Rapid Property Specification and Checking for Model-Based Formalisms

Daniel Balasubramanian, Gabor Pap,
Harmon Nine, Gabor Karsai
ISIS / Vanderbilt University, Nashville, TN 37212
Email: {daniel.a.balasubramanian, gabor.pap,
harmon.s.nine, gabor.karsai}@vanderbilt.edu
Michael Lowry, Corina Pasareanu, Tom Pressburger
NASA Ames Research Center, Moffett Field, CA 94035
Email: {michael.r.lowry, tom.pressburger,
corina.s.pasareanu}@nasa.gov

***Abstract*—In model-based development, verification techniques can be used to check whether an abstract model satisfies a set of properties. Ideally, implementation code generated from these models can also be verified against similar properties. However, the distance between the property specification languages and the implementation makes verifying such generated code difficult. Optimizations and renamings can blur the correspondence between the two, further increasing the difficulty of specifying verification properties on the generated code. This paper describes methods for specifying verification properties on abstract models that are then checked on implementation level code. These properties are translated by an extended code generator into implementation code and special annotations that are used by a software model checker.**

I. Introduction

Model-based development (MBD) is a software and system design paradigm based on abstractions called *models*. Domain-specific modeling languages (DSMLs) [1] provide the ability to represent models that are specific to a particular problem domain. Cast in this light, Matlab/Simulink [2] can be viewed as a DSML for physical and embedded systems, as they allow modeling the (dynamics of the) physical plant as well as the behavior of its controller software. Once the model is created, the system can be simulated, outputs observed, and the model changed according to the traces provided by the simulation.

Simulation alone, however, cannot provide rigorous guarantees about a model's behavior. In order to prove exhaustively that a model's dynamic behavior *always* satisfies a set of properties, some sort of *verification* [3] must be performed. Typical properties include state reachability, deadlock-freedom and a wide range of temporal properties. In recent years, model-level verification tools have been developed that can check models for such properties. While these tools play an important role in MBD and can provide guarantees about a model's behavior, their use is often limited to a small portion of a complex system, i.e. key properties and algorithms.

One of the key goals of MBD is to gradually refine abstract, high-level models until they can be automatically synthesized into an implementation that runs on a non-ideal computational platform. However, one crucial problem is often ignored: how can one verify that the synthesized implementation code satisfies the same properties as the models from which it was generated? Without verifying the implementation, the guarantees provided by checking the abstract models are lost. Checking or proving the correctness of the synthesis (transformation) algorithms is an open problem. Further, if no verification is performed on high-level models, then verifying the implementation is the only way to prove properties about the system.

The major difficulty in verifying model level properties on implementation level code lies in the different levels of abstraction. Abstract models are developed by hand and designed with readability in mind, while automatically generated code can be difficult to read. Further, the correspondence between model elements and their generated code is not obvious. Renamings and optimizations make it difficult to understand how a particular model element is represented in the generated code. As a result, knowing where to place properties that are to be verified becomes a challenge.

Another difficulty lies in the mismatch between the input languages of verification tools used at the different levels of abstraction. Individual verification tools typically each use their own input language for defining properties, so that properties checked at the model level must be rewritten in a new syntax to be checked on the implementation level code. This problem is exacerbated by the fact that code generators typically rename model elements in the generated code, so that, for instance, the names of variables in the generated code are not known on the model level. Without knowing the names of the variables, certainly verification properties cannot be defined.

We present in this paper a method for specifying properties on high-level models that are then used in the verification of the generated, implementation level code. Properties are written in an intuitive way, directly on the model elements. As the model is translated into various intermediate forms and

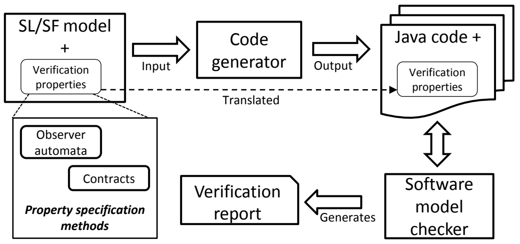


Fig. 1. Overview of framework. Verification properties can be specified using observer automata or contracts.

ultimately into executable code, the user defined properties are preserved and translated into implementation code and annotations that are checked by a software model checker. The translation is performed via a code generator that has been extended to handle the extra information. The results of the verification are then displayed to the user (in terms of the original high level model). While we focus on Matlab/Simulink, we believe that our method of defining properties on the model level that are checked against a generated implementation can be generalized and leveraged in other MBD tools as well. This approach makes property-based verification an integral part of the development workflow. Note that the framework enables run-time verification in addition to model checking.

The remainder of the paper is organized as follows. Section II gives an overview of our approach and background, including a description of the tool-suite. Section III provides details on how the user annotates Simulink models with properties. Section IV presents an end-to-end example. We compare our approach with related work in Section V and conclude in Section VI.

II. Overview and Background

An overview of our approach is depicted in Figure 1 and consists of the following steps: (1) a Simulink model is defined, (2) the model is annotated with properties to verify, (3) the code generator is invoked to produce executable code, (4) the software model checker is executed on the code and properties, (5) results about about the verification process are reported.

In this paper, we ignore the first and third points (defining the model and generating code) and instead concentrate on the other steps. For details on this part of the framework, please see [4]. In the work described in this paper we use a code generator that produces restricted form Java code. The main motivation for this choice of the target language was that the software model checker used can work with Java programs. The code generated by our toolchain is completely sequential and does not use dynamic memory (after initialization), hence it is suitable for embedded applications. The code is also object-oriented (an increasing trend in embedded software): subsystems are translated into Java classes that are instantiated at initialization time. Our code generator actually uses a retargetable back-end, such that either Java or C code can be produced from the same abstract syntax tree.

A. Property annotations

The second step in Figure 1 is annotating the Simulink model with properties to verify on the generated code. Since the development of model checking [5] in the early 1980s, a number of specification languages have been invented to formally define properties. Common ways of specifying these properties include regular expressions and temporal logic, such as LTL and CTL. However, the drawback to using temporal logics for property specification is their steep learning curve for industrial practitioners. Consequently, designers and developers will be less likely to use verification tools if they must devote large amounts of time to learning a specification language.

For this reason, we decided to take two approaches to property specification. The first uses the pattern-based system introduced in [6]. In that work, the authors studied a large body of existing property specifications and found that the majority of them were instances of a small set of parameterizable patterns: reusable solutions to recurring problems.

Patterns are entered into our system using a custom interface that we integrated directly into Simulink. After the parameters have been entered, our interface generates an observer automaton to represent am instance of that pattern. These observer automata are Stateflow subsystems inserted in the Simulink diagram that implement the logic of the specification described by the pattern. They contain input signals corresponding to the variables and events under observation, and the internal states that implement the logic of property. Full details can be found in Section III.

The second approach to property specification is based on *contracts* and is similar to the idea of programming by contract [7]. Programming by contract is a methodology for writing programs that use interface specifications on software components to define properties about their behavior. Typically, the specification on a component includes three elements: properties that must hold in order to use the component correctly (preconditions), properties that will hold when the component is finished executing (postconditions), and properties that must always be satisfied (invariants). We applied this idea of contracts to specifying properties for Simulink subsystems. On any subsystem, the user is allowed to write preconditions, postconditions and invariants that must be satisfied by that subsystem. During the code generation phase the contracts on various subsystems are translated into annotations on methods and classes implementing these subsystems in the generated code. A thorough description is given in Section III.

B. Software model checking

Our generated code is verified using Java Pathfinder (JPF) [8], a software model checker for Java. We chose JPF for two reasons. First, our toolsuite was already configured to generate Java code. Second, JPF provides libraries supporting a number of verification features especially useful in our toolsuite: code contracts, monitoring execution for exceptions and numerical problems, as well as symbolic execution.

The code contract feature of JPF permits annotations for preconditions, postconditions and invariants to be written on classes and methods. JPF monitors these conditions at runtime and reports any violations. This feature allows the preconditions, postconditions and invariants that are defined on the Simulink model elements to be translated to the generated code in a straightforward manner by the code generator.

The symbolic execution [9] feature of JPF allows us to perform state reachability and test case generation. The symbolic execution engine runs a program much like a normal program execution, but does not assign a concrete value to program input variables. Instead, input variables are left as symbolic values. When input variables are used in a branching condition, a constraint solver attempts to find values for the symbolic variables that will allow both branches of the condition to be taken. This idea is explained further in [9]. In this paper, we do not concentrate on the symbolic execution aspect.

III. Specification patterns and contracts

This section gives details on how properties are specified on the model level and then translated into generated code. We first describe the specification patterns, which can be attached to the model using a custom interface or from a supplied library. If the interface is used, a corresponding observer automata is automatically generated from the specifications. The interface can be used to insert basic properties, but to describe more complex properties, the observer automata can be compositionally defined using the supplied library. We also describe the details of how contracts are written on the model and then translated into annotations on the generated code.

A. Specification patterns

Property specification patterns describe commonly observed requirements in a generalized manner. They capture a particular aspect of a system's behavior as a sequence of state configurations. Note that the specifications can be state-based or event-based. In the discussion below we mention the state-based form, but the same approach applies to events as well.

To illustrate, consider the property that throughout a system's execution the value of a certain variable should always be greater than zero. There are two basic parts to this property that commonly occur. The first tells *when* the property should hold (in this case, at all times during execution), and the second tells *what* condition should be satisfied during this time (here, the variable should be greater than zero).

A property consists of precisely those two pieces: a *scope* and a *pattern*. The scope defines when a particular property should hold during program execution, and the pattern defines the conditions that must be satisfied. There are five basic kinds of scopes: *global* (the entire execution), *before* (execution up to a given state), *after* (execution after a state), *between* (execution from one state to another) and *until* (execution from one state even if the second never occurs).

There are three categories of patterns: *occurrence*, *order* and *compound*. The occurrence group contains the absence (never true), universality (always true), existence (true at least once) and bounded existence (true for a finite number of times) patterns. The order group contains the response (a state must be followed by another state) and the precedence (a state must be preceded by another state) patterns, and the compound group contains the chain precedence, and chain response patterns.

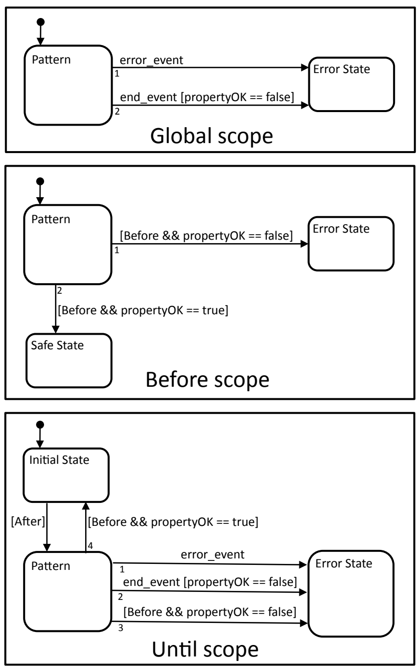

Pattern
error_event
1
Error State
end_event [propertyOK == false]
2
Global scope
Pattern
[Before && propertyOK == false]
1
Error State
2
[Before && propertyOK == true]
Safe State
Before scope
Initial State
[After]
[Before && propertyOK == true]
4
Pattern
error_event
1
Error State
end_event [propertyOK == false]
2
[Before && propertyOK == false]
3
Until scope

Fig. 2. Scope library.

Dwyer et al. [6] have shown how these scopes and patterns can be expressed in LTL, CTL, and other formalisms. However, the property specification patterns can also be easily expressed as parameterized observer automata, which is the approach we take. Note that many specifications can be added to a model and each one is translated into a separate automaton. Additionally, the definition of a simple interface allows the composition of the scope and pattern aspects of the specification, represented as two distinct automata templates. Furthermore, using the Stateflow language allows the observer automata to be created inside Simulink diagrams.

The Simulink model extended with the observer automata is then translated into the target language. Hence the generated, 'functional' code will be augmented with the code that implements the observer automata. Now the software model checker can monitor and verify the execution of the entire implementation, paying special attention to the error states and properties specified in observer automata. As specifications are translated into executable code, the distance between code-level monitoring and software model checking and model-level property specifications is reduced.

Figure 2 shows the automata for three of the five scopes. We now briefly describe each of these.

The automaton for the global scope is shown at the top of Figure 2. This scope indicates that a property should hold during the entire system execution. Initially, the state labeled "Pattern" is entered. There are two transitions from this state to the state labeled "Error State". The first is triggered by an event named "error_event". This event is generated by an enclosed property when that property has been violated. The second transition is triggered by an event named "end_event" and a guard condition requiring the boolean value "proper-

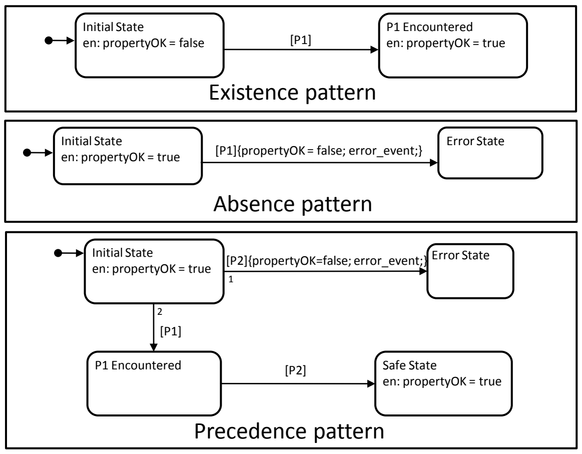


Fig. 3. Pattern library.

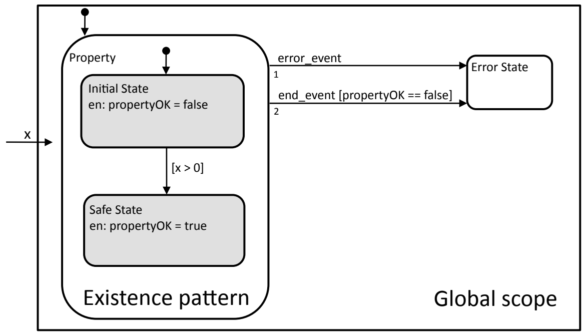


Fig. 4. Property describing that at some point, x should be greater than 0. Scope states are white and patterns states are shaded.

tyOK" to be false. The "end_event" is generated upon system termination and the "propertyOK" variable is set to false by the scope's enclosed property if that property is violated. That is, the second transition is taken if the system terminates and the property enclosed by this scope has been violated.

The automaton for the before scope is shown in the middle of Figure 2. This scope is used to express that a property should hold *before* some other condition is met. In the Figure, the event named "Before" is used to represent the condition. Initially, the "Pattern" state is entered. If "end_event" occurs (the system terminates) and the enclosed property has been violated ("propertyOK is false") then the first transition is taken and the "ErrorState" is entered. If the "Before" event occurs and "propertyOK" is false, the second transition is taken and "ErrorState" is entered. The state named "Safe State" is only entered if the "Before" event occurs *and* the enclosed property has not been violated ("propertyOK" is true).

The until scope captures the requirement that some condition should hold from one state to another even if the second condition never occurs, or stated differently, in between one condition and a second, even if the second condition never occurs. The bottom of Figure 2 shows the automaton for this scope. The two variables named "Before" and "After" are used to represent the two conditions in between which a property should hold. Upon entry, "Initial State" is entered. When the variable "After" becomes true, then the transition to the "Pattern" state is taken. While in this state, the automata is waiting for the property to happen *before* the second condition is satisfied. When the property is satisfied, the variable "propertyOK" becomes true. If before "propertyOK" becomes true either the "Before" condition becomes true or system execution ends ("end_event" occurs), the transition to "Error State" occurs and signals an error to the user. Otherwise, if "propertyOK" is true (the property is satisfied) and the second condition is also satisfied ("Before" is true), the transition back to "Initial State" is taken, and the cycle repeats.

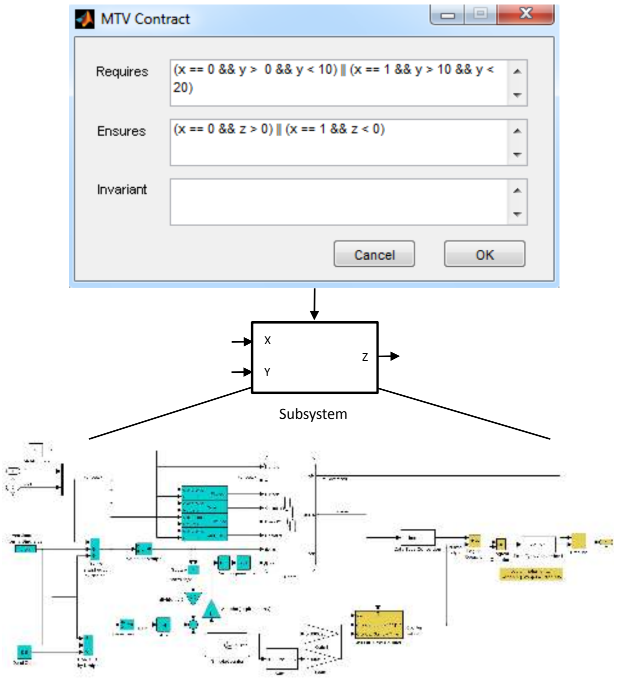
Figure 3 shows the automata for three of the patterns. At the top of the Figure is the automaton for the existence pattern. This pattern states that a condition (represented in the automaton by the boolean variable "P1") should occur during a specified scope. When the "Initial State" is entered, the "propertyOK" variable is set to false, indicating that the property is initially unsatisfied: P1 has not occurred. If "P1" does become true, then the transition to "P1 Encountered" is taken and "propertyOK" is set to true.

A simple pattern, absence, is shown in the middle portion of Figure 3. This pattern states that a condition (represented in the automaton by the boolean variable "P1") should not occur during a specified scope. When the "Initial State" is entered, the "propertyOK" variable is set to true, indicating that the property is initially satisfied: P1 has not occurred. If "P1" does become true, then the transition to "Error State" is taken,"propertyOK" is set to false and the "error_event" is emitted.

The automaton for the precedence pattern is at the bottom of Figure 3. This captures the property that some condition ("P2") must be preceded by another condition ("P1"). Note that in this automaton, the initial state sets the "propertyOK" variable to true: the property is initially satisfied. If "P2" is true before "P1", that is, the condition denoted by "P2" happens before the condition denoted by "P1" is met, then the transition to "Error State" is taken, "propertyOK" is set to false, and the "error_event" is emitted. Otherwise, the overall precedence pattern is satisfied.

Scopes and patterns are combined to form property specifications. Consider the example in Figure 4, which specifies the following property: at some point during system execution, the input variable "x" should be greater than 0. Stated differently, throughout the entire system execution (i.e., global scope), x should be greater than 0 at least once (i.e., existence property). To define this property, the existence pattern shown in Figure 3 is inserted into the "Pattern" state of the global scope shown in Figure 2. The difference is that the generic condition shown as "P1" in the basic existence pattern is replaced with the condition x $>$ 0. Note that the "propertyOK" variable is set by the pattern and its value is used by the scope.

Additionally, we developed a dedicated user interface that uses dialog forms for inputing property specifications. The dialogs capture both the kind of scope and pattern, as well as the parameters needed to instantiate and compose them. The user picks the scope and the pattern and enters the appropriate conditions. A composed automaton that composes an instance of both the scope and pattern is then automatically generated. An example using these dialog forms is described in Section


MTV Contract

Requires (x == 0 && y > 0 && y < 10) || (x == 1 && y > 10 && y < 20)

Ensures (x == 0 && z > 0) || (x == 1 && z < 0)

Invariant

Fig. 5. Contract example.

IV.

B. Contracts

The second method we use for describing verification properties is based on contracts. We extended Simulink with a custom interface that allows the user to annotate any subsystem with three additional items.

- Preconditions that the input signals to the subsystem must satisfy.
- Postconditions that the output signals of the subsystem must satisfy.
- Invariants that must always be satisfied by the subsystem.

Note that a subsystem translates into an executable function that is called by some scheduler, periodically. Hence, the above conditions and invariants can be checked during execution of that function block.

Figure 5 shows an example of specifying contracts on a subsystem block. The internal details of the subsystem are not important, but rather serve to show how our approach allows the complexities of certain elements to be ignored when writing specifications. The subsystem in Figure 5 has two inputs, x and y, and one output, z. Suppose the property we wish to check is the following: either x is equal to 0 and y is between 0 and 10, or x is equal to 1 and y is between 10 and 20. Suppose we also wish to check that if x is 0, then the output z is greater than 0, and if x is 1, then the output z is less than 0. These requirements are attached to the subsystem using the dialog box as shown at the top of Figure 5.

The contracts are added to the subsystem model as specially formatted descriptions (that are usually just unstructured text), using XML-like syntax. The code generator parses these descriptions, and if they are syntactically correct, it constructs the properly formatted strings (with variable names rewritten into their 'code' equivalent) that are suitable for the software model checker.

A Java implementation of the subsystem in Figure 5 that is very similar to the code produced by our code generator is shown in Listing 1. Note that in the contract, the inputs and outputs of the subsystem are referred to by their name in the model. This is an important part of our approach: the user always refers to the model elements as they are written in the model. No knowledge of the code generation process is needed to write specifications. The contract specified in the model is generated in the Java code as annotations that automatically reference the correct variable names. These annotations are used by the software model checker to monitor the code execution.

Listing 1. Java implementation of the subsystem in Figure 5.
```
public class Subsystem15 {
  private int value1 = 0;
  private int value2 = 0;

  @Requires(``(x13 == 0 && y25 >  0 && y25 < 10) ||
              (x13 == 1 && y25 > 10 && y25 < 20)'')
  @Ensures( ``(x13 == 0 && z65 > 0) ||
              (x13 == 1 && z65 < 0)'')
  public void Main23(int x13, int y25, int[] z65) {
    value1 = x13;
    value2 = y25;
    ...
    // Code implementing subsystem logic
    ...
  }
}
```

IV. Example

This section shows how our framework can be applied to realistic models. The example we use is the Apollo Lunar Module digital autopilot model, which is included with the Matlab/Simulink distribution as an example. The full model includes a dynamic model of the plant: the Apollo Lunar Module, as well as a model of the Reaction Jet Controller (RJC) – we focused on the embedded controller. A very high-level view is shown in Figure 6. The RJC receives attitude measurements and desired attitude values, and generates control signals to activate yaw, pitch and roll thrusters.

A. Step 1: Define Property

The "Yaw_Jets" output of the RJC block is a value from the set -2, 0, 2, which indicates that the yaw thruster should have a negative thrust, no thrust or a positive thrust, respectively. Suppose we wish to verify the property that the "Yaw_Jets" output can never go directly from -2 to 2 or directly from 2 to -2: at least one output of 0 must always be found in-between. Section III showed how a property like this could be built manually using automata. Using the scope and pattern automata as building blocks, one could define this property directly in Stateflow.

As mentioned above, we have also developed a custom extension to the Simulink environment that allows properties to be entered in an easier way using dialog forms. These dialogs decompose the patterns detailed in Section III-A: the user selects a pattern, enters a scope and a property and the equivalent automata is generated, including input ports. Our first task is to decide which pattern we need to implement the property that the “Yaw_Jets” can never go directly from -2 to 2 or directly from 2 to -2. Part of the property states that we do not want the value of “Yaw_Jets” to be -2 during a certain scope. The absence pattern fits this requirement, as it checks to see that some condition never occurs.

The dialog form for the absence pattern is shown in Figure 7. This dialog guides the user through the process of defining a property. After defining the condition that should never hold (Command == -2), we define the scope during which this condition should hold. In this example, we never want Command to go directly from 2 to -2, so the condition that Command should never be -2 should hold after Command is equal to 2 and before Command is equal to 0. The property that Command should never go directly from -2 to 2 is defined in an analogous way using the absence pattern dialog.

B. Step 2: Connect generated automata

After entering the parameters in the dialog form, the observer automaton monitoring the property is generated, as shown in Figure 8. The states representing the scope portion of the property are white, and the states representing the pattern are shaded. The transition from the initial state is taken when Command is 2, at which point we are “in scope” and want to verify the absence of the condition that Command is -2 before it is 0. If the value of Command is -2 before it is 0, the transition to the inner error state is taken, which sets the “propertyOK” variable to false and emits the “error_event”. When “error_event” is emitted, the outer transition to the error state is taken and the automaton remains in this state. Note that while the automaton is in scope, system termination (the “end_event”) will not cause the property to be violated as long as Command has not been set to -2. The input parameter for command is automatically generated, so the user must connect the “Yaw_Jets” signal to the automaton so that it can be monitored. In Figure 6, the “Command Constraint” and “Command Constraint2” automata have already been connected to the “Yaw_Jets” signal.

C. Step 3: Verification with JPF

The final step is to invoke the code generator and use JPF to verify our properties. There are two ways JPF can check the code for property violations. The first uses concrete inputs provided by the user. If this is done, JPF will perform a concrete system execution using those inputs and report any property violations in the form of stack traces. The second way JPF can check for property violations uses the symbolic execution module. In this case, JPF will try to determine inputs to the system that will cause properties to be violated. With either method, property violations can be reported to the user in the form of a stack trace showing the sequence of method invocations that led to an error state.

V. Related work

In more traditional forms of software development, verification is done in one of two ways. Either an abstract model of the

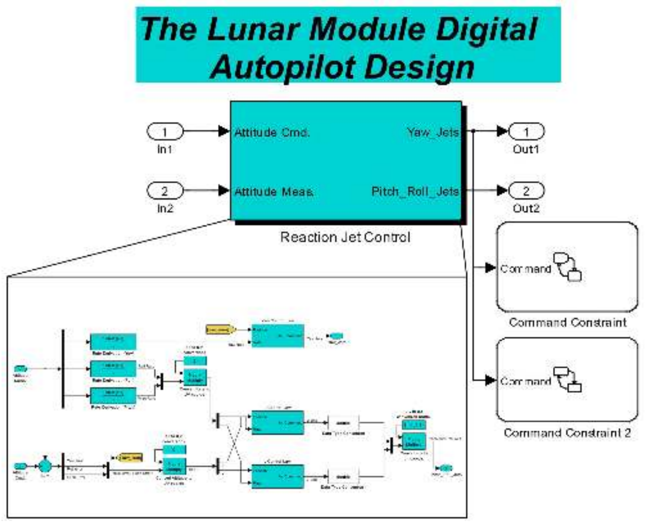
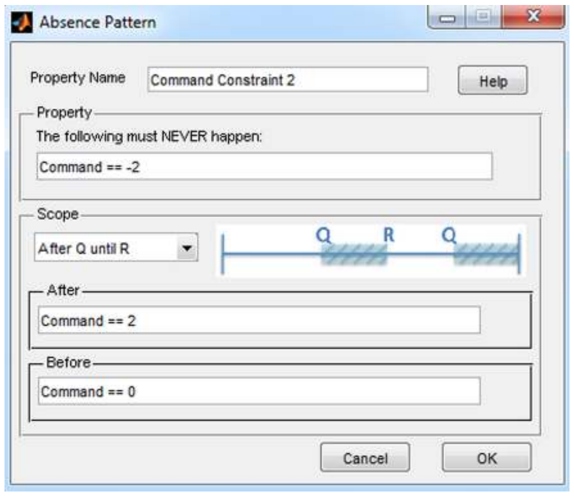


Fig. 6. High-level view of the Apollo Autopilot. The Command Constraint automaton was automatically generated using the property defined in Figure 7. The second automaton was also automatically generated.


Absence Pattern

Property Name: Command Constraint 2

Property

The following must NEVER happen:

Command == -2

Scope

After Q until R

After

Command == 2

Before

Command == 0

Fig. 7. Property dialog. The property says that after the input variable “Command” becomes 2, it should never be equal to -2 before returning to 0.

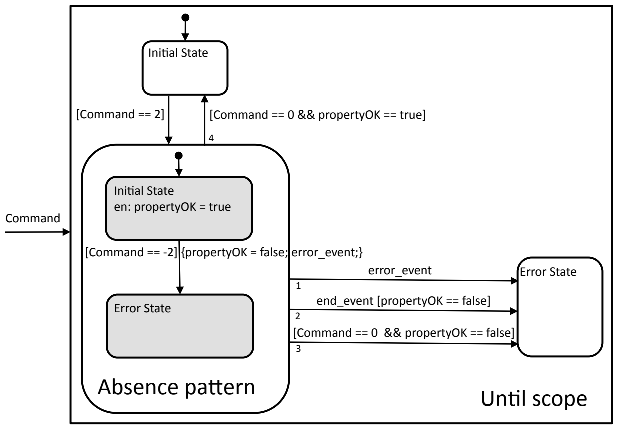

Initial State
[Command == 2]
[Command == 0 && propertyOK == true]
4
Initial State
en: propertyOK = true
Command
[Command == -2] {propertyOK = false; error_event;}
error_event
1
end_event [propertyOK == false]
2
[Command == 0 && propertyOK == false]
3
Error State
Error State
Absence pattern
Until scope

Fig. 8. Generated observer automaton implementing the property specified in Figure 7. Scope states are white and pattern states are shaded.

software is created and verified, or the executable code itself is verified. [10] discusses the ongoing trend towards placing the verification efforts directly on the executable code rather than on models. In MBD, however, one intentionally begins with models and gradually refines them until they are synthesized into the executable code, and ideally both artifacts can be verified. Our approach eases the burden of both specifying and checking properties on code generated during the MBD process.

A number of tools are available for verifying Simulink/Stateflow models. Simulink Design Verifier [11] and Reactis [12] are commercial tools for checking model properties. [13] describes an approach that is based on hybrid automata: models are translated from Simulink to a hybrid automata formalism and existing techniques for checking hybrid automata can then be applied. Our approach is complimentary to these methods and ensures the properties proved by these tools also hold for the generated code.

Our approach to specifying properties through patterns is based on the work of Dwyer et al. in [6]. The pattern library described there contains a general description along with mappings into multiple formalisms, including LTL, CTL and quantified regular expressions. Our implementation uses a dialog forms to chose and configure simple patterns from which observer automata are generated, and includes a library of observer automata for individual scopes and properties from which more complex patterns can be defined.

Runtime monitoring [14] is a related area in which formally specified properties are typically translated into executable code that is used to check program properties during program execution. Recent work in this area includes optimizing such monitors through static analysis techniques [15]. Our approach translates properties specified using observer automata into executable code that is checked by a software model checker and translates contracts on model elements into annotations that are used by the model checker.

VI. Conclusion

Checking model level properties on implementation code is a useful approach for practical model-driven development. In this paper, we have shown how relevant properties can be specified on the model level and then translated into implementation code that can be verified with a software model checker. Our approach is a pragmatic realization of the work described in [6], in the context of the Simulink/Stateflow environment. We have shown how the specification patterns can be instantiated from observer automata templates for scopes and properties and how subsystem blocks can be annotated with pre-, post-conditions, and invariants that are monitored by the software model checker. We have shown the use of the approach on a realistic example.

Our approach allows two ways for specification: contracts and property specifications based on patterns (that are translated into observer automata). For designers of embedded systems two extensions would be very useful: (1) specifying real-time properties, and (2) dealing with concurrency. Translated Simulink subsystems are typically executed periodically, with a fixed rate. Timing properties can be related to a single execution run (i.e. the worst-case execution time of a function block), as well as the temporal properties of the system over multiple execution runs (e.g. the system reacts to a triggering event within a bounded number of execution runs). Translated Simulink subsystems are also completely sequential; they are usually translated to functions in an implementation language. In order to run them on an execution platform, they have to be embedded into OS processes, and their communication and synchronization implemented outside of Simulink. Hence, we need to model these embeddings, and how the threads containing the function blocks communicate and synchronize. These topics are the subject of on-going research.

VII. Acknowledgments

The work described in this paper has been supported by NASA under Cooperative Agreement NNX09AV58A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration. The authors would also like to thank Michael Whalen for valuable discussions and feedback.

References

[1] Á. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *IEEE Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[2] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[3] G. J. Holzmann and R. Joshi, "Model-driven software verification," in *SPIN*, 2004, pp. 76–91.

[4] J. Porter, P. Völgyesi, N. Kottenstette, H. Nine, G. Karsai, and J. Sztipanovits, "An experimental model-based rapid prototyping environment for high-confidence embedded software," in *IEEE International Workshop on Rapid System Prototyping*, 2009, pp. 3–10.

[5] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking*, 2008, pp. 1–26.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, 1999, pp. 411–420.

[7] B. Meyer, *Object-Oriented Software Construction, 1st editon*. Prentice-Hall, 1988.

[8] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering (ASE)*, vol. 10, no. 2, pp. 203–232, 2003.

[9] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[10] G. J. Holzmann, "Trends in software verification," in *FME*, 2003, pp. 40–50.

[11] "Mathworks Inc. Simulink Design Verifier," http://www.mathworks.com/products/sldesignverifier/.

[12] "Reactive Systems, Inc." http://www.reactive-systems.com/.

[13] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar, "Symbolic analysis for improving simulation coverage of simulink/stateflow models," in *Proceedings of the 8th ACM international conference on Embedded software*, ser. EMSOFT '08. New York, NY, USA: ACM, 2008, pp. 89–98.

[14] S. Sankar and M. Mandal, "Concurrent runtime monitoring of formally specified programs," *IEEE Computer*, vol. 26, no. 3, pp. 32–41, 1993.

[15] E. Bodden, L. J. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," in *ECOOP*, 2007, pp. 525–549.