to hold true then a violation of property occurs and a counterexample is provided by CBMC in that case. It should be noted that bounded model checking can only suggest satisfiability of a property up to the specified bound rather than the complete behavioral state-space of a model.

CBMC provides some set of functions [9] that are of relevance to the presented work in this thesis. They are as follow:

1. \texttt{CProverAssert(condition, string)} is a CBMC assert function which generates an assertion when the supplied condition evaluates to be false. Example: \texttt{CProverAssert(a == 3, ”Property violated!”);} generates an assertion if variable ‘a’ is not equal to 3 and prints the passed string in the counterexample output.

2. \texttt{CProverAssume(condition)} takes a condition over a variable in the program and prunes the further exploration of state-space whenever during an exploration path the condition evaluates to be false. Example: \texttt{CProverAssume(a \geq 2 && a \leq 5);} prunes further state-space exploration along a path whenever variable ’a’ has a value not between 2 and 5 (both 2 and 5 are inclusive).

3. \texttt{nondet_int()}, \texttt{nondet_double()}, \texttt{nondet_char()} are functions that generate random values with respect to the type of function used (return type is \texttt{int} for function \texttt{nondet_int}). They can be used to achieve better coverage on analysis by randomly assigning test values to intended input or dummy variables. Example: \texttt{a = nondet_int();} assigns a random value of type ’int’ to variable ’a’.

Adhering to object-oriented approach may lead to development of several .c and .h files in any project. CBMC provides facility to create goto-binaries for multiple C files and link them together into a single .exe goto binary by providing its own compiler and linker.

The features of CBMC to model check large C programs against user specified assertions and generation of random data values for better coverage analysis motivate its usage in the work presented in this thesis.

III.9 NuSMV model checking tool

NuSMV [27] is an open-source model checker written in ANSI-C. It is able to perform model checking on system models written in the NuSMV language. A system’s architecture may be modeled in a modular fashion in the NuSMV language. The interface definitions of the written modules facilitate a fine communication mechanism between them. The parser in NuSMV reads files defining system models. The compiler in NuSMV then converts the parsed model into BDD [23] representations containing boolean formulae translated from the model descriptions. This enables NuSMV to perform efficient model checking by constructing and manipulating Finite State Machines (FSMs) at BDD level. NuSMV features an interactive textual shell and a graphical user interface. NuSMV facilitates the manipulation of FSMS at BDD level by providing options for choosing values of variables interactively at every step during symbolic simulation of models. Alternatively, it may assign random values to the variables by itself. NuSMV checks semantics, for example, circular dependencies between different modules, after parsing the models from files. LTL-based model checking is performed with NuSMV based upon the properties which are specified as LTL formulae over the variables representing different states of a system. If the model checking procedure concludes any property violations of the properties, NuSMV provides counterexample traces to aid in debugging. NuSMV uses state-of-the-art tableau construction methods for LTL model checking that are described in Section III.6.3. NuSMV provides the Bounded Model Checking method to check for violation of a property up to a certain bound. These features motivate NuSMV’s use in this research work.
III.10 SPOT Library

Spot Produces Our Traces (SPOT) [35] library is an object-oriented C++ library providing functions that facilitate verification tasks. Due to the useful features of TGBA, as discussed in Section III.6.3, SPOT developers chose to rely upon it. The rich set of model checking functions provided by the SPOT library makes it a good candidate for integration into verification tools. The Python bindings provided for SPOT also make it a favorable candidate when a tool’s development framework comprises of Python code. But, Python bindings were of no concern during our implementation work as the tools presented in this thesis were developed using C++. The emptiness checking and LTL translation algorithms from Couvreur [32], [33] are implemented in SPOT. In this thesis, our interest lies only in converting LTL formulae to TGBA and hence we are concerned only with LTL translation algorithms provided in SPOT, and not emptiness checking algorithm. The two LTL translation algorithms in [35] are termed as ’SPOT/FM’ [35] and ’SPOT/LaCIM’; the former generates compact equivalent automata, while the latter is not intended to generate compact automata. This motivates the use of ’SPOT/FM’ in implementation presented in this thesis.

SPOT provides a minimalist interface to an abstract class that contains TGBA structure. Some appealing features of this interface are - (1) easy extraction of initial state of a SPOT-generated TGBA by the function ’get_init_state()’. (2) iterator based traversal technique to read all states and transitions of a TGBA by function ’succ_iter()’. (3) understandable sequence of error log statements printed on a tools console in cases when TGBA generation fails. These features further motivate our use of the SPOT library in this work. The abstract diagram in Fig. III.16 depicts the usage of SPOT for verification purposes in this work.

III.11 Google CTemplate

Google CTemplate [3] is a template-based approach to separate computation of data from its presentation while output is being generated from any tool. As opposed to implementing application logic in a tool to compute data with correct presentation, separating relevant
variable data computations belonging to repetitive output code helps in minimizing application logic that needs to be implemented. Further, it allows us to avoid any change in application logic due to slight modifications in the presentations of desired output. This feature is facilitated by Google CTemplate programming by using templates that contain the generic presentation structures of desired output and annotating them with *dictionary markers* to later fill in the computed data wherever necessary. Hence, an application is required to contain only the implementation logic to compute variable data for filling in templates. An example of a simple template is given below:

```cpp
1 {{#ENTRY}} {{EMPLOYEE_NAME}} has a salary of {{SALARY_AMOUNT}} U.S. Dollars.
2 {{/ENTRY}}
```

In the template given above, a *section dictionary* with title 'ENTRY’ contains two tag names - 'EMPLOYEE_NAME’ and 'SALARY_AMOUNT’. Given below is a C++ program that can be associated with the above template to compute data:

```cpp
1 #include <stdlib.h>
```
According to the program given above, every time a section dictionary is added to the `TemplateDictionary` object 'dict' by using function 'AddSectionDictionary()', a new entry for an employee is printed in the output from the template.

CTemplate has appealing use-cases in development of extension toolset for model-based design tools. An intuitive example could be the following scenario - suppose a tool automatically generates a program which contains function calls for several components in a system. Now, while calling a function in a programming language it is imperative to pass correct input variables, which in this case maybe the name of interface ports of a component. In such a scenario, parameters for the function calls of multiple components having different names for their interface ports can be computed by implementing component traversal logic in a tool and using a relevant presentation in the template such as:

```
"{{#CALL}} componentMain( {{PORTNAME}} ) ; {{/CALL}}".
```
The above explained features of the CTemplate system motivate its use in the presented work.
As per the proposed solution in Chapter I, for integration of the verification workflow shown in Fig. I.2 with the CyPhyML framework, interpreter tools in context of CyPhyML and the VTC meta-model are implemented as extensions to GME to realize the proposed implementation shown in Fig. I.3. These tools are packaged as a toolchain - the 

**Verification Tool Chain (VTC)**. In this chapter, a conceptual overview is provided along with the architecture of VTC and description of the features and usage of its interpreter tools. The next chapter shall contain an illustrative example to show the process of verification for CyPhyML cyber-domain models using the implemented tools.

**IV.1 Conceptual Overview**

A conceptual overview of VTC is given in Fig. I.1. The cyber components in CyPhyML (for which VTC is proposed) are Simulink/Stateflow models representing discrete-time controllers that are translated to CyPhyML models (refer to Fig. III.8). Temporal verification properties can be modeled for these cyber components in GME using VTC, which must be linked to the cyber components under test. The whole verification setup is developed in a CyPhyML TestBench. The verification properties can be modeled in three ways - (1) using a set of English-like patterns for convenience, (2) directly writing LTL formulas for a temporal property, 3) modeling a temporal property using an $\omega$-automaton (refer to TGBA in Section III.6.3). To support our research aim of proving correctness of translators, model checking should be performed directly at the model level on cyber component models. To achieve this, the NuSMV tool is used. It should be noted that no tools were developed for automated verification with NuSMV and hence for conducting experiments models were translated manually to NuSMV representations for their verification. To prove correctness of translator tools, their generated code output needs to be verified against the same set of
properties which are used during NuSMV verification. To achieve this, CBMC tool is used for verification, and for providing input to CBMC, C code needs to be generated for all the cyber components and the modeled verification properties that are contained in a developed TestBench. To facilitate verification of generated C code, its structure is ensured to be such that the code fragments corresponding to a modeled property act as observing monitors for the output signals of the cyber components under test. CBMC can then symbolically simulate the behavior of cyber components and the property monitors to check the status of the monitors so as to prove or disprove the violation of properties. The later sections of this chapter contain the descriptions of procedures for property modeling, C code generation for verification, and the tools used to achieve the goal of verification.

IV.2 Architectural Overview

An architectural overview of VTC is given in Fig. IV.1. The two interpreter tools developed under VTC are – Verification Property Converter (VPC) and CyPhyTB2Ccode.Gen. The two tools remain available to be invoked from within a TestBench. Any of the three types of property models (Pattern-based, LTL statement or Automata) can be associated with the component under test in the TestBench. The RangeGuarantee elements define a range on the input signals of the component under test allowing the verification tools to symbolically execute the behavior code of the component within that range. VPC generates a TGBA property model from a pattern-based temporal property model or an LTL-based property model. The CyPhyTB2Ccode.Gen tool traverses the assembly of components under test connected to the property model that it is invoked from, according to the causality order of the components in the assembly. It then generates a verification-enabled C code file containing execution call statements for the components and the observing monitor. This file can then be readily fed to CBMC for verification at the code-level. The following sections describe the features and usage of the tools in further depth.
IV.3 VTC meta-model

This section describes a meta-model for VTC that is developed in GME. This meta-model facilitates modeling of temporal properties using a pattern-based approach, LTL formula writing and an automaton-based approach. The idea behind the pattern-based approach is to make property modeling for systems convenient for domain engineers who lack expertise in temporal logic. Further, this model-based approach to property modeling allows for easy association of verification properties with the system models to be tested from the beginning of the development cycle. The meta-model for VTC is shown in Fig. IV.2. VTC meta-model facilitates modeling three types of property blocks -

1. PatternBased Requirement - This type of modeling block uses intuitive English-like patterns for modeling temporal logic. An example is shown in Fig. IV.3. This type of property block has two menu-type attributes: PatternType and ScopeType. PatternType specifies behavior patterns while ScopeType specifies scope patterns. The
behavior patterns are used to describe temporal semantics over atomic propositions. The current set of behavior patterns are given in Table IV.1. Currently, only two atomic propositions are allowed to be defined in the property block as objects of the Proposition meta class. A Proposition can be contained only in this type of property block and it can have either of the two titles ‘P’ or ‘S’. The titles are used to associate a Proposition object with the behavior pattern used in its parent pattern-based property block. The condition attribute for a Proposition needs to have a conjunction or disjunction of boolean predicates which are defined over the names of VTC Signal ports. Of course, these variables must refer to the state of the component under test. For example, to use behavior ‘Immediate Response(P & S)’ two Proposition blocks must be inserted within the property block, one with title ‘P’ and other with title ‘S’. An example of a condition attribute for a Proposition can be: \( a = 5 \land b = 6 \lor c \leq 8 \), where a, b, c are names for VTC Signal ports representing the names of signals to be tested.
<table>
<thead>
<tr>
<th>Behavior Pattern</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existence(P)</td>
<td>'P' holds true</td>
</tr>
<tr>
<td>Absence(P)</td>
<td>'P' does not hold true</td>
</tr>
<tr>
<td>Immediate Response(P &amp; S)</td>
<td>if 'P' occurs at some time-step then 'S' occurs in the next time-step after 'P'</td>
</tr>
<tr>
<td>Response(P &amp; S)</td>
<td>if 'P' occurs at some time-step then 'S' occurs in the future after 'P'</td>
</tr>
<tr>
<td>Precedence(P &amp; S)</td>
<td>'S' must have already occurred before 'P' occurs at some time-step</td>
</tr>
</tbody>
</table>

Table IV.1: Semantics of Behavior Patterns for Temporal Property modeling in VTC

The scope patterns define the scope for which a chosen property behavior is valid.

The scopes for such a pattern are defined by inserting Event_Trigger_Condition type blocks that represent atomic propositions with title being either 'R' or 'Q'. The title attribute is used to associate an Event_Trigger_Condition with the scope pattern defined for its parent property block. The condition attribute for Event_Trigger_Condition is similar to that of Proposition. The semantics of different scope patterns are given in Table IV.2.

2. LTLSPEC_Requirement - This type of property block is used to write the LTL for-
<table>
<thead>
<tr>
<th>Scope Pattern</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Globally</td>
<td>A defined behavior must be true always</td>
</tr>
<tr>
<td>Before R</td>
<td>A defined behavior must be true before occurrence of event 'R'</td>
</tr>
<tr>
<td>After Q</td>
<td>A defined behavior must be true after occurrence of event 'Q'</td>
</tr>
<tr>
<td>Between Q and R</td>
<td>A defined behavior must be true between occurrences of events 'Q' and 'R', in that order. Uses strong until temporal operator (U).</td>
</tr>
<tr>
<td>After Q Until R</td>
<td>Analogous to Between Q and R but Uses weak until temporal operator (W).</td>
</tr>
</tbody>
</table>

Table IV.2: Semantics of Scope Patterns for Temporal Property modeling in VTC

mula representing a temporal property directly as a string in the LTLSPEC attribute of the block without making use of any patterns. The boolean predicates of atomic propositions in an LTL formula must be defined over signal variables which are the same as names of VTC Signal ports inserted in the property block.

3. PropertyTGBA.Requirement - This type of property block is used to model a TGBA that is equivalent to a temporal logic of a property. Figure IV.4 shows an example of this type of property block. States of the TGBA are inserted as VTC_State type objects. A VTC_State has three attributes: LabelName, InitialState, State_Number. A LabelName is a label string for the TGBA state. InitialState is a boolean variable for specifying whether a TGBA state inserted is an initial state or not. State_Number must be unique within a TGBA.

Transitions between TGBA states are inserted as VTC_Transition connections between Transition.Condition ports in two different VTC_States. A Transition.Condition is contained inside a VTC_State and its condition attribute contains the boolean condition for which the transition can be true. The signal variables over which this condition is defined must be the same as the names of VTC_Signal ports contained in the property block. A Transition.Condition also has two other attributes: Acceptance_Transition and Condition_Acceptance. The Acceptance_Transition is a boolean attribute that defines whether a transition is an accepting transition of the TGBA or not. Condition_Acceptance is a string-based attribute and contains a condition for
Figure IV.4: An automaton based (TGBA) property modeled using VTC meta-model

which the transition becomes accepting. An example of a Transition.Condition defined inside a VTC.State is shown in Fig. IV.4.

This type of property block is defined with two attributes: Acceptance.Condition and No.Acceptance.sets. The Acceptance.Condition attribute is a boolean attribute and if it is false then it means that all the transitions of the TGBA are accepting transitions otherwise if it is true then only a subset of all the transitions of the TGBA are accepting. No.Acceptance.sets are the number of accepting transitions sets that the TGBA contains.
A *PatternBased_REQUIREMENT* can have *VTC.BusPortInterface* and *VTC.Signal* type ports. A *VTC.BusPortInterface* type port is a collection of *VTC.SignalPort* type port objects and signifies a named collection of signals rather than a single signal. A *VTC.BusPortInterface* inside a property block indicates that the block is receiving a collection of signals from another component under test in a CyPhyML *TestBench* via a CyPhyML *BusPort* that represents a collection of multiple connection links. To indicate a connection of a property block via a single signal with another component under test inside *TestBench* a *VTC.Signal* port can be used. The *VTC.BusPortInterfaceRef* and *VTC.SignalPortRef* type objects refer to *VTC.BusPortInterface* and *VTC.Signal* type ports. This type of referencing scheme is used when a *PatternBased_REQUIREMENT* is automatically converted to *LTLSPEC.Requirement* and *PropertyTGBA.Requirement* blocks using *Verification Property Conversion* Interpreter Tool (VPC). This conversion is described further in the next section. An *LTLSPEC.Requirement and/or PropertyTGBA.Requirement* block, converted using VPC, contains references to ports in the *PatternBased_REQUIREMENT* block from which they are converted. The *VTC.BusPortInterfaceRef* and *VTC.SignalPortRef* type objects are only used during automated property block conversions. As an alternative to converting from a *PatternBased_REQUIREMENT* block, whenever *LTLSPEC.Requirement and/or PropertyTGBA.Requirement* blocks are developed from scratch, *VTC.BusPortInterface* and *VTC.Signal* type ports can be inserted to receive signals directly from components under test in a *TestBench*. A *RequirementParameter* type object inside a property block is used to define variables that are not signals received by ports in a property block but are used in boolean predicates in propositions for *Proposition and Event_Trigger.Condition* attributes.

All the three types of property blocks can be inserted inside a CyPhyML *TestBench* where they can be connected to other components that need to be verified. Figure IV.5 shows a meta-model that depicts the VTC property block containment and connection relationships within a *TestBench*. A *MapToVTC.Signal* connection represents the connection of the monitor specification to an *OutputSignalPort* of the component under test; a *Bus-
Port.MapToVTC_Signal connection represents the connection of the monitor specification to a BusPort of the component under test. For the purpose of verification of a component with the use of VTC and CBMC, InputRangeGuarantee type objects are associated with InputSignalPort type ports of a component under test to define a range on the component’s input signals. The two attributes, Maximum and Minimum, of InputRangeGuarantee specify the minimum and maximum values for an input signal. These values are used to generate $\_CProverAssume()$ (refer to Section III.8) statements for CBMC verification. To define ranges for input signals that are collectively presented by a BusPort of a component, a RangeGuarantee_Container_BusPort can be used which can contain one or more InputRangeGuarantee objects/ blocks. Figure IV.6 shows an example of a TestBench constructed for CBMC verification using meta-models described above.

IV.4 Pattern-based property to equivalent TGBA to C code monitor translation

Verification Property Converter (VPC) is the interpreter tool of VTC that converts a TGBA from a pattern-based temporal property or an LTL formula. VPC is developed in C++ using the object-oriented API for CyPhyML and VTC meta-models using UDM [56]. VPC translates the transitions of the TGBA into a set of if-else statements (refer to Fig. III.12). Figure
Figure IV.6: Example of a CyPhyML TestBench constructed using the VTC meta-model
IV.7 shows the usage for VPC. Figure IV.3 shows the icon for VPC tool in GME. During the conversion of a PatternBased Requirement to a PropertyTGBA Requirement using VPC, an LTLSPEC Requirement is generated as an intermediate result. All the VTC Signal and VTC BusPortInterface ports contained in the property block are converted to their corresponding reference ports, i.e. VTC SignalPortRef and VTC BusPortInterfaceRef, respectively, in the generated property blocks by VPC. This allows domain engineers to first conveniently model a property using patterns and then invoke VPC on it to generate the property’s equivalent LTL formula and TGBA to further modify the properties if needed or use them as is. To convert a pattern-based property to an LTL formula VPC uses mappings encoded within it which are implemented from [7]. The LTL mappings for different combinations of behavior and scope patterns are given in Table IV.3. VPC then uses the mapped LTL formula as input to the SPOT/FM function (provided within SPOT library) for LTL to TGBA conversion to produce a fine TGBA structure as a result. The TGBA structure resulting from SPOT/FM function is traversed in a Breadth-first-search fashion using its iterators provided within SPOT library to generate a PropertyTGBA Requirement block.

IV.5 Generation of verification facilitated C code files from TestBench

C code representing behavior of cyber components in CyPhyML is generated using the CyPhy2SLCCodeGen interpreter tool (the GME icon for this tool is shown in Fig. IV.3). CyPhy2SLCCodeGen generates .c and .h type of source code and header files for a Simulink/Stateflow model of a cyber component in CyPhyML. The generated code provides initialization and main execution functions for the cyber components. For the purpose of verifica-
<table>
<thead>
<tr>
<th>Behavior Pattern</th>
<th>Scope Pattern</th>
<th>Mapped LTL Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existence(P)</td>
<td>Globally</td>
<td>G &quot;p&quot;</td>
</tr>
<tr>
<td>Before R</td>
<td>!&quot;r&quot; W (&quot;p&quot; &amp; !&quot;r&quot;)</td>
<td></td>
</tr>
<tr>
<td>After Q</td>
<td>G( !&quot;q&quot; &amp; (!&quot;q&quot; &amp; F&quot;p&quot;))</td>
<td></td>
</tr>
<tr>
<td>Between Q and R</td>
<td>G(( !&quot;q&quot; &amp; !&quot;r&quot;) → (?&quot;r&quot; W (&quot;p&quot; &amp; !&quot;r&quot;)) )</td>
<td></td>
</tr>
<tr>
<td>After Q Until R</td>
<td>G(&quot;q&quot; &amp; !&quot;r&quot; → (?&quot;r&quot; U (&quot;p&quot; &amp; !&quot;r&quot;)) )</td>
<td></td>
</tr>
<tr>
<td>Absence(P)</td>
<td>Globally</td>
<td>G !&quot;p&quot;</td>
</tr>
<tr>
<td>Before R</td>
<td>F &quot;r&quot; → ( !&quot;p&quot; U &quot;r&quot;)</td>
<td></td>
</tr>
<tr>
<td>After Q</td>
<td>G( !&quot;q&quot; → G( !&quot;p&quot;) )</td>
<td></td>
</tr>
<tr>
<td>Between Q and R</td>
<td>G(( !&quot;q&quot; &amp; !&quot;r&quot; &amp; F&quot;r&quot; ) → ( !&quot;p&quot; → X( &quot;s&quot; &amp; !&quot;r&quot; ) ) U &quot;r&quot; )</td>
<td></td>
</tr>
<tr>
<td>After Q Until R</td>
<td>G(&quot;q&quot; &amp; !&quot;r&quot; → ( !&quot;p&quot; → X( &quot;s&quot; &amp; !&quot;r&quot; ) ) W &quot;r&quot; )</td>
<td></td>
</tr>
<tr>
<td>Immediate Response(P, S)</td>
<td>Globally</td>
<td>G ( &quot;p&quot; → X &quot;s&quot; )</td>
</tr>
<tr>
<td>Before R</td>
<td>F&quot;r&quot; → ( !&quot;p&quot; → X ( &quot;s&quot; &amp; !&quot;r&quot; ) ) U &quot;r&quot;</td>
<td></td>
</tr>
<tr>
<td>After Q</td>
<td>G( &quot;q&quot; → G( &quot;p&quot; → X&quot;s&quot;) )</td>
<td></td>
</tr>
<tr>
<td>Between Q and R</td>
<td>G(( !&quot;q&quot; &amp; !&quot;r&quot; &amp; F&quot;r&quot; ) → ( !&quot;p&quot; → X(&quot;s&quot; &amp; !&quot;r&quot;) ) U &quot;r&quot; )</td>
<td></td>
</tr>
<tr>
<td>After Q Until R</td>
<td>G(&quot;q&quot; &amp; !&quot;r&quot; → ( !&quot;p&quot; → X(&quot;s&quot; &amp; !&quot;r&quot;) ) W &quot;r&quot; )</td>
<td></td>
</tr>
<tr>
<td>Response(P, S)</td>
<td>Globally</td>
<td>G ( &quot;p&quot; → F &quot;s&quot; )</td>
</tr>
<tr>
<td>Before R</td>
<td>F&quot;r&quot; → ( !&quot;p&quot; → ( !&quot;r&quot; U ( &quot;s&quot; &amp; !&quot;r&quot; ) ) ) U &quot;r&quot;</td>
<td></td>
</tr>
<tr>
<td>After Q</td>
<td>G( &quot;q&quot; → G( &quot;p&quot; → F&quot;s&quot;) )</td>
<td></td>
</tr>
<tr>
<td>Between Q and R</td>
<td>G(( !&quot;q&quot; &amp; !&quot;r&quot; &amp; F&quot;r&quot; ) → ( !&quot;p&quot; → ( !&quot;r&quot; U ( &quot;s&quot; &amp; !&quot;r&quot; ) ) ) U &quot;r&quot; )</td>
<td></td>
</tr>
<tr>
<td>After Q Until R</td>
<td>G(&quot;q&quot; &amp; !&quot;r&quot; → ( !&quot;p&quot; → ( !&quot;r&quot; U ( &quot;s&quot; &amp; !&quot;r&quot;) ) ) W &quot;r&quot; )</td>
<td></td>
</tr>
<tr>
<td>Precedence(P, S)</td>
<td>Globally</td>
<td>G !&quot;p&quot; W &quot;s&quot;</td>
</tr>
<tr>
<td>Before R</td>
<td>F&quot;r&quot; → ( !&quot;p&quot; U ( &quot;s&quot; &amp; !&quot;r&quot; ) )</td>
<td></td>
</tr>
<tr>
<td>After Q</td>
<td>G( !&quot;q&quot;</td>
<td>F( !&quot;q&quot; &amp; ( !&quot;p&quot; W &quot;s&quot;) )</td>
</tr>
<tr>
<td>Between Q and R</td>
<td>G(( !&quot;q&quot; &amp; !&quot;r&quot; &amp; F&quot;r&quot; ) → ( !&quot;p&quot; → ( !&quot;r&quot; U ( &quot;s&quot; &amp; !&quot;r&quot; ) ) )</td>
<td></td>
</tr>
<tr>
<td>After Q Until R</td>
<td>G(&quot;q&quot; &amp; !&quot;r&quot; → ( !&quot;p&quot; W ( &quot;s&quot; &amp; !&quot;r&quot;) ) )</td>
<td></td>
</tr>
</tbody>
</table>

Table IV.3: Mappings between VTC Property Patterns and LTL formulae

The verification of a component a verification wrapper file is generated by the CyPhyTB2Ccode.Verification interpreter tool in an automated fashion, wherein an observing monitor (C code fragment for a property) is integrated with the execution calls of the cyber components in the TestBench. This enables the monitor to investigate the output signals of the component under test. Like VPC, CyPhyTB2Ccode.Verification is also developed in C++ using the API for VTC and CyPhyML meta-models (the API is generated using UDM). CyPhyTB2Ccode.Verification uses Google CTemplate file for template based C code generation. The template used for wrapper file generation is given below:

```c
1 // Header Declarations
2 #include <stdio.h>
3 {(#HEADER_FILE)}#include "{{FILE}}"
4 {(/HEADER_FILE)}
5
6 #define bool int
```

58

#define true 1
#define false 0

// Structure for the TGBA Automaton
{{TGBA_CODE}}

// Main function
int main(void)
{
    // Variable Declarations : Type Double ; Name: Port/
    RequirementParameter/InSignalRangeGuarantee names
    {{#VAR_DECL}}double {{VAR_NAME}} {{#INIT_ZERO}} = 0 {{/INIT_ZERO}};{{/VAR_DECL}}

    // Declaring Contexts for all the TopLevel Subsystems inside all
    found SignalFlowModels
    {{#SUBSYS_CONTEXT_OBJECT}}{{SFTOPLEVELSUBSYS_NAME}}_context {{SFTOPLEVELSUBSYS_NAME}}_context_Object;

    // Initializing the Signal Flow Models by initializing their
    Toplevel Subsystems
    {{#SUBSYS_INIT_CALL}}{{SFTOPLEVELSUBSYS_NAME}}_init(&{{SFTOPLEVELSUBSYS_NAME}}_context_Object);

    while(1)
    {
        // Declaring Assumptions for CBMC Verification
        {{#ASSUME}}{{ASSUME_VALUE}} = nondet_double();
        __CPROVER_assume({{ASSUME_VALUE}}>={{LOW_VALUE}} && {{ASSUME_VALUE}}<={{HIGH_VALUE}});
          {{/ASSUME}}
// Execution Calls to Toplevel Subsystems
{{SUBSYS_MAIN_CALL}}{{CALL}};
{{/SUBSYS_MAIN_CALL}}

printf("OUTPUT---- {{VAR_DECL}} {{VAR_NAME}}: %4.2f,{{/VAR_DECL}}");

// Verification TGBA Automaton Observe Call
{{OBSERVE_CALL}}

// Verification TGBA Acceptance check - For eventuality defining properties.
{{ACCEPTANCE_CALL}}

return 0;
}

In the template given above, lines 1-4 are used to print the header file names for all the cyber components in TestBench. A C code structure for an observing monitor is printed in the wrapper file using the template tag in line 9. The generation of the monitor’s C code structure is explained later in this section. The main function in line 11 becomes the entry point for CBMC symbolic simulation, if no other specific function is stated to be an entry point while using CBMC (CBMC can check explicit functions if the function’s identifier is provided to it, otherwise it treats the main function as an entry point). Line 14 is used to print variable declarations for input and output signals (names of interface ports) of all cyber components in the TestBench. Line 22 is used to print the initialization functions for all the cyber components in the TestBench, which is necessary before calling their main execution functions. In line 25, an infinite while loop is used to indicate an infinite number
of execution cycles for the cyber components under test. To enable only a finite number of executions for cyber components, this `while` loop can be manually modified to a `for` loop. Line 29 is used to print CBMC-compatible assume statements (refer to Chapter II for CBMC assume functions) for input signal variables of the cyber components that are assigned random test values for guiding the execution of the cyber components. Assume statements are only printed for the input signal variables of interface ports which are associated with `InputRangeGuarantee` type objects in the `TestBench`. The random values for these variables are generated using CBMC’s random value generator function (refer to Chapter II) as given in line 28. Line 34 is used to print the main execution function calls of the cyber components in the `TestBench`. These functions are provided input and output variables as parameters. The values of output variables change as per the behavioral logic of the main execution functions. Line 40 is used to print the observe function calls of observing monitors. The functions are passed a set of input variables as parameters for the output signal variables of the cyber components composed within the system under test. The variables represent the state of the system under test and are checked after every single execution cycle of all the cyber components within the system. This enables monitoring based verification. The acceptance function calls of the monitors are printed using line 44. The descriptions of monitor’s observe and acceptance calls are discussed with the description of observing monitor’s C code generation in the next paragraph.

The observing monitors are C code translations of TGBA equivalents of temporal properties. As discussed in Section IV.4, the TGBA equivalent (`PropertyTGBA.Requirement` for a modeled `PatternBased.Requirement` or `LTLSPEC.Requirement` property block can be generated using VPC. The VPC-generated TGBA property block is translated to C code by `CyPhyTB2Ccode.Verification` tool. To perform this translation another CTemplate file is used that is given below:

```c
1 struct {{SPEC_NAME}}
2 {
3     // C code implementation
4 }
```
bool accepting_observer;
bool accepted;
int state;
{(SPEC_NAME)}_obj={{{ACC_OBS}}, false, {{{INIT_STATE}}}};

void {SPEC_NAME}_observe(struct {SPEC_NAME} *specobj, double {SIGNAL_NAME})
{
    printf("OUTPUT---------TGBA state: %d", specobj->state);

    {#REQUIREMENT_PARAMETERS}double {RP_NAME} = {RP_VALUE};{#REQUIREMENT_PARAMETERS}
    specobj->accepted = false;
    {#TRANSITION}
    // STATE_{(CURR_STATE)} ----{CONDITION}----> STATE_{(NEW_STATE)}
    if ((specobj->state=={{CURR_STATE}}) && {CONDITION})
        {#ACCEPTANCE_TRANSITION}specobj->accepted = true;{#ACCEPTANCE_TRANSITION}
    else{#OBSERVE_ASSERT}
        __CPROVER_assert(0,"{SPEC_NAME} violated!");
    }{#OBSERVE_ASSERT}
}

void {SPEC_NAME}_acceptance_check(struct {SPEC_NAME} *specobj)
{
    if (specobj->accepting_observer == true)
    {
        if (specobj->accepted == false)
In lines 1-6 of the template given above, a C structure is defined for a TGBA-based observing monitor. The C structure for a TGBA has three member variables: accepting observer, accepted, state. The member variable accepting observer is a boolean variable which when is TRUE signifies that the TGBA has only a subset of accepting transitions, and when is FALSE then signifies that all are accepting transitions (refer to Chapter II and Fig. III.11). The member variable accepted is a boolean variable which is checked in the acceptance check call (as in lines 24-34) whenever a TGBA has only a subset of transitions as accepting ones. The member variable state specifies the current state of a TGBA during an execution cycle and is used in the conditions of if-else statements in the monitor’s observe call (as in lines 7-23). As described in Chapter II, a TGBA can be represented by a set of if-else statements which are based upon the transitions of the given TGBA (refer to Fig. CcodeTGBAEx1). The set of if-else statements in a monitor’s observer call represent the transitions of the TGBA. The assert functions in lines 21 and 31 are used as property violation assertion statements. These assert statements are used as user-based assertions by CBMC for checking violation of the given property. If these assert statements are reachable during symbolic execution of CBMC then it proves violation of the given property, otherwise it disproves any such violation within the loop unwinding bound analyzed by CBMC. Examples of C code generation for a TGBA by CyPhyTB2Ccode can be seen in the illustrative examples that are discussed later in this chapter.

CyPhyTB2Ccode is invoked from within a property block. Once invoked it generates the verification-enabled C code file as follows:
1. If the interpreter tool is invoked from within a PatternBased.Requirement or a LTL-SPEC.Requirement block then it searches for its equivalent PropertyTGBA.Requirement block within the same CyPhyML TestBench. The name of the equivalent PropertyTGBA.Requirement block is composed of the source property block’s name and an augmented string ”_TGBA”. For example, to search for an equivalent PropertyTGBA.Requirement for a source property block with name 'Prop1', the interpreter searches for a PropertyTGBA.Requirement block with name 'Prop1_TGBA' within the same TestBench. This step is not performed by the interpreter if it is invoked from within a PropertyTGBA.Requirement block.

2. The interpreter generates the C code for the TGBA modeled inside the PropertyTGBA.Requirement that is found above.

3. The interpreter traverses the components in given TestBench that are connected to the property block. Every MapToVTC.connection type connection for a property block’s interface ports (or ports referenced by property block’s VTC.SignalPort_Ref and VTC.BusPortInterface_Ref type objects) are traversed in the reverse direction in a Breadth-first-search fashion. If two components A and B are connected to each other such that output of A drives the input for B in the model, then according to the concept of causality relationships for input stimulations of different components that are connected to each other, A must be executed before B to provide meaningful input to B. Hence, a reverse Breadth-first-search traversal approach is used to make sure that the sequence of main execution functions of cyber components in the TestBench printed in the wrapper file adheres to the causality relationships between interconnected components. Currently for use of the CyPhyTB2Ccode.Verification tool, it is assumed that the graph of interconnected cyber components in a TestBench has no cycles, as there is no mechanism developed to resolve cyclic dependencies between the components within a CyPhyML TestBench for C code generation to enable
verification with CBMC.
CHAPTER V

ILLUSTRATIVE EXAMPLE

V.1 Ignition Model

In this section a Simulink/Stateflow model is discussed as an experimental example. The Stateflow given in Fig. V.1 is a model of Ignition Logic controller of a vehicle which controls the ignition light on the dashboard of the vehicle and controls the starter of the vehicle based on a signal from the ignition key and the running status of the engine of the vehicle. Table V.1 gives the description of the I/O signals and variables used in the model. The expected behavior of the controller is as follows: when the ignition key is turned on while the engine is not running, the starter should be engaged so as to start the engine and be disengaged once the engine has started. The ignition light on the dashboard must reflect the status of the engine at all times correctly.

This model was imported to the CyPhyML environment in GME using MDL2MGA, the Simulink import utility, and its behavioral C code was synthesized using CyPhy2SLC_Code-Gen. Using VTC, a few temporal properties were modeled in GME and were linked to the CyPhyML wrapper component which contained the imported model. For every property a separate CyPhyML Testbench was prepared and C code verification files for CBMC were

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Data Type</th>
<th>Type of Signal</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_pos: integer or double</td>
<td>Input</td>
<td>[0, 2]</td>
<td>Signifies position for ignition key. 0 - key off, 1 - key on (electrical system turned on), 2 - turn on engine</td>
<td></td>
</tr>
<tr>
<td>engine_running: integer or double</td>
<td>Input</td>
<td>[0, 2]</td>
<td>For status of engine. 0 - engine is off, 1 - engine is running</td>
<td></td>
</tr>
<tr>
<td>ignition_signal: integer or double</td>
<td>Output</td>
<td>[0, 2]</td>
<td>Status for ignition on dashboard of vehicle. 0 - engine off, 1 - engine running, 2 - engine is getting started</td>
<td></td>
</tr>
<tr>
<td>engage_starter: integer or double</td>
<td>Output</td>
<td>[0, 1]</td>
<td>Status of starter. 0 - starter off, 1 - starter on</td>
<td></td>
</tr>
</tbody>
</table>

Table V.1: IO Signals description for Ignition Logic controller
Figure V.1: Stateflow design for Ignition Logic controller
generated by using VPC and CyPhyTB2Ccode_Verification interpreter. Figure V.2 shows the TestBench developed for property 1 (the details of the property are discussed later in this section). In the given figure, IgnitionController is the CyPhyML component wrapper for the Stateflow model and InputRangeGuarantee objects are defined for both input signals. For key_pos the range is specified to be [0, 2] and for engine_running it is [0,1].

At the model level, NuSMV was used to perform verification of the same properties that were intended to be checked by CBMC. Bounded model checking was performed with both the verification tools, CBMC and NuSMV, with a bound of 30. The NuSMV representation of the controller was written manually and is given in Appendix VII.1. As only integer values are allowed in NuSMV’s modeling language, the non-integer values were approximated to the closest relevant integer values.

Verification property modeling using patterns in VTC of three properties, their equivalent LTL formulae and TGBA, and C code translations used for CBMC verification are given in forthcoming sub-sections. The experimental results are summarized in the final sub-section.
V.1.1 Property 1: States of the Engine and the Ignition Light

The first property to be verified was - "the engine should be running before the ignition light reflects that the engine is running". The parameters for the pattern-based property block modeled using VTC for this property and its equivalent LTL formula generated using VPC is as follows:

```
PatternBasedRequirement:
  Behavior Pattern: Precedes(P & S) - S precedes P:
    S: (engine_running>=0.5),
    P: (ignition_signal==1.00)
  Scope Pattern: Globally
LTLSPECRequirement:
  LTLSPEC: !"ignition_signal==1.00" W "engine_running>=0.5"
```

In NuSMV the weak until operator (W) is not valid and hence the above LTL formula is re-written with strong until operator (U) for NuSMV as (using conversion given in [7]) -

```
!(Ignition Logic.ignition_signal=1) U ((engine_running=1) | G(!(Ignition Logic.ignition_signal=1)))
```

The TGBA equivalent for the LTL formula for Property 1 is given in Fig. VII.1 where all of the transitions are accepting transitions.
The `if-else` statements generated by using VTC from the TGBA in Fig. VII.1 are given in the above code snippet. State 1 is the initial state for the TGBA. The if-statement on line 2 is specified with the condition on the transition from state 1 to state 2 in the TGBA. When the condition is true the current state member of the TGBA object is set to state 2 from state 1. Line 7 shows the if-statement with condition on the self-looping transition of state 1. When the condition becomes true then the state of the TGBA object doesn’t change as shown in line 10. The self-looping transition’s condition on state 2 of the TGBA is specified as if-statement on line 14. If none of these if-statements hold true at any point during the symbolic execution of the code by CBMC, then the assert statement in the else-
block on line 19 will be reached resulting in violation of the property. In the context of Property 1 it can be seen that when the engine is not running and the ignition light remains on then none of the transitions’ condition from state 1 can hold true resulting in violation. This behavior of the automaton is desired as per the logic of the property.

```c
// Initializing the Signal Flow Models by initializing their Toplevel Subsystems
ignition_init(&ignition_context_Object);

while(1)
{
    // Declaring Assumptions for CBMC Verification
    engine_running = nondet_double();
    __CPROVER_assume(engine_running>=0 && engine_running<=1);

    key_position = nondet_double();
    __CPROVER_assume(key_position>=0 && key_position<=2);

    // Execution Calls to Toplevel Subsystems
    ignition_main(&ignition_context_Object, key_position, engine_running,
                  &engage_starter, &ignition_signal);

    printf("OUTPUT---- engage_starter: %4.2f, engine_running: %4.2f,
            ignition_signal: %4.2f, key_position: %4.2f,", engage_starter,
            engine_running, ignition_signal, key_position);

    // Verification TGBA Automaton Observe Call
    engineON_PRECEDES_ignitionON_TGBA_observe(&
        engineON_PRECEDES_ignitionON_TGBA_obj, ignition_signal,
        engine_running);
}
```
The above code listing shows a snippet from the generated verification-enabled C code wrapper file from the TestBench for Property 1 by the CyPhyTB2Ccode_Verification tool. The snippet is taken from the main function of the generated C file. Line 2 contains the initialization call to the ignition controller model. The function is defined in a header file specified in an include statement at the beginning of the generated file. The header file is one of the generated file from the code-generator under test that is used for generating the behavior C code for the controller model. Though line 4 contains an infinite while loop, CBMC will only unwind it till the specified bound, which is 30 in this case. In lines 7 & 10 the input signals to the controller are assigned a random value. If the values are not within the range as specified in the assume-statements in lines 8 & 11 then CBMC will avoid unwinding the loop further during the current iteration and will reiterate through the loop. The value range specified in the assume-statements comes from the values as were specified in the InputRangeGuarantee atoms in the verification TestBench. Line 14 contains the execution call to the controller model to make it proceed to the next logical time step. In the statement on line 19 a call to execute TGBA monitor is made where the property is checked for violation during every iteration of the containing while loop. It should be noted here that the possibility of generating same combinations of the values for the randomly assigned variables always remains. To make sure that such situation does not prevail, the output from CBMC was manually observed to check for the assignment of all possible permutations and combinations of random values that were being assigned to relevant variables. The complete generated file is given in Appendix VII.2.2 for reference.

After performing verification with NuSMV and CBMC, the results from both tools proved satisfiability of the verification property.

V.1.2 Property 2: Constraint on the Starter’s Engage state

The second property for verification states - “if the ignition key is turned on when the engine is not running then the starter should be engaged next so as to start the engine”. Using
VTC, the property was modeled as follows:

**PatternBased Requirement:**
- Behavior Pattern: Immediate Response (P & S) - S occurs next after P:
  - S: (engage_starter==1.00)
  - P: (key_position>1.00 && engine_running<1.00)
- Scope Pattern: Globally

**LTLSPEC Requirement:**
- \( \mathit{LTLSPEC}: G \left( \neg \left( \text{key_position}>1.00 \land \text{engine_running}<1.00 \right) \implies X \text{engage_starter==1.00} \right) \)

The LTL formula for property 2 used in NuSMV verification is:

\[
\mathit{LTL}: G \left( \left( \text{key_position}>1.00 \land \text{engine_running}<1.00 \right) \implies X \text{engage_starter==1.00} \right)
\]

The TGBA equivalent for the LTL formula for Property 2 is given in Fig. VII.2. The generated C code *wrapper file* for CBMC verification is given in Appendix VII.3.

Verification with both NuSMV and CBMC resulted in violation of the property with the following counterexample traces:

**NuSMV Counter-Example:**
- State: 1.1 <-
  - key_position = 2
  - engine_running = 0
  - Ignition Logic.ignition_signal = 0

![Figure V.4: TGBA equivalent of Verification Property 2 of the Ignition Logic Controller](image)
The NuSMV tool prints the values for only those variables in the counter-example trace which get changed when entering into a new state. In the above counter-example trace from NuSMV, it can be seen that during State 1.1 the ignition key’s position value and the engine’s running state variables indicate that the ignition is turned to ON state when the engine is not running. According to the description of Property 2 the starter should be engaged in the next state, that is, the engage_starter signal should get ’1’ assigned to it in State 1.2. But, in the counter-example trace there is no information printed out on the starter’s signal in State 1.2 indicating no change in the signal’s value from State 1.1. Thus, the property is violated. The CBMC counter-example trace shown for the violation of the property can be understood in the same manner from its two output statements.

V.1.3 Property 3: Transition of the Starter states

The third property for verification states - "always whenever the ignition key is turned off and the starter is on then next the starter should be disengaged". Using VTC, the property was modeled as follows:
Figure V.5: TGBA equivalent of Verification Property 3 of the Ignition Logic Controller

PatternBased
Requirement:

Behavior Pattern: Immediate Response (P & S) - S occurs next after P:

S: (engage_starter<1.00)
P: (key_position<1.00 && engage_starter>0.00)

Scope Pattern: Globally

LTLSPEC
Requirement:

LTLSPEC: \(G ( "key_position<1.00 && engage_starter>0.00" \rightarrow X "engage_starter<1.00" )\)

The LTL formula for property 3 used in NuSMV verification is:

\(G ( (key_position<1 \& Ignition Logic.engage_starter>0) \rightarrow X (Ignition Logic.engage_starter<1) )\)

The TGBA equivalent for the LTL formula for Property 3 is given in Fig. VII.3. The generated C code wrapper file for CBMC verification is given in Appendix VII.4.

Verification with both NuSMV and CBMC resulted in violation of the property with following counterexample traces:

NuSMV Counter-Example:

\(-\rightarrow State: 1.1<-
key_position = 2
engine_running = 0
Ignition Logic.ignition_signal = 0
Ignition Logic.engage_starter = 0
Ignition Logic.state = Off
\(-\rightarrow State: 1.2<-\)
engine_running = 1
Ignition Logic.ignition_signal = 2
Ignition Logic.state = Start
-- Loop starts here
-> State: 1.3 <-
key_position = 0
Ignition Logic.ignition_signal = 1
Ignition Logic.engage_starter = 1
Ignition Logic.state = On
-> State: 1.4 <-

CBMC Counter-Example:
OUTPUT---- engage_starter: 0.00, engine_running: 0.00,
ignition_signal: 0.00, key_position: 1.00,
OUTPUT---- engage_starter: 0.00, engine_running: 0.00,
ignition_signal: 0.00, key_position: 2.00,
OUTPUT---- engage_starter: 0.00, engine_running: 1.00,
ignition_signal: 0.00, key_position: 0.00,
OUTPUT---- engage_starter: 0.00, engine_running: 0.00,
ignition_signal: 0.00, key_position: 0.00,
OUTPUT---- engage_starter: 0.00, engine_running: 0.00,
ignition_signal: 0.00, key_position: 2.00,
OUTPUT---- engage_starter: 1.00, engine_running: 0.00,
ignition_signal: 0.00, key_position: 0.00,
OUTPUT---- engage_starter: 1.00, engine_running: 0.00,
ignition_signal: 0.00, key_position: 0.00,
ignition_signal: 0.00, key_position: 0.00,

The NuSMV tool prints the values for only those variables in the counter-example trace which get changed when entering into a new state. In the above counter-example trace from NuSMV, it can be seen that during State 1.3 the ignition key’s position value indicates that the ignition is switched off. According to the description of Property 3 the starter should be disengaged in the next state, that is, the engage_starter signal should get ’0’ assigned to it in State 1.4. But, in the counter-example trace there is no information printed out for the
State 1.4 indicating no change in any variable’s value from State 1.3. Thus, the property is violated. The CBMC counter-example trace shown for the violation of the property can be understood in the same manner from its last two output statements where turning off the ignition key doesn’t change the signal to the starter in the next step.

V.1.4 Experiment Results Summary

By performing verification of critical properties for the Ignition Logic controller we intend to support correct code generation from the C code generator tool, CyPhy2SLCCodeGen, for CyPhyML cyber models. As per the consistency in the results of NuSMV and CBMC verification, as summarized in Table V.2, we are able to generate better trust on the translation by the code generator tool under test.

<table>
<thead>
<tr>
<th>Verification Property</th>
<th>NuSMV Verification</th>
<th>CBMC Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>Not violated</td>
<td>Not violated. Verification Time: 36.68 sec</td>
</tr>
<tr>
<td>Property 2</td>
<td>Violated</td>
<td>Violated. Verification Time: 36.323 sec</td>
</tr>
<tr>
<td>Property 3</td>
<td>Violated</td>
<td>Violated. Verification Time: 36.804 sec</td>
</tr>
</tbody>
</table>

Table V.2: Verification results for Ignition Logic controller
VI.1 Conclusions

The presented tool chain, Verification Tool Chain (VTC), is expected to reduce the efforts of domain-specific design engineers and encourage to include verification procedures during CPS development life cycles. The pattern-based modeling feature of the toolchain allows use of English-like patterns to model complex temporal properties, and the automated workflow makes the verification procedure less tedious. We do not claim that the patterns to model verification properties in VTC are sufficient to model any complex temporal logic, but, as mentioned by Dwyer et al. [36], [37], the captured patterns allow a major percentage of the verification properties in the industry to be modeled conveniently. Additionally, the engineers who find it more intuitive to model properties as automata are provided with the automata-based property modeling feature in VTC. Integration of developed interpreter tools with GME and automation schemes for VTC enable quick and easy way of integration of verification schemes with development life-cycle for CPS. With the use of VTC, major behavioral errors in the systems can be exposed early during development to benefit from in the long run. Verification of functional behavior preservation during translation of the synthesized code leads to trustable deployable code in-house-developed interpreter or translator tools. Given the above factors, the overall cost and time for development is expected to reduce significantly. Automated property augmentation and propagation schemes will enable different engineering teams working on a project easily adhere to the safety requirements of systems and verify them during each and every phase of the development life-cycle in a convenient and automated manner. The discussed impacts are shown in a visual diagram in Fig. VI.1.

During the course of this research work, major lessons were learned regarding verifi-
cation of systems. First, we learned that integration of verification workflows with development life cycles of CPS is difficult without domain-specific knowledge of the systems and model-based design approach remains beneficial here. This is because, the domain-specific knowledge in model-based design allows the interpreter tools to be more context-aware which enables us to tackle system models in a much better way. Second, we realized the limitations of bounded model checking. We performed bounded model checking with CBMC and NuSMV, which proves absence of errors down to certain depths in the behavioral state-spaces of the system models under test. This may lead to spurious examples. Third, we realized the difficulty in using property monitors to verify asynchronous system models. As a meaningful logical time step is hard to be defined for the asynchronous systems we can not decide the sampling durations for the output signals from the models to be observed by the monitors. This raises difficulty in using monitors in asynchronous or distributed environments. Fourth, we learned that C code checking with CBMC is still computationally cumbersome and large state-spaces of generated property monitors or systems under test leads CBMC to halt with insufficient memory for further computations. The discussed lessons are presented as a visual diagram in Fig. VI.2.

Our approach is not directed towards providing a complete context-free proof of correctness of the code-generators. Nevertheless, establishing that the code-generators are
property-preserving while generating code for software components proves useful for developing trust on them during fast-paced CPS development life cycles.

VI.2 Future Work

Currently, verification with NuSMV is performed by manually translating Stateflow designs to the NuSMV language. As a future work, it will be helpful to extend VTC metamodel and its toolchain so as to perform translation of CyPhyML cyber models to NuSMV in an automated manner. Also, as explained in Section VI.1 and Chapter V, the bounded model checking approach used here doesn’t provide complete proof of a property. Rather, it only proves a model or its generated code to satisfy a verification property only up to a certain bound in the behavioral state-space. Hence, to avoid generation of spurious results, in the future, it shall be helpful to develop reachability analysis tools for CyPhyML components so that pre-mediated trustworthy bounds can be specified before the application of bounded model checking.
CHAPTER VII

APPENDIX

VII.1 NuSMV translation of Ignition Logic Controller

```plaintext
MODULE main()
VAR
key_position : {0, 1, 2};
engine_running : {0, 1};
Ignition Logic : stateflow(engine_running, key_position);

MODULE stateflow(engine_running, key_position)
VAR
ignition_signal : {0, 1, 2};
engage_starter : {0, 1};
state : {Off, Start, On};

ASSIGN
init(ignition_signal) := 0;
init(engage_starter) := 0;
init(state) := Off;

next(state) :=
case
state = Off & (key_position > 1) & (engine_running < 1) : Start;
state = Start & (engine_running = 1) : On;
state = Off & (engine_running = 1) : On;
state = On & (engine_running < 1) : Off;
TRUE : state;
esac;

next(ignition_signal) :=
```

81
case
    state = Off & (key_position > 1) & (engine_running < 1) : 2;
    state = Off & (engine_running = 1) : 1;
    state = Start & (engine_running = 1) : 1;
    state = On & (engine_running < 1) : 0;
    TRUE : ignition_signal;
esac;

next(engage_starter) :=
case
    state = On & (engine_running < 1) : 0;
    state = Start & !(engine_running = 1)) : 1;
    TRUE : engage_starter;
esac;

VII.2  Property 1 of Ignition Logic Controller

VII.2.1  TGBA equivalent of Property

![TGBA diagram]

Figure VII.1: TGBA for Property 1 for Ignition Logic controller

VII.2.2  C code verification wrapper file

```c
// Header Declarations
#include <stdio.h>
#include "ignition.sl.h"

#define bool int
#define true 1
```

82
# define false 0

// Structure for the TGBA Automaton

struct engineON_PRECEDES_ignitionON_TGBA
{
    bool accepting_observer;
    bool accepted;
    int state;
} engineON_PRECEDES_ignitionON_TGBA_obj={false, false, 1};

void engineON_PRECEDES_ignitionON_TGBA_observe(struct engineON_PRECEDES_ignitionON_TGBA *specobj, double ignition_signal, double engine_running)
{
    printf("OUTPUT———TGBA state: %d", specobj->state);

    specobj->accepted = false;

    // STATE_1 ———((engine_running >=0.5))——> STATE_2
    if (specobj->state ==1) && ((engine_running >=0.5))
    {
        specobj->state = 2;
    }
    else
    {
        // STATE_1 ———(!ignition_signal ==1.00) && !(engine_running >=0.5))——> STATE_1
        if (specobj->state ==1) && (!ignition_signal ==1.00) && !((engine_running >=0.5))
        {
            specobj->state = 1;
        } else
        {

        }
    }
}
// STATE_2 ----(1)----- > STATE_2
if ( (specobj->state == 2) && (1) )
{
specobj->state = 2;
}
else
{
 CPROVER assert (0, "engineON_PRECEDES_ignitionON_TGBA_violated!");
}

void engineON_PRECEDES_ignitionON_TGBA_acceptance_check(struct
 engineON_PRECEDES_ignitionON_TGBA *specobj)
{
if (specobj->accepting_observer == true)
{
if (specobj->accepted == false)
{
 CPROVER assert (0, "engineON_PRECEDES_ignitionON_TGBA_violated!");
}
}

// Main function
int main(void)
{
// Variable Declarations: Type Double; Name: Port/RequirementParameter
  // InSignalRangeGuarantee names
 double engage_starter;
double engine_running;
double ignition_signal;
double key_position;

// Declaring Contexts for all the TopLevel Subsystems inside all found SignalFlowModels

ignition_context ignition_context_Object;

// Initializing the Signal Flow Models by initializing their TopLevel Subsystems
ignition_init(&ignition_context_Object);

while (1)
{
    // Declaring Assumptions for CBMC Verification
    engine_running = nondet_double();
    _CPROVER_assume(engine_running >=0 && engine_running <=1);

    key_position = nondet_double();
    _CPROVER_assume(key_position >=0 && key_position <=2);

    // Execution Calls to TopLevel Subsystems
    ignition_main(&ignition_context_Object, key_position, engine_running, &
                      engage_starter, &ignition_signal);

    printf("OUTPUT—— engage_starter: %4.2f, engine_running: %4.2f,
           ignition_signal: %4.2f, key_position: %4.2f," , engage_starter,
           engine_running, ignition_signal, key_position);

    // Verification TGBA Automaton Observe Call
    engineON_PRECEDES_ignitionON_TGBA_observe(&
                                                engineON_PRECEDES_ignitionON_TGBA_obj, ignition_signal,
                                                engine_running);

}
// Verification TGBA Acceptance check – For eventuality defining properties.
engineON_PRECEDES_ignitionON_TGBA_acceptance_check(&
    engineON_PRECEDES_ignitionON_TGBA_obj);
return 0;
}

VII.3 Property 2 of Ignition Logic Controller

VII.3.1 TGBA equivalent of Property

Figure VII.2: TGBA for Property 2 for Ignition Logic controller

VII.3.2 C code verification wrapper file

// Header Declarations
#include <stdio.h>
#include "ignition_sl.h"

#define bool int
#define true 1
#define false 0

// Structure for the TGBA Automaton
struct keyONengineOFF_NEXTstarterON_TGBA
{
    bool accepting_observer;
bool accepted;
int state;
}

void keyONengineOFF_NEXTstarterON_TGBA_obj={false, false, 1};

void keyONengineOFF_NEXTstarterON_TGBA_observe(struct
keyONengineOFF_NEXTstarterON_TGBA *specobj,
double engage_starter,
double engine_running,
double key_position)
{
    printf("OUTPUT----TGBA state: %d", specobj->state);
    specobj->accepted = false;

    // STATE_1 ----(1)----> STATE_2
    if ((specobj->state == 1) && (1))
    {
        specobj->state = 2;
    }
    else
        // STATE_1 ----(!key_position >1.00 && engine_running <1.00))----> STATE_1
        if ((specobj->state == 1) && (!key_position > 1.00 && engine_running < 1.00))
        {
            specobj->state = 1;
        }
    else
        // STATE_2 ----((engage_starter==1.00))----> STATE_2
        if ((specobj->state == 2) && ((engage_starter ==1.00)) )
        {
            specobj->state = 2;
        }
    else
        // STATE_2 ----((engage_starter==1.00) && !(key_position >1.00 &&
        // engine_running <1.00))----> STATE_1

if ( (specobj->state == 2) && (engage_starter == 1.00) && ! (key_position > 1.00 && engine_running < 1.00) )
{
specobj->state = 1;
}
else
{

_CPROVER_assert(0,"keyONengineOFF_NEXTstarterON_TGBA violated!");
}
}

void keyONengineOFF_NEXTstarterON_TGBA_acceptance_check(struct keyONengineOFF_NEXTstarterON_TGBA *specobj)
{
if (specobj->accepting_observer == true)
{
if (specobj->accepted == false)
{
_CPROVER_assert(0,"keyONengineOFF_NEXTstarterON_TGBA violated!");
}
}
// Main function
int main(void)
{
// Variable Declarations : Type Double ; Name: Port/RequirementParameter /InSignalRangeGuarantee names
double engage_starter;
double engine_running;
double key_position;
double ignition_signal;
// Declaring Contexts for all the TopLevel Subsystems inside all found SignalFlowModels
ignition_context ignition_context_Object;

// Initializing the Signal Flow Models by initializing their Toplevel Subsystems
ignition_init(&ignition_context_Object);

while (1)
{
    // Declaring Assumptions for CBMC Verification
    key_position = nondet_double();
    _CPROVER_assume(key_position >= 0 && key_position <= 2);

    engine_running = nondet_double();
    _CPROVER_assume(engine_running >= 0 && engine_running <= 1);

    // Execution Calls to Toplevel Subsystems
    ignition_main(&ignition_context_Object, key_position, engine_running, &engage_starter, &ignition_signal);

    printf("OUTPUT−−−−− engage_starter: %4.2f, engine_running: %4.2f, key_position: %4.2f, ", engage_starter, engine_running, key_position);

    // Verification TGBA Automaton Observe Call
    keyONengineOFF_NEXTstarterON_TGBA_observe(&
        keyONengineOFF_NEXTstarterON_TGBA_obj, engage_starter, engine_running, key_position);
}

VII.4 Property 3 of Ignition Logic Controller

VII.4.1 TGBA equivalent of Property

Figure VII.3: TGBA for Property 3 for Ignition Logic controller

VII.4.2 C code verification wrapper file

// Header Declarations
#include <stdio.h>
#include "ignition_sl.h"

#define bool int
#define true 1
#define false 0

// Structure for the TGBA Automaton
struct GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA {
    bool accepting_observer;
    bool accepted;
}
int state;
} GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA_obj={false, false, 1};

void GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA_observe(struct
    GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA *specobj,
    double engage_starter, double key_position)
{
    printf("OUTPUT--------TGBA state: %d", specobj->state);
    specobj->accepted = false;

    // STATE.1 -----(!(key_position<1.00 && engage_starter>0.00))---->
    // STATE.1
    if ((specobj->state==1) && (!(key_position<1.00 && engage_starter>0.00))
    )
    {
        specobj->state = 1;
    }
    else
    { // STATE.1 -----(key_position<1.00 && engage_starter>0.00))----> STATE.2
        if ((specobj->state==1) && ((key_position<1.00 && engage_starter>0.00))
        )
        {
            specobj->state = 2;
        }
        else
        { // STATE.2 -----(engage_starter<1.00) && !(key_position<1.00 &&
            engage_starter>0.00))----> STATE.1
            if ((specobj->state==2) && ((engage_starter<1.00) && !(key_position
            <1.00 && engage_starter>0.00)) )
            {
                specobj->state = 1;
            }
```c
else

// STATE_2  ----((engage_starter < 1.00) && (key_position < 1.00 &&
               engage_starter > 0.00))----> STATE_2
if ((specobj->state == 2) && (engage_starter < 1.00) && (key_position < 1.00 &&
    engage_starter > 0.00))
{
    specobj->state = 2;
}
else
{
    _CPROVER_assert(0, "GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA violated!");
}
}

void GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA_acceptance_check(struct GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA *specobj)
{
if (specobj->accepting_observer == true)
{
    if (specobj->accepted == false)
    {
        _CPROVER_assert(0, "GLOBALLYkeyOFFstarterOnNEXTstarterOFF_TGBA violated!");
    }
}

// Main function
int main(void)
```
{  
// Variable Declarations: Type Double; Name: Port/RequirementParameter
    // InSignalRangeGuarantee names
  double  engage_starter;
  double  engine_running;
  double  ignition_signal;
  double  key_position;

// Declaring Contexts for all the TopLevel Subsystems inside all found SignalFlowModels
  ignition_context ignition_context_Object;

// Initializing the Signal Flow Models by initializing their Toplevel Subsystems
  ignition_init(&ignition_context_Object);

while (1)  
{  
// Declaring Assumptions for CBMC Verification
  engine_running = nondet_double();
  _CPROVER_assume(engine_running >= 0 && engine_running <= 1);

  key_position = nondet_double();
  _CPROVER_assume(key_position >= 0 && key_position <= 2);

// Execution Calls to Toplevel Subsystems
  ignition_main(&ignition_context_Object, key_position, engine_running, 
                 engage_starter, &ignition_signal);

  printf("OUTPUT—— engage_starter: %4.2f, engine_running: %4.2f, 
          ignition_signal: %4.2f, key_position: %4.2f,", engage_starter, 
          engine_running, ignition_signal, key_position);
}
// Verification TGBA Automaton Observe Call
GLOBALLYkeyOFFstarterOnNEXTstarterOFF.TGBA.observe(&
    GLOBALLYkeyOFFstarterOnNEXTstarterOFF.TGBA_obj, engage_starter, key_position);
}

// Verification TGBA Acceptance check – For eventuality defining properties.
GLOBALLYkeyOFFstarterOnNEXTstarterOFF.TGBA.acceptance_check(&
    GLOBALLYkeyOFFstarterOnNEXTstarterOFF.TGBA_obj);
return 0;
}

BIBLIOGRAPHY


95


