Industrial applicability of advanced model/code-based V&V techniques for verifying program properties in embedded applications

C Hote, M Lalo, P Munier, D Pilaud

Industrial applicability of advanced model/code-based V&V techniques for verifying program properties in embedded applications

C. Hote[1], M. Lalo[2], P. Munier[3], D. Pilaud[4]

1: The MathWorks
2: The MathWorks
3: The MathWorks
4: Casserousse Consulting

Abstract:
The constant and significant increase of computer power at low cost and many recent major technological advances in program properties verification techniques show that designers and developers can now efficiently and practically use proving techniques either at model or source code level.

Those new V&V techniques convey major benefits to industrial sectors where software quality is at stake including early detection of errors (at specification, design and coding levels), and proof of absence of errors. Those techniques strengthen software application development process and minimize the likelihood of errors found either late or released in the field.

The paper describes several advanced techniques for statically verifying dynamic properties of programs including logical, functional, run-time errors, how those techniques fit within current development processes and how they may be used for monitoring software quality over time. The paper primarily applies to the development of embedded applications and demonstrates how the combined usage of techniques such as model-checking and abstract interpretation effectively handles industrial problems today.

Keywords: model checking, abstract interpretation, test, verification and validation

1 Introduction

The last three decades have been very beneficial to the embedded software engineering standpoint and many software tools today help engineers to build larger and more complex software applications. Out of many examples, design tools introduce graphical formalisms (block diagram, state chart) from which early simulations, rapid-prototyping, and early specification error detection can be achieved without writing a single line of source code. In the same vein, auto-code generation was first introduced in the late 80' for supporting the design of embedded avionic systems, nuclear systems, and railways controllers. Later on in the mid 90s, those techniques spread out towards automotive sectors and are now used in many industrial sectors including consumer electronics, and medical devices.
In the same time, the verification and validation techniques did not improve at the same pace and as a result, companies developing embedded software struggle in the testing phases of the software development process or have to deal with annoying software quality issues including product recalls, delay in deliveries or mission losses.
Based on recent advances in static verification of program properties, a new hope is unveiled.
The first section of the paper defines and positions several proving and testing activities available in the embedded software domain. The second section gives examples of properties which can be proven at the model and/or at the code level. Finally, this paper presents future directions mixing these different approaches.

2 Definition and problematic

2.1 From error detection to property proving

There are many definitions to validation and verification activities although most of them lead to two fundamental objectives;
1 - Maximizing the number of errors found during the software development process
2 – Minimizing the number or errors released in final software applications.

For many decades, testing techniques have been focusing on the first objective based on the assumption that item #1 infers a good confidence level in item #2 provided enough time has been spent in testing, and provided item #2 is, in general, not a solvable problem.

As a result, many testing techniques improved for easing the detection of more bugs and very few focused on measuring the amount of remaining bugs in software applications. The first domain refers to bug detection techniques while the latter refers to property proving, generally, an NP-hard problem.

Out of many, once could mention some interesting properties embedded systems should comply with:

- Functional property or compliance with specifications: the software must compute some known outputs with some known inputs, within a defined sequence. In other words the software implementation should behave in compliance with its specifications.
- Timing property or compliance with operational requirements: the software must provide with the right answers within a known time frame and using a finite amount of memory.
- Absence of run-time errors in programs: the software must react deterministically, without any unpredictable behaviors, and be reliable to any execution and to any possible input values.

The following paragraphs discuss several techniques that may be used for achieving those objectives and depending when they apply.

2.2 Dynamic testing

Dynamic testing consists in setting test objectives, in generating test cases, and then in verifying the results obtained from the execution of the tested software component match the test objectives.
Besides very few examples (table truth, code coverage), dynamic testing cannot be exhaustive and cannot bring a definitive proof a test objective has been achieved. In that regard, dynamic testing is a bug detection technique more than a proving technique.

If you want to prove that a property is true, you need to rely on mathematical techniques. If you want to prove that a property is false, you can bring a counter example. Nevertheless, if no counter example can be brought, it does not constitute any proof.
In theory, if one is able to run an exhaustive testing campaign and check for each test that the oracle is true, this would constitute a proof that the property is true. For real-time embedded application however, this is out of reach for almost any system. Testing is great to find errors, and to gain confidence that the property is true. Though, it's never a proof.

2.3 Static analysis

When the focus is on bringing counter examples, the goal is to only warn the user if a counter example can be found. This makes these static techniques shift from "being sound and complete" to partial, in order not to swamp the user with false positives. The drawback is that these techniques miss errors (false negative) and can not prove the absence of errors in case nothing is reported. But these techniques work pretty well on the early detection of errors.

The only way to bring a proof is to use mathematical approaches of software, hence require static techniques. The next section will detail one view of different techniques.

2.4 Model and Code-based verification

Although the scope of this document is not to describe which formal techniques exist and what their application domain is, we can restrict this paper to two main techniques:

- Model-checking and their derivative (symbolic model checking, NP prover approach is suitable for solving problems at the level of model.
- Abstract interpretation is working on a semantically strong language, most of the time a low level development language. This technique is suitable for understanding the data flow of a program.

"Automatic test case generation" can also be a mean to look for counter examples. . This is a technique can be derivate from the first two, and will not be detailed in this paper

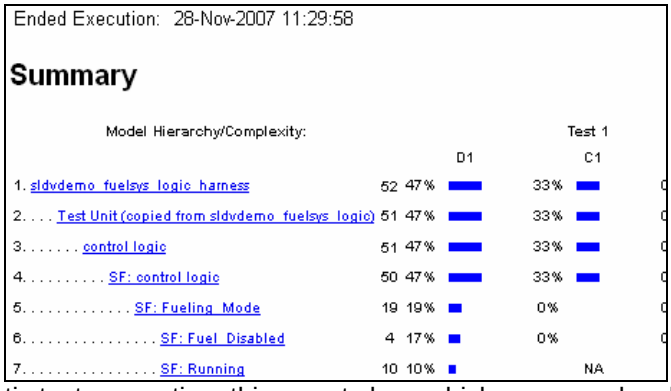
These techniques are mature enough for being available in commercial tools such as Mathworks tools or Esterel technologies tools. This will be detailed in the upcoming section with examples on which types or properties they

can handle. For model checking, we'll provide examples using Simulink® Design Verifier. For abstract interpretation, we'll provide examples using PolySpace™ Model Link SL.

3 Examples of properties and how they can be proven

3.1 Prove model coverage

The first demo shows the ability to prove 100% coverage on a finite state machine model. The oracle for this model is: 100% coverage can be reached. The demo is "sldvdemo_fuelsys_logic" using Matlab 07b. The demo has been stopped before reaching 100% of the objectives


Ended Execution: 28-Nov-2007 11:29:58

Summary

Model Hierarchy/Complexity:		D1		Test 1 C1	
1. sldvdemo_fuelsys_logic_harness	52	47%		33%	
2. . . . Test Unit (copied from sldvdemo_fuelsys_logic)	51	47%		33%	
3. . . . . control logic	51	47%		33%	
4. . . . . SF: control logic	50	47%		33%	
5. . . . . SF: Fueling_Mode	19	19%		0%	
6. . . . . SF: Fuel_Disabled	4	17%		0%	
7. . . . . SF: Running	10	10%		NA	

Figure 1. After automatic test generation, this report show which coverage has been reached for each state, and each transition. The oracle was defined for 100% state-transition coverage, not for a decision, condition or MC/DC coverage

3.2 Temporal logic oracle

This second demo shows the ability to prove a functional property (based on temporal logic) of a model. The oracle is the only way to have 6 outputs equal to 1 in a row is that the input is equal to 2. The demo is "sldvdemo_debounce_testobjblks" using Matlab 07b

The objective was to prove that if an output is equal to 1, than the previous output was greater than 1, as shown in the figure below

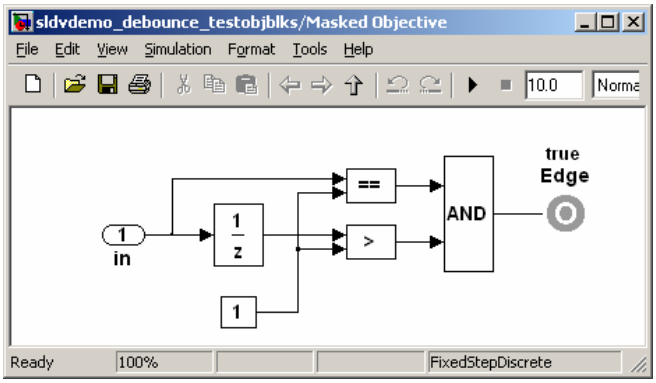


Figure 2. Oracle showing a functional property based on temporal logic.

3.3 While-loop based algorithms

For abstract interpretation, C language is an appropriate language to show what it can prove at that level as it is close enough to a run-time to be dealing with timing issues, run-time errors and dead code.

This example shows how abstract semantic allows converging on a dataflow whose previous values are re-used for the next output.

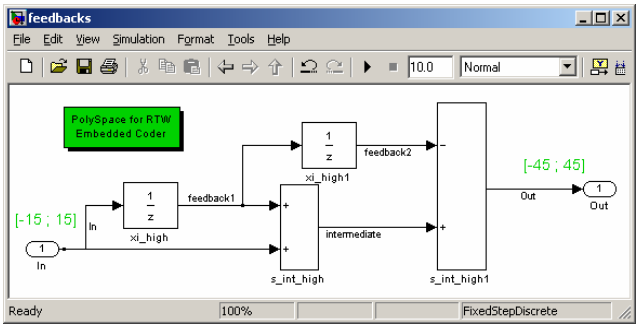


Figure 3. The figure above shows how data ranges can be propagated throughout a code called within a while loop and invoking 2 steps memorization of variables.

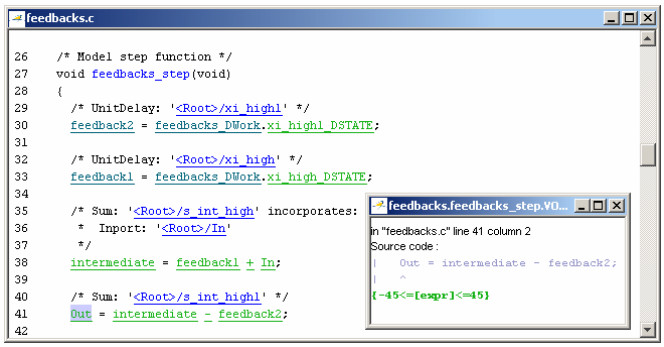


Figure 4. This is the same figure at the generated code level

It must be noticed here that working at the generated code level enable tools to link back to the model, which consists in terms of process in a functionality overlap. These findings can be done both at the model and the code level, providing the model has a strong syntax and semantic.

3.4 Dead code

Here is an example on the same demo than Simulink® Design Verifier. We'll see that the types of errors it finds are different, as they are more related to the dataflow than the temporal logic.

Instead of having the default full range entries which are not functionally realistic, the code verification was performed on bounded entries. Because of the fixed point implementation of the model, the real C bounded entries are different. They map the implementation ranges in the generated C code.

Throttle	0	120
Enginespeed	0	1000
EGO	0	1536
MAP	0	256

Using the rtwdemo_fuelsys_fixpt of MATLAB release 07b, abstract interpretation can show a portion of dead code which indicates that the state Fuelsys/Overspeed can not be reached. Knowing that this model is a closed-loop model, this means that the system is stable and can not run in over-speed. If the functional property is that the engine should never run in over-speed, this property can be proved using abstract interpretation. If this portion of dead code is not intended, this can be a highlight of a functional error.

This is an example of a functional property which can be proved at the code level. This is possible if the oracle is expressed in a binary way. For instance, the following oracle can be proven true or false by abstract interpretation: "the value of a given variable should always be within a range or never within this range (here dead code)".

3.5 Run-time errors

Another example of errors which is half a functional property, half a run-time property, can be a run-time property whose cause will most probably be non-compliance to a functional property.

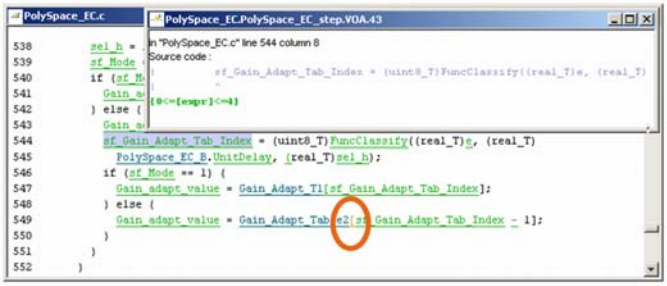


Figure 5. The figure above shows a possible out of bound array index in the source code.

We will not discuss here whether the cause of the error is a typo of the table whose index should start with 1 (and therefore get rid of the array[index – 1] in the generated code), of if the index should not be between 0 and 4, resulting in a violation of a functional property for instance. If the table represents arbitrary calibrations, this type or errors might be detected at a very late stage of the overall development process, if not after the generated code goes into production.

3.6 Proving code Correctness

Another property which can be proven with abstract semantic is the robustness and reliability optimizations done in the code. For instance, the question about overflows can be answered comparing ranges of the destination type with the actual computed ranges.

Thanks to its sound approach, abstract semantic also allow to prove the code correctness on the same topic

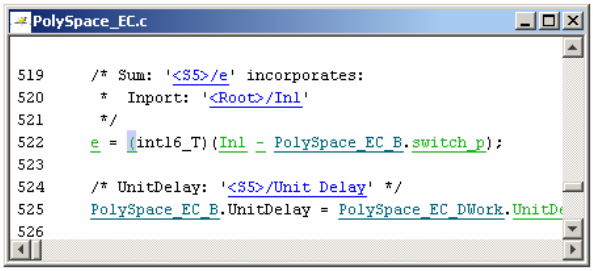


Figure 6. This figure proves that no overflow will ever happen at this line

By looking closer at this line, abstract interpretation can give you more details about why this oracle is always true

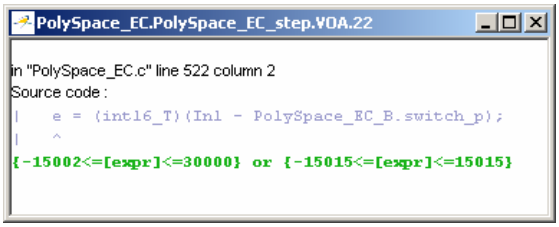


Figure 7. This figure show data ranges known at this stage of the program, here a union of data-ranges.

3.7 Code generation optimization

Code optimizations can also be detected and traced back to the model as well.

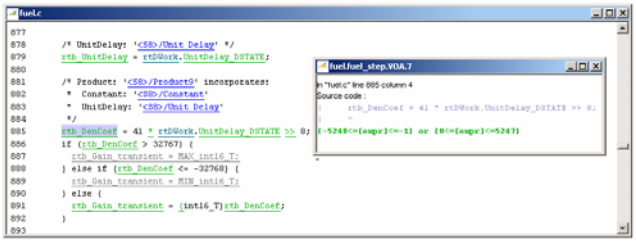


Figure 8. This figure shows dead code (in grey) of saturation because data ranges are small enough to fit into a 16 bit signed type without saturation.

The saturation above is indicated as grey because the data ranges saved into the variable are much smaller than the destination type. A typical application can be code generation optimization.

4 Workflow considerations

4.1 Introduction

These proving techniques have made significant progress and are used operationally in the industry. They are used within several types of process

- Software development process: in this context, the final goal is to reduce the testing efforts required to have confidence in the final code and keep (if not increase) the level of confidence of the final delivered code
- For Quality assessment, the goal is to perform final audit of source code and measure the final code correctness to reduce the risk of leaving errors in the source code.
- With Code certification, the goal is to prove to certification authorities that everything has been done to reduce the risk of leaving errors in the field.
- When proving can not reach a 100%, the techniques can be used to generate defensive behavior within the final embedded code, similar to what exception handling does in some languages, but focused on very few non-proven properties.

4.2 Development

4.2.1 Adopt a development process which is component based only

Some development processes are component based and define their Verification and Validation activities at the component level only, without having some robustness test at the system level. This is only possible under certain conditions.

- First, the development process must define all the component dependencies and their impacts toward the entire application. If one component is modified and its outputs vary within another range than previously, the process must clearly identify which other components should be re-verified.
- On top of this impact analysis, the process must define for each component all its interfaces: with inputs and outputs and toward unknown parameters at the development stage such as calibrations. For each parameter, some data ranges must be specified: they can be either restricted to functional values only or/and may also contain worst case ranges.

For this kind of process, the idea is to reduce the amount of work in integration testing, as the total size of the code represents several hundred thousand lines of code, with more thousands points of calibrations, which might change the behavior of the whole application, even after delivery. For the purpose of code correctness and software reliability, testing is out of reach to prove anything.

So abstract interpretation can solve that by successfully verify – for each component – that the outputs match the specification, providing no run-time error happens and therefore preserve data integrity. This process can be automated by using data-ranges coming from specification documents, such as databases associating ranges for each software component, including calibrations.

4.2.2 Reduce iteration in Software development cycle

When the software development process is not particularly component oriented, errors can be equally spread over the whole development cycle. A run-time error for instance, such as an overflow, may be local to a function which performs computation on its local data, or may happen globally because of the effects of another software component. This requires abstract interpretation to work both at the component level and at the application level. The type of review, however, will be different for each step. In both cases, doing code verification functional test allows to find defect at early stages of each project, and therefore reduce testing activities in the field, or on test benches.

Other techniques may highlight where errors lie, but cannot guarantee that other operations are safe. That usually forces the user to proceed with redundant tests. Abstract Interpretation flags all operations that will never experience run-time errors—no matter the operating conditions. This contributes to reducing testing efforts and thus reduces the number of iterations in a development cycle.

4.3 Quality Assurance

4.3.1 Quality assessment for OEM

Within the relationships of suppliers and integrators, source code has always been an area of doubt. Traditional quality metrics, derived from ISO 9126 are in place for almost two decades, but can only rate the quality of the process more than the reliability of the final product. They work on the assumption that doing things right will produce the right software. Abstract interpretation, on the other hand, does not measure the quality of the development process, but only focuses on proving the code correctness of the final product. This is crucial for OEM which owns the responsibility of the final software reliability. Assessing a source code at his final stage of production can either be done by sampling files, classes, packages from the entire application or by verifying the whole source code. It provides a direct measure of the source code, and indicates areas where errors can happen, no matter how these lines were produced. This is ideal in situations where the results count more than the process to produce them.

4.3.2 Collaboration between OEM and supplier: Process improvement

On the margin of quality assessment, having a supplier using this technology in his tool chain is a great value for an OEM. Knowing how and when PolySpace products for C/C++ were used during coding gives great confidence in code reliability. It provides a guarantee that software robustness and reliability are ensured in the most efficient way possible. This is particularly appealing for suppliers companies, which see in abstract interpretation a mean to measure and improve source code reliability and not only a final code assessment tool.

4.4 Certification

Certification is about providing evidence that the development process has been done according to what has been defined. In that context, proving that a code verification tool can not miss one error is essential for certification authorities. Abstract interpretation is one of them by its exhaustive and sound approach. Every data approximation is done in a conservative way, in order to ensure that no value was missed. This way, it's possible to show certification authorities reports from the software validation efforts

- Showing all flagged sections of the code which cannot return a software fault
- Demonstrating for the operations that can return a software fault that they have been reviewed.

For the first task, the certification authority can audit the tool vendor creating the code verification tool. Evidences should be provided that the underlying technology and his development process allow creating a code verification tool which can not miss errors. The second task can be achieved by providing evidence of code review operated on the non-proven points.

4.5 Generating defensive code based on non-proven properties

When it is impossible to demonstrate all the properties by these techniques: For example, code verification cannot prove 100% of run time errors if the C program is not driven by coding rules. It is therefore necessary to learn how to use the test and formal approaches together.

One idea would be to generate code or to create a model accordingly to the non proven properties. This would of course impact the associated development cycle. From a classic Design-Generate code-verify, some new steps could be introduced

- Design at the model level
- Generate code
- Formal verification at both model and code level.
- (new) From the non-proven properties, produce both a model and the associated code
- (new) Test of the of these properties, and decision to incorporate (or not) defensive code in the embedded system

Let's consider two examples:

- A model can be enhanced with non-proven properties. This can include the error-case behavior of the property. The model will thus be designed to behave properly in case the error happens. This will require a change in the development process: once the design is complete, the user will need to check which properties can be proven. On the remaining, he will have to choose whether the implementation in the design of this defensive behavior is appropriate or not. Code generation will follow, and produce the right defensive code.

- The code can be enhanced with non-proven properties. For run-time errors, it is possible to include defensive code for non-proven checks. The principle is similar to exception handling, but focused on specific run-time checks in the code: the benefit is to have limited overhead in the execution time and the size of the code. This enables the user to embed the code in the final real-time system.

5 Conclusion

In order to generate model and code for these non-proven properties, it's required to maximize the number of proven properties.
In the first part of this paper, we have seen which area checking can be successfully applied to: proving temporal and Boolean properties, proving model coverage, and finding counter example to functional properties.
In the second part of this document, we have seen which areas abstract interpretation can be successfully used: proving the absence of runtime errors, solving while-loop algorithms, and dealing with data related errors.
Combining these techniques together would help covering all areas and all type or errors which can be introduced in a development cycle. With proof ratios over 90% for both techniques can be reached separately, it can be anticipated that combined techniques will over a much higher proof ratio.
In a third part, we have discussed the possibility to combine these techniques with testing, thus enabling to embed C code containing focused defensive programming on non-proven properties. This can be an alternative to having 100% proof ratios.
The next question is now how fast industrial will modify their development process to adopt these techniques.