# A Visual Studio Plug-In for CProver

Mohamed Nassim Seghir and Daniel Kroening
Computer Science Department, University of Oxford, UK
{seghir,kroening@cs.ox.ac.uk}

***Abstract*—In recent years, automatic software verification has emerged as a complementary approach to program testing for enhancing software quality. Finding bugs is the ultimate aim of software verification tools. How do we best support the programmer who has to diagnose and understand those bugs? Unfortunately, most of the existing tools do not offer enough support for error diagnosis. We have developed a plug-in which implements a graphical user interface for the CProver tools within the Visual Studio IDE. Our plug-in enables visual debugging and error trace simulating within C programs as well as co-debugging C programs in tandem with wave-form views of hardware designs. Another feature of our plug-in is background verification. Each time a program source is saved, the verification process is silently triggered in background. If an error is found, its location is highlighted in the program. The user interacts directly with the program source to obtain information about the error.**

***Index Terms*—Software Model Checking, Program Verification, Co-Verification, Visual Studio, Plug-in.**

## I. Introduction

The software engineering discipline has gained a high degree of maturity. Many tools have been developed to support the different phases of the software development cycle. Program testing is widely used to check conformance between software requirements and their implementation. While test can show the presence of bugs, it cannot prove their absence. In recent years, formal methods have emerged as a complementary approach to testing, and enable rigorous reasoning about the software. This is essential for safety-critical software, where cost of errors is high.

Model checking is an automatic technique for system verification based on exhaustive exploration of all system states. Thus, it can guarantee the absence of bugs with respect to a given formal specification. Model checking has been successful in hardware verification. The development of model checkers for software verification is a formidable research challenge, which has received a lot of attention in the last years. Many tools, e.g., SLAM [6], BLAST [21], MAGIC [9], SATABS [13] and TERMINATOR [14], have been developed and successfully applied to real-world programs. Finding bugs is an ultimate aim of these tools, and the question that follows is: how are these bugs fixed? Deep understanding of the causes of the bugs found is required for proper repair.

Graphical support for software verification tools can be an invaluable aid for the diagnosis and comprehsion of bug causes. It can also increase the degree of acceptance of these tools within the developer community. Ideally, the verification tool should be part of the development, or at least must be seamlessly invokable from that environment, in order to permit the repair of bugs as early as possible. To this end, we have developed a plug-in which implements a graphical user interface for the CProver tools within the Visual Studio IDE [1]. The CProver suite is a set of tools for software and hardware verification as well as software/hardware co-verification [2].

Our plug-in offers support for visual debugging and error trace simulation. It allows to inspect states of a given program while stepping through the statements of an error trace. It also permits to co-debug C programs in tandem with wave form views of a hardware design, enabling understanding of the interaction of low-level software with hardware components. An additional feature of our plug-in is the background verification option. Each time a source code file is saved, the verification process is silently triggered in background. If an error is found, its location is highlighted in the program. The user can directly interact with the highlighted portion of source code to obtain information about the error.

The remainder of the paper is organized as follows: in Section II, we provide an overview of the CProver tools integrated via our plug-in. In Section III, we illustrate the integration of the different tools into the GUI by describing their invocation via the IDE. Finally, Section IV summarizes the technologies used for the implementation of the plug-in.

## II. CProver: An Overview

The CProver suite is a set of tools for the formal verification of software and hardware. All tools are fully automatic and most of them are based on the model checking approach. In what follows, we introduce some of the tools that are supported by our plug-in.

### A. CBMC

CBMC is a Bounded Model Checker for ANSI-C and C++ programs [12, 3]. It allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check ANSI-C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure. While CBMC is aimed at embedded software, it also supports dynamic memory allocation using malloc and new.

Supported by the Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/H017585/1, the EU FP7 STREP PINCETTE, the ARTEMIS VeTeSS project, and ERC project 280053.

### B. SATABS

SATABS is a verification tool for ANSI-C and C++ programs [13, 4]. SATABS transforms a C/C++ program into a Boolean program, which is an abstraction of the original program in order to handle large amounts of code. The Boolean program is then passed to a Model Checker. To build the Boolean program, SATABS uses *Predicate Abstraction* [20], which is best suited for lightweight properties such as array bounds (buffer overflows), pointer safety, exceptions and control-flow oriented user-specified assertions.

### C. HW-CBMC

HW-CBMC is a tool to verify the consistency between a hardware design in Verilog and a reference description written in ANSI-C [11, 5]. This technique is known as *co-verification*. A common design approach employed by many companies is to first write a quick prototype that behaves like the planned circuit in a language like ANSI-C. This program is then used for extensive testing and debugging, in particular of any embedded software that will later on be shipped with the circuit. Due to market constraints, companies aim to sell the chip as soon as possible, i.e., shortly after the HDL implementation is designed. There is usually little time for additional debugging and testing of the HDL implementation. Thus, an alternative to this consists of verifying the consistency of the HDL implementation using the ANSI-C model as a reference.

## III. Integration of CProver with Visual Studio

Our plug-in is integrated as an extension for the Visual Studio IDE. When the IDE is launched the new menu CProver appears (Figure 1). This menu offers different commands, among which the option of configuring the plugin-in, including the selection of the verification tool (Figure 2). In what follows, we describe the integration of the different tools with the Visual Studio IDE and the interaction with these tools through the GUI.

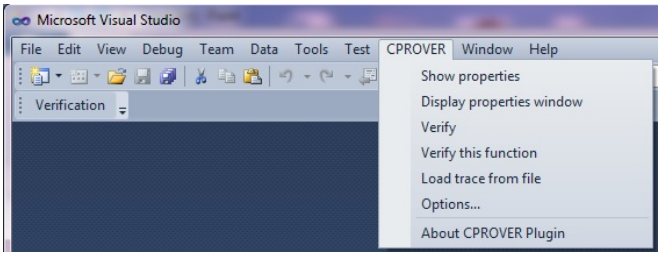


Fig. 1. CProver menu within the Visual Studio IDE

### A. Software Verification

Our plug-in supports the verification of C and C++ programs via CBMC and SATABS. These two tools are able to check whether a property is always valid for any execution of the program. Both CBMC and SATABS use assertions to specify program properties. Assertions are claims about the state of the program when the program reaches a particular program location. Assertions are often written by the programmer by means of the assert macro.

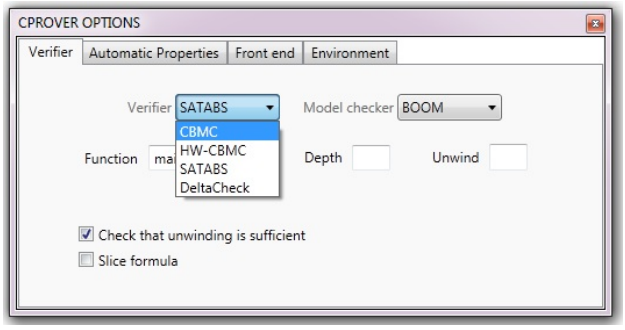


Fig. 2. Selection of the verification tool

In addition to the assertions written by the programmer, assertions for specific properties can also be generated automatically by CBMC and SATABS, often relieving the programmer from writing "obvious" assertions.

CBMC and SATABS come with an assertion generator called goto-instrument, which performs a conservative static analysis to determine program locations that potentially contain a bug. Due to the imprecision of the static analysis, it is important to emphasize that these generated assertions are only potential bugs; the Model Checker first needs to confirm that they are indeed real bugs. The plug-in permits to select which category of properties to account for as shown in Figure 3.

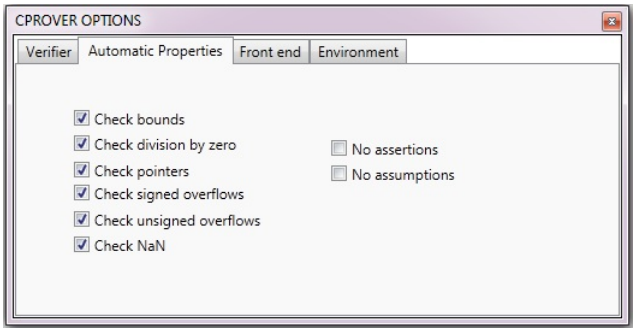


Fig. 3. Selection of the category of claims to be automatically generated

The assertion generator can generate assertions for the verification of the following properties:

- **Buffer overflows**. For each array access, check whether the upper and lower bounds are violated.
- **Pointer safety**. Search for NULL-pointer dereferences or dereferences of other invalid pointers.
- **Division by zero**. Check whether there is a division by zero in the program.
- **Not-a-Number**. Check whether floating-point computation may result in NaNs.

An example of a list of automatically generated assertions is given in Figure 4. The fist column in the properties window

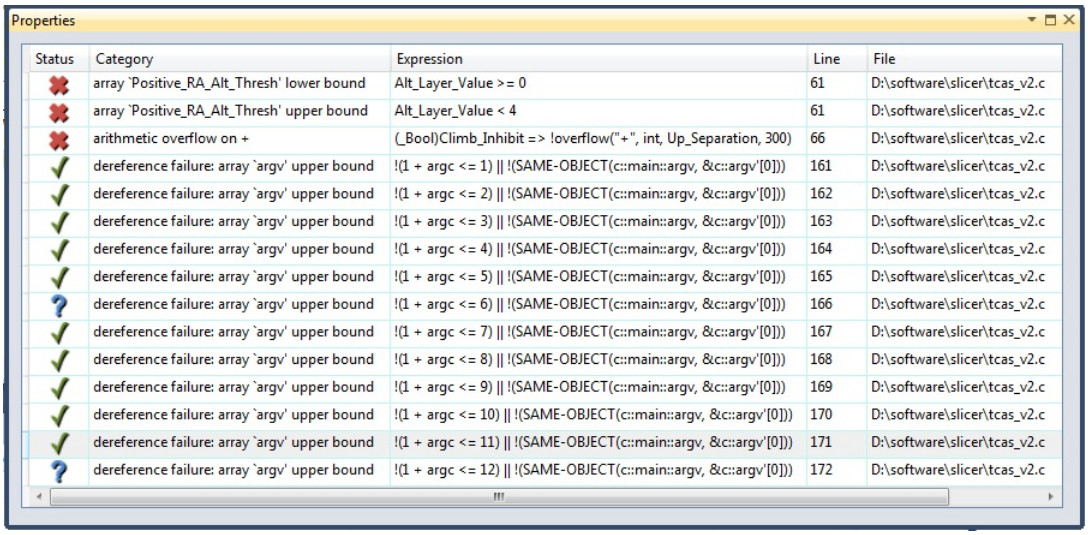


Fig. 4. List of claims which are automatically generated by the plug-in

contains the status of the property. The status can be one of the cases given in Figure 5.

| | |
|---|---|
| ? | Unknown: the property has not yet been checked. Initially, the status of all properties is unknown. |
| ✓ | The property is valid. |
| ✗ | The property has been shown to fail. |

Fig. 5. Claim status

When a property is determined to fail, the analysis tool generates a diagnostic counterexample. The counterexample aids in understanding and ultimately fixing the problem. Error traces are displayed in the Trace Viewer window, shown in Figure 6.

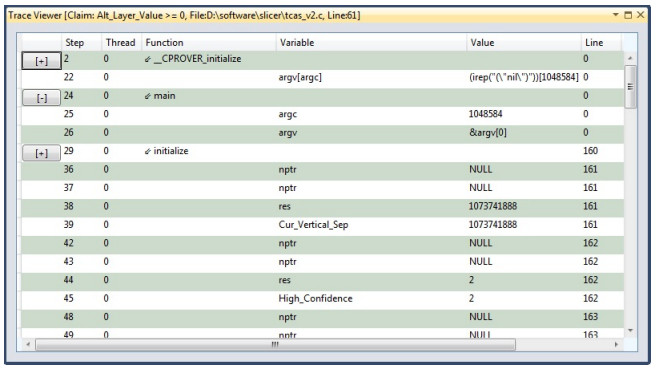


Fig. 6. Trace viewer window

When navigating through the statements in the trace, the corresponding lines in the source code are highlighted. The buttons on the far left are used to expand any function call in the trace, thus we can either step over functions or enter into them. Simultaneously, the program state (the variable values and the call stack) can be inspected in the State window (Figure 7).

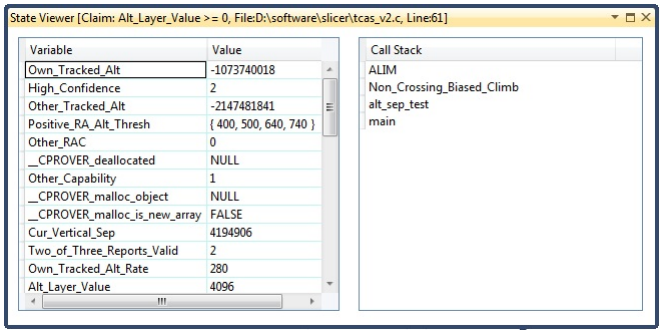


Fig. 7. State window

### B. Hardware Verification

Hardware verification is carried out via HW-CBMC. As mentioned previously, the verification process consists of checking the conformance between a program in C/C++ and a hardware design in Verilog. In this case, the program and the Verilog file are both provided as input. Diagnostic counterexamples showing inconsistencies (if present) are displayed to the user. The counterexample allows to co-debug the program with the hardware design, hence it consists of a double view: the program trace and a wave form view corresponding to hardware design as shown in Figure 8.

### C. Background Verification

Inspired by the way spell checkers in actual text editors proceed, the plug-in offers an option for background verification. Each time a document is saved the verification process is silently triggered in background. If a claim is violated its location in the source code is underlined in green. The user just needs to hover over the underlined portion of text to obtain a description of the error, as illustrated in Figure 9.

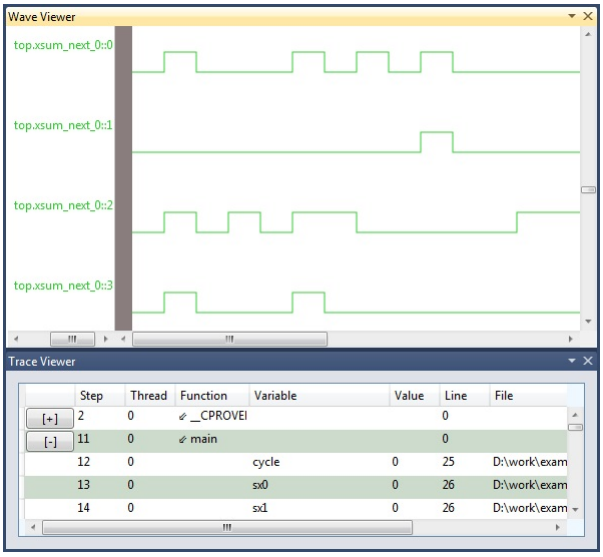


Fig. 8. Co-debugging: an error trace (bottom) and the corresponding wave form (top)

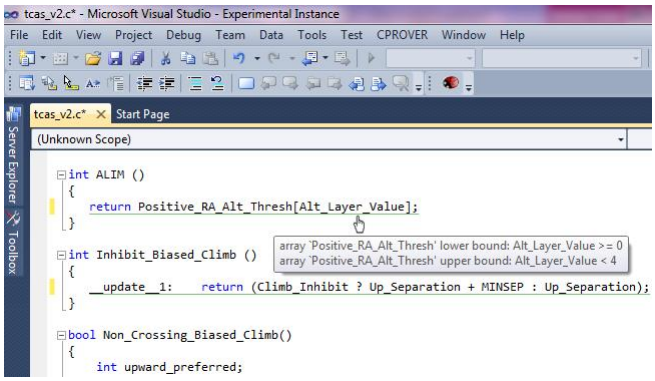


Fig. 9. Background verification within the Visual Studio IDE

## IV. Implementation

The CProver plug-in is implemented in C# as an extension package (VSPackage)[1]. It has been successfully tested with Visual Studio 2010 and 2012. Most of the windows (claim window, trace viewer, state viewer, wave viewer) are implemented as Tool Windows[2], thus they inherit some of the characteristics which are common to other IDE windows: we can dock them at any part of the IDE. We heavily used the Windows Presentation Framework (WPF)[3] technology to implement the different graphical controls for the windows as it has the advantage of separating the representation from the business logic. Finally, the background verification is based on the Managed Extensibility Framework (MEF)[4] which offers an elegant way to import and export services from the editor. In what follows, we give more details about the implementation.

[1]http://msdn.microsoft.com/en-us/library/dd885119.aspx
[2]http://msdn.microsoft.com/en-us/library/vstudio/bb165920(v=vs.100).aspx
[3]http://msdn.microsoft.com/en-us/library/vstudio/ms754130.aspx
[4]http://msdn.microsoft.com/en-us/library/dd885013.aspx

### A. Communication with CProver

Communicating with CProver tools involves calling them with the adequate parameters, managing their executions, and collecting the results they return. The .NET framework offers the class System.Diagnostics.Process to launch and monitor external processes. This class also offers means to redirect and read process output. Our initial implementation was based on a synchronous communication scheme, i.e., when a CProver tool is launched, the caller waits for the tool until it terminates. The problem with this approach is that it freezes the IDE interface until the tool terminates. This is not noticeable by the user in case of small programs, when the verification is fast, but it is impractical when the verification process needs a considerable amount of time. Our second choice was an asynchronous communication mechanism based on notification events offered by the class System.Diagnostics.Process. This time a performance problem has emerged and the communication with the verification tool became the bottleneck. As a notification event is raised each time a new line of data is produced, there is too much fine-grained communication. We finally opted for a multithreaded solution. A CProver tool is launched in a separate thread and it is synchronized with the caller when it terminates. This solution has additional advantages as we are now able to trigger the verification of multiple claims in parallel.

### B. Result Presentation

The result returned by a CProver tool is in XML format and should be presented to the user in an appropriate control element. An interesting feature that WPF offers is *data binding*, which is a simple and consistent way for control elements to interact with data. Assigning data to a given control element consists just of specifying the data source and its presentation within that element. Updates of the data source are reflected in the visual view offered by the associated control element. For example, we are representing a program trace as a list of objects of type *statement* (that we defined), and we are using a DataGrid control to present it to the user. In this case, we store the trace internally in a list of statements and describe the binding between the different fields of statements and columns of the DataGrid. So, we do not need to manipulate the DataGrid explicitly as any modification of its data source (list of statements) is automatically reflected. As mentioned previously, we use Tool Windows to host control elements. A Tool Window has a frame and a client area. The frame is provided by the IDE, it serves to control the size, docking style, etc. The client is the customizable part and hosts the control elements. A Tool window can be single- or multi-instance. In our case, the claim window and the wave viewer window are single-instance. However, trace viewer and state viewer are multi-instance windows, thus, we can visualize multiple traces simultaneously.

### C. Interaction with the IDE

Our plug-in permits the interaction with the IDE at several points. The key to this is the DTE object which represents

the Visual Studio .NET IDE and is the top-level object in the automation hierarchy. It allows us to access different components of the IDE. For example, we use the error list window of the IDE to report parsing errors and other exceptions. We interact with the solution explorer to access the different files constituting the current project. We also use the DTE object to access the active document and to interact with the editor to highlight or retrieve text. Our plug-in also relies on events raised by the IDE to trigger certain actions, e.g., launching the verification in background when a file is saved. The Visual Studio editor provides extension points as MEF component parts. MEF is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. Thus, the editor can be extended by providing components that it consumes. Based on this model, we have implemented the part handling the interaction with the source code in the background verification.

## V. Related Work

Many ESC/Java-like verification tools [18], such as Dafny [22, 8], TACO [10], CodeContracts [17], VCC [15], Boogie [7, 19] and JForge [16] have plug-in support. Most of these tools offer a rich language to specify procedure pre- and post-conditions, loop invariants, pre- and post-state relations, etc. They also have constructs to express properties over collections of elements. This makes them suitable for reasoning about data oriented properties in a modular (assume-guarantee) fashion. Thus, the plug-in support for these tools goes in the direction of supporting the user to write specifications and visualize collections of data (heap) like in [10]. The tools supported by our plug-in are more suitable the verification of control-oriented software. In our case, all we require as specification is the target assertion. In some instances assumptions are required, but both the assertion and assumption are quantifier-free formulas. Moreover, as mentioned earlier, many of the claims (assertions) that we check are generated automatically. With regards to the modular aspect, our tools offer the possibility of checking individual procedures and also to use stubs to model library functions when the implementation is missing. Finally, the co-debugging of software in tandem with hardware design makes our approach suitable for the verification of low-level embedded software.

## VI. Conclusion and Future Work

We have presented a Visual Studio plug-in for the CProver tools. This is part of our effort to make verification tools more accessible and less intrusive. A possible future direction is to improve the interaction directly through the source code, as opposed to separate, specialized views for trace and variable value visualization, property selection and so on.

## References


[1] Plug-in website: http://www.cprover.org/visual-studio/.

[2] Cprover website: http://www.cprover.org.

[3] CBMC website: http://www.cprover.org/cprover-manual/cbmc.

[4] SATABS website: http://www.cprover.org/cprover-manual/satabs.

[5] HW-CBMC website: http://www.cprover.org/cprover-manual/hwsw.

[6] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[8] N. Catano, K. Leino, and V. Rivera. The EventB2Dafny Rodin plug-in. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 49–54, June.

[9] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.

[10] M. Chicote and J. Galeotti. TacoPlug: An Eclipse plug-in for TACO. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 37–42, June.

[11] E. M. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *ASP-DAC*, pages 308–311, 2003.

[12] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.

[13] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, pages 570–574, 2005.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV*, pages 415–418, 2006.

[15] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430, 2009.

[16] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, pages 109–120, 2006.

[17] M. Fahndrich, M. Barnett, D. Leijen, and F. Logozzo. Integrating a set of contract checking tools into visual studio. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 43–48, June.

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.

[19] C. L. Goues, K. R. M. Leino, and M. Moskal. The Boogie Verification Debugger (Tool Paper). In *SEFM*, pages 407–414, 2011.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[22] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.