

A Software Safety Certification Plug-in for Automated Code Generators

Feasibility Study and Preliminary Design

Ewen Denney

USRA/RIACS

NASA Ames Research Center, Moffett Field, CA 94035
edenney@email.arc.nasa.gov

November 29, 2006

1 Executive Summary

This report summarizes the results of a feasibility study into the applicability of automated certification technology to auto-generated code, and presents a preliminary design for a software safety certification plug-in (working title, AUTOCERT) to the MathWorks Real-Time Workshop (RTW) automated code generator. The proposed tool is an adaptation of a pattern-based annotation inference technology previously developed in NASA funded research at NASA Ames.

The principal functionalities to be offered in the first version of the tool, building on RTW's code generation features, are safety certificate creation, and tracing between certification artifacts—comprising verification conditions and their proofs—and the generated code. This will build on the existing model-to-code tracing features of RTW.

The Vertical Motion System flight simulator project at NASA Ames was used as a case study. RTW was used to generate code from Simulink models provided by the VMS developers, and the code was analyzed by the verification tool. The results of the analysis were presented to the VMS team, who also provided feedback on the top-level preliminary design and functionalities of the tool.

We conclude that the use of a tightly-coupled generation/analysis tool can allow system engineers to concentrate on modeling and design, with less need to worry about low-level software details, and that the core idea of pattern-based annotation inference is a feasible basis for such a tool.

This report includes an analysis of the language subset targeted by the code generator, the extensions which are required to the underlying logical framework of the analysis tool, and the proposed architecture for integration with the Matlab environment. The preliminary design consists of logical adaptations of the existing Prolog analysis engine, with an addition of a backend to generate V&V artifacts in a form which can then be inserted in RTW-generated files.

The report concludes with plans for future development of the tool and opportunities for further application within NASA.

2 Background

We begin in Section 2.1 with some necessary background on automated code generation and different approaches which can be taken to V&V. We then give the background to our formal certification approach in Section 2.2. In Section 3, we describe our case study with the Vertical Motion Simulator. The next two sections give the preliminary design of the AUTOCERT plug-in being developed in this project, concentrating first in Section 4 on the integration with the Matlab environment, and then in Section 5, on the adaptations to the core logical machinery. Finally, Section 6 summarizes our results and describes our plans for future development.

2.1 Automated Code Generation

Model-based design and automated code generation are being used increasingly at NASA. They promise many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors.

There are now numerous successful applications of both in-house custom generators for specific projects, and generic commercial generators. One of the most popular code generators within NASA is the MathWorks Real-Time Workshop (with the add-on product Embedded Coder), an automatic code generator that translates Stateflow/Simulink models into embeddable (and embedded) C code. By some estimates, 50% of all NASA projects now use Simulink and Real-Time Workshop for at least some of their code development.

Code generators have traditionally been used for rapid prototyping and design exploration [SD06], or the generation of certain kinds of code (user interfaces, stubs, header files etc.), but there is a clear trend now to move beyond simulation and prototyping to the generation of production flight code, particularly in the Guidance, Navigation, and Control domain. Indeed, the prime contractor for the CEV has announced that code generators will be used for the development of the flight software.

Nevertheless, there remain substantial obstacles to more widespread adoption of code generators in such safety-critical domains, principally, how auto-generated code should be assured.

Ideally, the code generator, itself, should be qualified. However, this is a non-trivial and expensive process, and is therefore rarely done. Moreover, the qualification is only specific to the use of the generator within a given project, and needs to be redone for every project and for every version of the tool. Also, even if a code generator is generally trusted, user-specific modifications and configurations necessitate that V&V be carried out on the generated code [Erk04].

Since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently the generated code still needs to be fully tested and certified.

Certification requires more than black box verification of selected properties, otherwise trust in one tool (the generator) is simply replaced with trust in another (the verifier). Moreover, some understanding of why the code is safe helps the larger certification process. However, without tool qualification the regeneration of code after the model has been modified can require complete recertification, which offsets many of the advantages of using a generator. Finally, the direct V&V of code generators is too laborious and complicated due to their complex (and often proprietary) nature, while testing the generator itself can require detailed knowledge of the transformations it applies [SC03, SWC05].

2.2 Automated Code Certification

In contrast to approaches based on directly qualifying the generator, itself, or on testing of the generated code, we instead propose a generator *plug-in* to support the subsequent certification of the code created by the generator. Specifically, our tool will support certification by formally verifying that the generated code is free of different safety violations, by constructing an independently verifiable certificate, and by explaining its analysis in a textual form suitable for code reviews. This will enable missions to obtain assurance about the safety and reliability of the code without excessive manual V&V effort and, as a consequence, increase the acceptance of code generators in safety-critical contexts. The generation of explicit certificates is particularly well-suited to supporting independent V&V. The key technical idea of our approach is to exploit the idiomatic nature of auto-generated code in order to automatically infer *logical annotations*. Annotations are crucial in order to allow the automatic formal verification of the safety properties without requiring access to the internals of the code generator, as well as making a precise analysis possible. The approach is independent of the particular generator used, and need only be customized by the appropriate set of patterns.

We follow the tradition in formal methods of referring to techniques which conclusively demonstrate the absence of bugs (rather than simply search for existing bugs) as performing *certification*. However, in an IV&V context, we must consider the larger picture of certification, of which formal verification is a part, and therefore produce assurance *evidence* which can be checked either by machines (during proof checking) or by humans (during code reviews).

Rather than use a separate third-party analysis tool, we are designing a plug-in that is tightly coupled to the Real-Time Workshop code generator. We adopt the working title, AUTOCERT/RTW (AUTOCERT for short), for this safety certification plug-in. Following the plug-in philosophy, the tool acts as an extension of RTW, and is therefore closely integrated from the user's perspective, but the implementation does not require a deep integration with the internal operations of RTW.

The following sections describe the components of our system: the style of safety properties which we check, the inference of annotations, the creation and discharge of

verification conditions, and the overall system architecture.

2.2.1 Safety Properties

AUTOCERT supports certification by formally verifying that the generated code is free of violations of specific safety properties. In our approach, we distinguish between various kinds of safety properties. Language-specific properties concern those safety aspects of the code which only depend upon the semantics of the programming language. Examples include memory safety (e.g., absence of array bounds violations), variable initialization, and scoping requirements. Domain-specific properties relate to details which are specific to the use of a given code generator in a particular domain. For example, all values of x for an interpolation table (x,y) must be disjoint and in increasing order. Finally, project-specific and application-specific properties talk about guarantees for a family of applications or a single application, respectively. For example, flight-rules can be considered to comprise typical project-specific properties.

For the case study, we focused on initialization safety, but a range of other safety properties, including absence of out-of-bounds array accesses and nil-pointer dereferences, have already been formalized [Nec97, DF03] and can in principle be used with our algorithm. Initialization safety (*init*) ensures that each variable or individual array element has been explicitly assigned a value before it is used. In total, we have defined five different safety properties and implemented the corresponding safety policies. Array-bounds safety (*array*) requires each access to an array element to be within the specified upper and lower bounds of the array, and is a typical example of a language-specific property. Matrix symmetry (*symm*) requires certain two-dimensional arrays to be symmetric. Sensor input usage (*inuse*) is a GN&C specific property which is a variation of the general *init*-property guaranteeing that each sensor reading passed as an input to a state estimation algorithm is actually used during the computation of the output estimate. Another example (*norm*), from the data analysis domain, ensures that certain one-dimensional arrays represent normalized vectors, i.e., that their contents add up to one.

Details of how safety properties are formalized in our approach are given in Appendix A.

2.2.2 Hoare-Style Safety Certification

Our certification approach uses the well-known Hoare-style framework to prove the safety properties. This is based on proof rules that derive triples of the form $P \{C\} Q$, meaning “if pre-condition P holds before execution of statement C , then Q holds after”. For each safety property and each statement a corresponding rule is given. A verification condition generator (VCG) then applies the rules to a program, which produces a number of logical statements or proof obligations.

Unfortunately, the Hoare-style framework requires a large amount of logical annotations attached to statements of the code, which describe pre- and post-conditions and loop invariants. This has so far limited its practical applicability. However, it is important to observe that correctness of the proofs does not depend on correctness of the (untrusted) annotations; rather, they can be seen as hints which guide the proof process. This allows us to automatically infer the annotations without compromising the safety guarantees provided by the certification tool.

For each notion of safety the appropriate safety property and corresponding policy must be formulated. This is usually straightforward; in particular, the safety policy can be constructed systematically by instantiating a generic rule set that is derived from the standard rules of the Hoare calculus [DF03]. The basic idea is to extend the standard environment of program variables with a “shadow” environment of safety variables which record safety information related to the corresponding program variables. The rules are then responsible for maintaining this environment and producing the appropriate verification conditions (VCs). This is done using a family of *safety substitutions* that are added to the normal substitutions, and a family of *safety conditions* that are added to the calculated WPCs. Safety certification then starts with the outermost (i.e., at the end of the program) postcondition *true* and computes the weakest safety precondition (WSPC), i.e., the WPC together with all applied safety conditions and safety substitutions. If the program is safe then its WSPC will be provable without any assumptions. See Appendix A for some example rules.

2.2.3 Annotation Inference

For arbitrary (i.e., manually written) code it is impossible to automatically generate the required annotations and most annotations must be provided by the user—a prohibitively tedious and costly task. However, a code generator like RTW produces highly structured and idiomatic code. Consequently, only a few, standardized annotations need be used.

Intuitively, *idiomatic code* exhibits some regular structure beyond the syntax of the programming language and uses similar constructions for similar problems. Even manually written code already tends to be idiomatic, but the idioms used vary with the programmer, and are much less regular. Automated generators eliminate this variability because they derive code by combining a finite number of building blocks.

The idioms are essential to our approach because they (rather than the templates) determine the interface between the code generator and the inference algorithm. For each generator and safety property, our approach thus requires a customization step in which the relevant idioms are identified and formalized as patterns. Note that neither missed idioms nor wrong patterns can compromise the assurance given by the safety proofs because the inferred annotations remain untrusted. They can, however, compromise the “completeness” of the approach in the sense that safe programs can fail to be proven safe, and in our experience, a few iterations can be required to identify all patterns. Note also that the id-

idioms can be recognized from a given code base alone, even without knowing the templates that produced the code. This gives us two additional benefits. First, it allows us to apply our technique to black-box generators. Second, it also allows us to handle optimizations: as long as the resulting code can still be characterized by patterns, neither the specific optimizations nor their order matter.

We have developed a generic *pattern language* to describe these code idioms. The patterns let us define *annotation schemas* to encapsulate certification cases for matching code fragments. The annotation schemas are then applied using a combination of planning and aspect-oriented techniques to produce an annotated program, which can then be certified in the Hoare-style framework. We can thus check conformance of generated code with a range of safety properties fully automatically. As an example, consider a matrix that is initialized using a nested loop. In order to verify that the code completely initializes the matrix, we need at least four annotations: inner and outer loop invariants, which formalize “snapshots” of the matrix initialized “up to that point”, and inner and outer post-conditions, which formalize successful initialization of all or part of the matrix. Different annotations are required for row-major and column-major memory layouts. Additional complications arise when information from the initialization block needs to be propagated to parts of the code where it is needed, taking into account scope, control flow, and context. However, although the resulting annotations can become quite complex, several underlying principles can be used to generate them automatically. The only input which is needed is the basic pattern of two-dimensional iteration (which captures both memory layouts), and a definition of the initialization safety property. We already have a prototype of the inference engine, and a library of patterns which allows us to infer annotations for code generated by our own generators, AUTOBAYES [FS03] and AUTOFILTER [WS04]. We have also used the engine to analyze code produced from models in the VMS project (Section 3).

2.2.4 VC Processing

The implementation of the VCG is quite straightforward — it simply implements the semantics of the programming language and the proof rules of the safety policy.

The VCG traverses the annotated code and applies the rules of the calculus to produce VCs. These are then simplified, completed by an axiomatization of the background theory and given to an off-the-shelf high-performance automated theorem prover (ATP). If all obligations are proven it is guaranteed that the safety property is obeyed and the resulting proofs comprise the evidence for that. The VCG can be seen, therefore, as performing a *compositional* verification of the property. Note that the ATP has no access to the program internals; hence, all pertinent information must be taken from the annotations, which become part of the VCs. For full functional verification, annotations are thus usually very detailed and, consequently, annotation inference remains intractable for this case. For safety certification, on the other hand, the safety conditions are regular and relatively small,

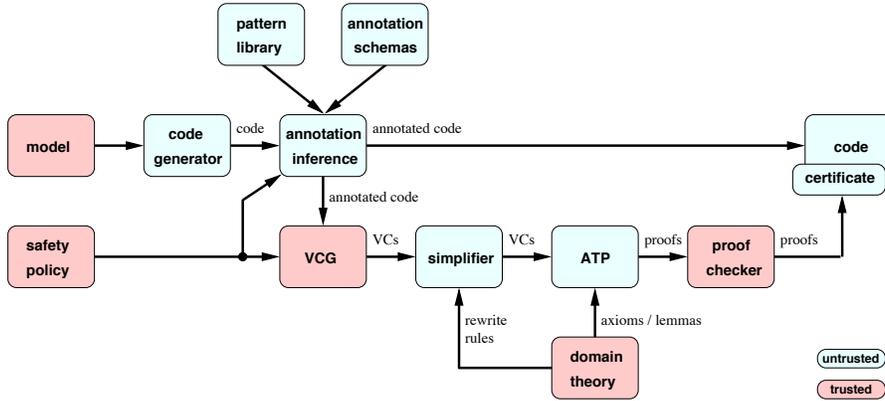


Figure 1: System Architecture

so that the required annotations are a lot simpler. For example, initialization safety just requires that the logical annotations entail at each use of a variable x that the corresponding shadow variable x_{init} has the value INIT.

2.2.5 System Architecture

Figure 1 shows the overall system architecture of our certification approach. At its core is the original (unmodified) code generator (in this case, Real-Time Workshop) which is complemented by the annotation inference subsystem, including the *pattern library* and the annotation schemas, as well as the standard machinery for Hoare-style techniques, i.e., VCG, simplifier, ATP, proof checker, and domain theory. The architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. *Trusted* components *must be correct* because any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the assurance provided by our approach does not depend on the correctness of the two largest (and most complicated) components: the original code generator, and the ATP; instead, we need only trust the safety policy, the VCG, the domain theory, and the proof checker. Moreover, the annotation inference subsystem (including the pattern library and annotation schemas) also remain untrusted since the resulting annotations simply serve as “hints” for the subsequent analysis steps.

We will omit further technical details. These components and their interactions are described in more detail in publications [DF03, DFS06, DF06a, DF06b].

3 Case Study: Vertical Motion Simulator

Our case study involved analyzing code which had been autogenerated using RTW, from Simulink models provided by the Vertical Motion Simulator (VMS)¹ facility at NASA Ames.

3.1 Overview of the VMS

The Vertical Motion Simulator (VMS) is a world-class research and development facility located in the Aviation Systems Division at NASA Ames Research Center, that offers unparalleled capabilities for conducting experiments involving aeronautics and aerospace disciplines. The six-degree-of-freedom VMS, with its 60-foot vertical and 40-foot lateral motion capability, is the world's largest motion-base simulator. The large amplitude motion system of the VMS was designed to aid in the study of helicopter and vertical/short take-off landing (V/STOL) issues specifically relating to research in controls, guidance, displays, automation, and handling qualities of existing or proposed aircraft. It is also an excellent tool for investigating issues relevant to nap-of-the-earth flight, and landing and rollout studies.

Since the VMS is effectively a piloted vehicle, the system must be human rated. Specifically, the VMS satisfies NPG 8705.2A, "Human Rating Requirements for Space Systems".

3.2 Mode Control Unit

The VMS has four hydraulic axes. Three rotational axes control roll, pitch, and yaw, respectively, and a linear axis that controls longitudinal movements. The VMS developers provided a Simulink block diagram of a single hydraulic rotational servo axis controller for use with our analysis tool. Although the Simulink block diagram provided has not yet been implemented into the VMS system, plans are underway to replace the old analog electronics that now deliver this functionality. This model was originally built with MATRIXx System Build block diagrams. Preliminary testing was conducted with this model controlling the simulator motion. Later, the model was manually converted from MATRIXx to Simulink and it is this Simulink model that will be integrated into the VMS.

The hydraulic axis model will be executed on a VME platform with a Motorola single board computer. VxWorks will be used as the real-time operating system. Real-Time Workshop will be used to generate C code from the Simulink model. Analog and discrete, input and outputs are provided by third party vendor VME boards. The model implements

¹We had originally planned to work with the RASCAL project, also at NASA Ames. However, due to some unanticipated ITAR issues this turned out not to be possible. Fortunately, we were able to find a replacement case study without any significant impact on the schedule (see Initiative Revision Request submitted on 2006-06-30).

a servo loop controller with a servo current loop, a velocity loop and a position loop. The model accepts position and velocity feed forward signals over a fiber optic digital network and provides current drive to the hydraulic actuator.

Another controller in the VMS is the Mode Control Unit (MCU) which provides the interface between the host aeronautic computer and the motion control electronics and provides manual control for the motion safety operator. This unit, once implemented with analog electronics was replaced by a digital controller built up on VME using MATRIXx and its components, System Build, AutoCode and RealSim. Plans are in place to convert this system to Simulink by manually converting the model and then using Real-Time Workshop to produce C code that will run under VxWorks.

3.3 IV&V

Since the VMS project is moving to the use of a new autocoder, namely Real-Time Workshop, the engineers are interested in tools which can ease the transition from the previous MATRIXx models.

The VMS team supplied us with their Simulink model for the MCU, and described the settings they typically use for generating code using RTW. We were then able to generate the same C code as they do, and manually translated this code into the intermediate representation format of our analysis tool. Next, we analyzed the code for the initialization safety property using a range of simplification settings.

On most settings, the code could be verified with all VCs already discharged (i.e., proven) by the simplifier. This takes under one minute. At the other extreme, performing no simplification at all produces over 700 VCs. Clearly, some experimentation is required to determine the settings which provide the most insightful output.

V&V activities for the conversions to digital controllers are done in the VMS at the system level. This is a time consuming process but is critical to get safety certification for human occupancy. For the the conversion of the MCU from MATRIXx to Simulink, the VMS team will be replacing only the software on the device and the same hardware platform will be used. The VMS developers report that they expect to see much benefit from a tool that would help them to verify that the new software (autocoded by RTW) behaves like the old software, adhering to all the requirements, without having to repeat all the many functional tests that were required when the initial installation of digital equipment was done. A tool like AUTOCERT will obviate the need to construct a huge test-suite to ensure that no low-level errors exist, and therefore helps engineers concentrate on higher-level properties.

We conclude that it is feasible to use our tool to analyze RTW-generated code, but further case studies should be carried out. The VMS engineers will provide additional models as the migration to Simulink/RTW progresses.



Figure 3: Choosing Certificate Options

useful for generating C code tuned for embedded devices. EC will not be discussed further in this document, since the differences from RTW are not relevant at this level of design. More relevantly for our purposes, however, RTW provides browsing capabilities for its generated code by generating parallel HTML files, which can be viewed with the Matlab browser. That is for every .c and .h file it generates a parallel `_c.html` and `_h.html` file. These files contain hyperlinks to type declarations and back to Simulink model elements. Clicking on a link to a model element in the code causes the corresponding box in the Simulink model to be highlighted.

4.2 Certification Functionality

We will describe the proposed integration of AUTOCERT with RTW from the point of view of the user via a series of use cases or scenarios. Note that the screenshots below are based on a current standalone implementation, so should not be taken as representative of exactly how the GUI would appear. They are used only to illustrate the functionalities being described.

The integrated tool will offer functionality in three main areas: code creation, certificate creation, and browsing/tracing between the various artifacts.

Certificate Creation It is assumed that the user has already generated code from their model using RTW. The generated files can be viewed in Matlab's "Current Directory" frame (the leftmost in Figure 4) for the project. Now the user selects the appropriate file (here `rtwdemo_counter.c`) and via a right mouse click, brings up the AUTOCERT choices dialog which allows them to choose a variety of certification options as shown in Figure 3.

First the user selects a safety policy from a predetermined list of choices – this choice

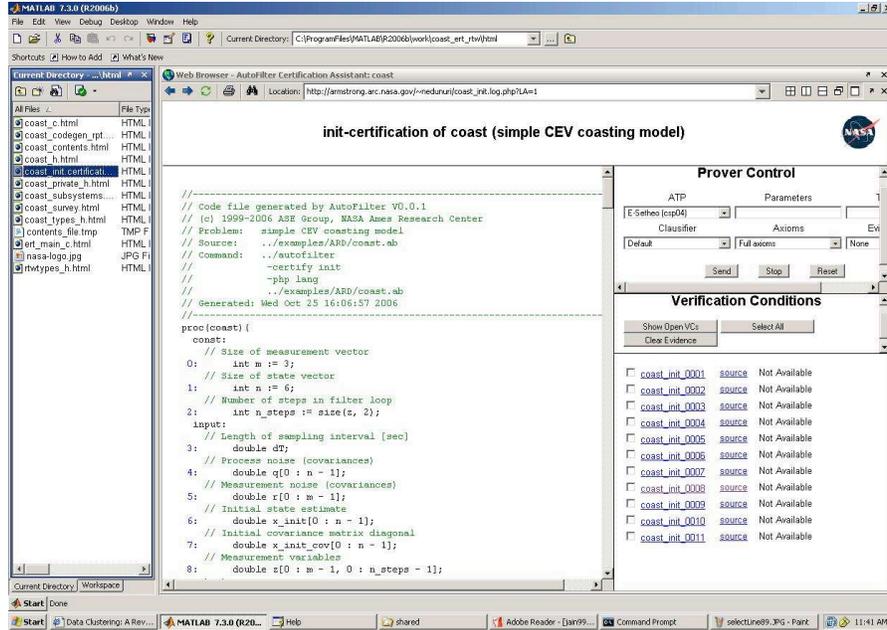


Figure 4: Generated Code and VCs

determines what kind of property of the code is to be certified. Then the user can select to generate annotations. If, and only if, the user selects generate annotations, the next choice is made available, namely whether or not to generate verification conditions (VCs). Again, if, and only if, that choice is made, the user is allowed to choose whether or not to generate certificates (formally verified proofs of the VCs). The user then selects OK to invoke AUTOCERT or Cancel to escape.

The results of the analysis can be viewed in the Matlab browser. The browser can also be started either from the menu or from the command line (using the Matlab web command). The top-level generated file is XXX.certification.html. The browser can be docked with the Matlab development environment (see Figure 4).

Tracing between Code and VCs The user can browse the generated code, and by selecting a line, see the list of VCs (bottom right frame) that are dependent on that line (Figure 5).

The user can also select a VC and navigate to its source in the code (Figure 6). This action highlights the lines in the RTW-generated code (in the left hand pane of the browser) which “contribute” to the chosen VC (that is, they had either an annotation from which the VCG generated the given VC or contributed a safety obligation).

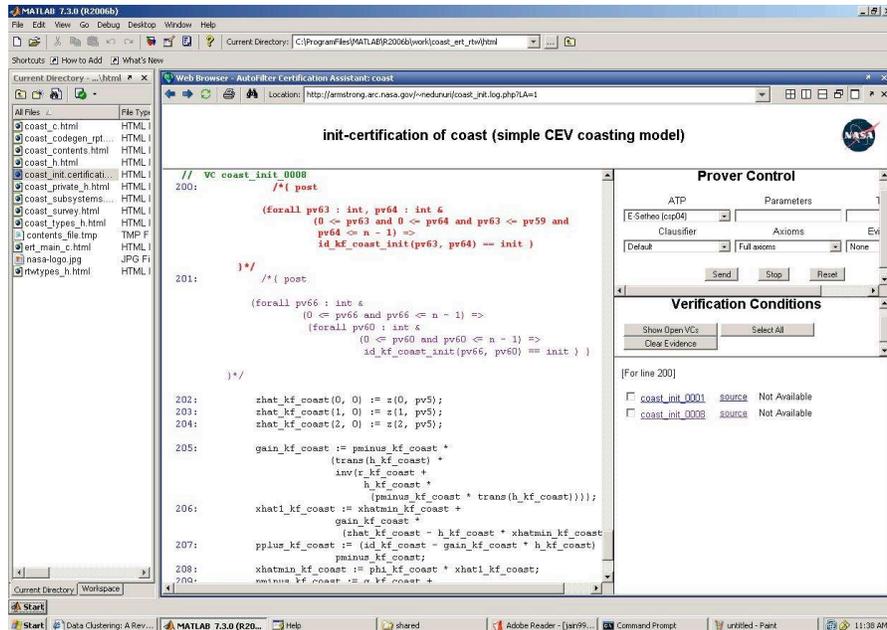


Figure 5: Tracing Code to VCs

Figure 6 shows how the tracing information can be used to support the certification process. A click on the source link associated with each VC prompts the certification assistant to highlight in boldface all affected lines of the code. A further click on the verification condition link itself displays the formula (see Figure 7), which can then be interpreted in the context of the relevant program fragments. This helps domain experts assess whether the safety policy is actually violated, which parts of the program are affected, and eventually how any violation can be resolved. This traceability is also mandated by relevant standards such as DO-178B [RTC92].

In practice, safety checks are often carried out during code reviews [NS04], where reviewers look in detail at each line of the code and check the individual safety properties statement by statement. To support this, linking works in both directions: clicking on a statement or annotation displays all VCs to which it contributes.

Along with RTW's model-to-code tracing capability, the code-to-VC tracing provides users with the ability to navigate from VCs to model elements. A more thorough and integrated functionality that permits a user to navigate directly from VCs to model and vice versa is planned.

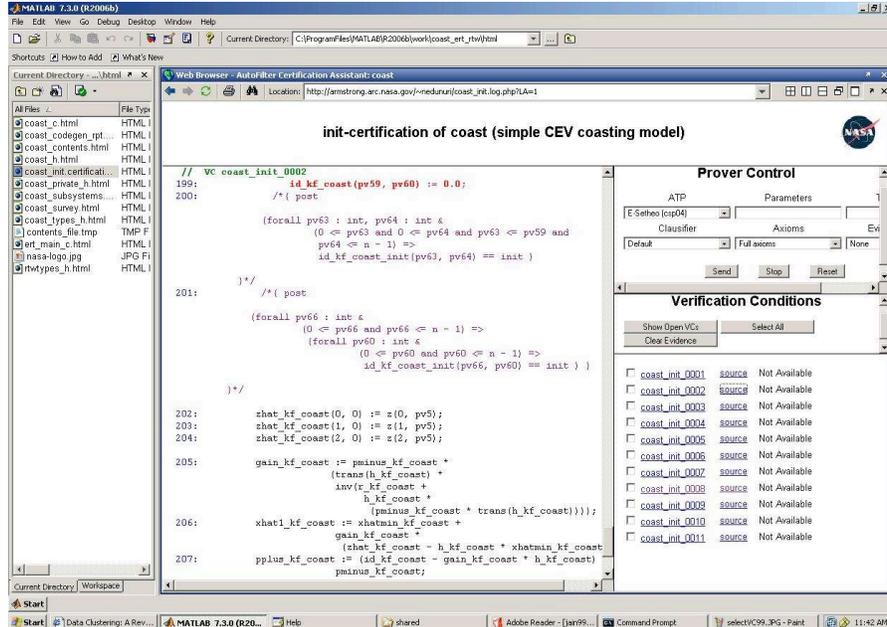


Figure 6: Tracing VCs to Code

4.3 Implementation

The use cases described above will be effected by reusing (with some modifications) existing code along with some new code. We will now consider the case of tracing code to VCs in some more detail, as it is the one that requires the most additional functionality.

There are two aspects to consider for implementing the interface. First, the mechanism by which tracing information can be incorporated into RTW-generated code; second, the representation format and language for implementing the tracing and controls (implemented as a backend to the inference engine).

4.3.1 Integration with RTW

There are a number of options for providing links from the code to the VCs. The first, and easiest, would be to generate *parallel* files that are very similar in structure to their HTML but contain links to the VCs instead of links back to the model. However, this is not very desirable from a usability standpoint as it would require the user to co-ordinate between two very similar files (the RTW generated `foo_c.html` and our generated file `foo_cert.html`). This option was not considered further. A second approach would be possible if we had access to the HTML documentation templates used by RTW (similar to the way in which

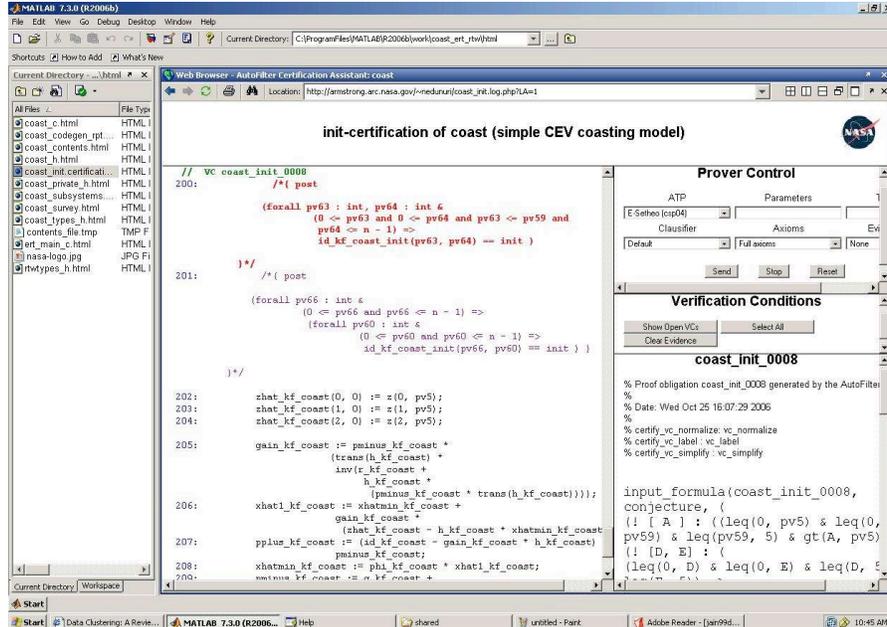


Figure 7: Viewing the Formula of a VC

the code generation templates can be customized). In that case, we could alter the HTML generation so we can add in our own links (e.g., to the VCs) in addition to the ones to the model generated by EC. Navigating between code, VCs, and model would then be seamless. Second, we could re-generate the HTML that RTW currently generates augmented by our additional content into a single HTML file. Third, we could post-process the generated HTML files to insert additional links, using an XSL script or a custom Prolog program. The work involved in each of these options is summarized below:

1. Generating Integrated HTML by changing the RTW/EC HTML templates
 - (a) Determine the required changes to the templates to incorporate (frame based) JavaScript instead of HTML
 - (b) Implement changes to templates
2. Generating Integrated HTML by generating both the existing RTW navigation and our additional navigation. In addition to (1) above we would also need to:
 - (a) Determine how to propagate tracing information from model to code (and ultimately vice-versa)

- (b) Connect datatype definitions and usage
 - (c) Design the HTML templates/schemas incorporating the above, as well as existing Matlab formatting information, such as highlighting and/or code formatting options.
 - (d) Augment the RTW build process to generate the composite HTML
3. Generating Integrated HTML by post-processing the generated HTML
- (a) Write a weaver program that combines the RTW generated HTML with the annotations and line number information generated by the inference engine into calls to pre-generated JavaScript functions
 - (b) Determine the required JavaScript to be generated
 - (c) Implement changes to the backend code generator

We chose option (3) as RTW does not currently provide access to the HTML code formatting templates, while option (2) would essentially require duplicating much of the work already done by the RTW/EC backend. The weaver program can either be an XSL script or a Prolog program that imports the RTW-generated HTML using packages such as *sgml2pl* and *sgml_write* available in SWI Prolog. The XSL script would require some way of exporting Prolog data to the XSL engine, whereas the Prolog program has the advantage that it does not require a way of passing Prolog terms to XSL.

Further integration could be achieved if the files generated by AUTOCERT could be viewed in Model Explorer instead of the browser, but that would require modifying either the Matlab generated file `XXX_contents.html` or the template that generates it. At the time of writing, however, it appears that this is not possible, although MathWorks has indicated that it may be provided for in a future release.

4.3.2 AUTOCERT Backend

There are several alternatives for representing the tracing information and we discuss these now. One option is to retain as much as possible of the existing prototype, which is based on PHP; another option is to use Matlab's native interface language (GUIDE); finally, and the option which we have chosen for the integration, we can use a JavaScript-based solution.

PHP Based Implementation

The user brings up their model in Simulink, selects the RTW menu item, and then requests code generation, with additional options being generate annotations, generate verification conditions, and generate certificate. The Matlab browser is started either from

the menu or from the command line (using the Matlab web command). From here it is possible to connect to the web server and display the top level xxx.certification.html file and interact as is done currently. Alternatively RTW generated HTML code files can be browsed. The traceability links to the model in the RTW generated HTML files work as expected. The browser can be docked with the Matlab development environment.

This approach has the benefit of being able to retain most of the current PHP setup, including the Prolog that generates it. It would enable quick prototyping of the actual functionality that we wish to demonstrate.

However it has drawbacks. It offers little possibility of integration into the Model Explorer component of RTW. This is because, unlike JavaScript, PHP must be executed on a web server. This in turn requires the user to switch between two different places: the browser for the VC traceability and Model Explorer for the rest of the functionality provided by RTW/EC. It also requires access to a PHP enabled web server.

Matlab GUI Based Implementation

In this approach, a separate GUI window, developed using the GUIDE language, would be launched from the Matlab command line. All widgets and text displays are done using Matlab GUI controls. Functionality previously implemented in PHP would be carried out using GUI callbacks.

This has better integration with Matlab than using PHP but is still a separate standalone piece of functionality from RTW. The rest of the tracing and reporting functionality provided by RTW/EC is browser based (that is, the code is viewed in an internal browser, navigation is via hyperlinks). Integrating the AUTOCERT navigation into the RTW generated HTML code files would not be possible with this option. The existing Prolog formatting code would also have to be enhanced significantly to generate Matlab GUI formatting information. Also, functionality that usually comes “for free” in a browser (e.g., hyper-linking and search, extremely useful when searching for text within code) would have to be explicitly programmed. Finally the GUI controls and callbacks would be implemented in Matlab’s custom language, thereby making it harder to port this functionality to other code generation environments.

JavaScript Based Implementation

A JavaScript based implementation allows us to integrate our functionality into Matlab in the most seamless manner. This is because JavaScript files are just HTML files with additional functions (defined in JavaScript) that are interpreted by the Matlab browser. That is, they do not require an external web server. It also allows us to retain the framework of the existing Prolog code that generates PHP since they both assume an HTML output format (as opposed to a Matlab GUI, which would be quite different).


```
function showLineNormal( line#, line)
{
    output line# as blue anchor
    output line
}
```

Accessing Matlab

The following is a fragment from the RTW/EC generated xxx.c.html file showing how RTW/EC allows linking back from code to model:

```
<SPAN class="Comment">
/* Sum: '<a href="matlab: rtwprivate rtwctags_hilite rtwdemo_counter/Sum;">
<FONT COLOR=#117755><I>&lt;Root&gt;/Sum</I></FONT></a>'
*/
</SPAN>
```

The use of the `matlab:` designates the namespace, and `rtwprivate` informs the Matlab command line interpreter that what follows is an internal function, namely the `rtwctags_hilite` function. We could probably use a similar mechanism to enable navigation between Matlab model elements and our code if needed.

4.4 Summary

We have described how we plan to integrate our additional functionality (AUTOCERT) with the Matlab/RTW GUI in a way that preserves the user experience and is as seamless as possible. Existing RTW navigation is HTML based, so we have chosen to continue with that in order to preserve the user experience. A Matlab GUI based approach was considered but rejected because it would not have been consistent with the HTML based approach used by Matlab. Although retaining the existing PHP approach would have minimized work effort it would have required access to a PHP enabled web server, and also would have precluded integration into Model Explorer.

The functionality of the use cases will generally be provided by emulating the functionality of several generated PHP files. That is, we will need to design JavaScript files that are similar to their PHP counterparts, and additionally change the presentation generators (in AUTOFILTER and AUTOBAYES) to generate this JavaScript instead of PHP.

5 Adapting Certification Infrastructure

In addition to our pilot study using the VMS, we also carried out a more general analysis of the code produced by RTW, using some alternative models. We first look in Section 5.1 at

the C code generated by the Real-Time Workshop code generator from Simulink/Stateflow models. We analyze the code constructs and idioms used by the generator, based on a range of discrete-time input models. The purpose of this is to identify the necessary extensions of the AUTOBAYES/AUTOFILTER intermediate code used in the certification system (Section 5.2), possible pre-processing steps to simplify the input for the certification plugin, and analyze the (initialization) code patterns which are used by Real-Time Workshop (Section 5.3).

5.1 Language and Code Structures

The analyzed code base uses two different versions of the Real-Time Workshop code generator:

- RTW V6.1 (R14SP1), dated September 05, 2004, together with TLC V6.1, dated August 24, 2004
- RTW V6.4 (R2006a), dated February 03, 2006, together with TLC V6.4, dated January 31, 2006

V6.1 is used in one application, while V6.4 is used in three other applications.

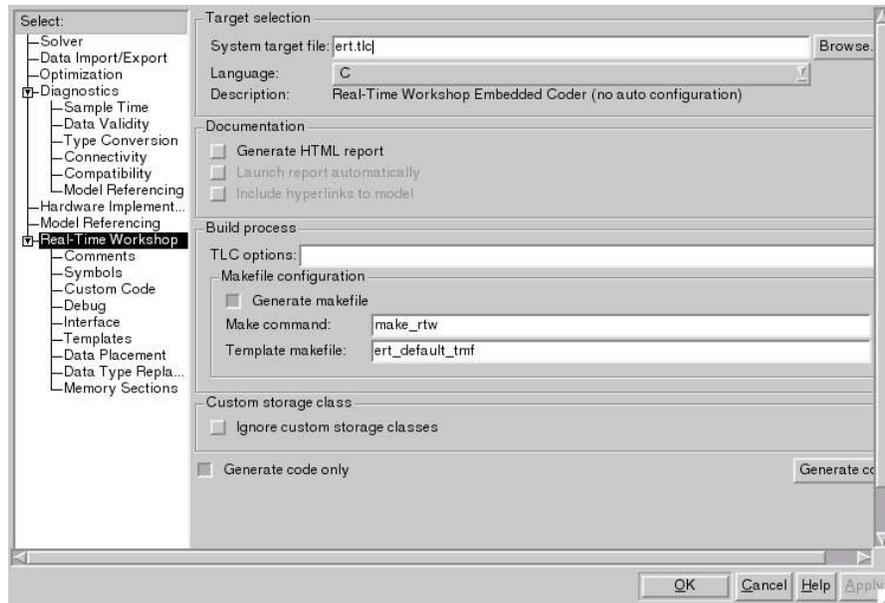


Figure 8: RTW Options Window

RTW has a number of options for the generation of code from a given Simulink model. An options window (Figure 8) is used to select the appropriate options. RTW also provides a number of target configuration files (Figure 9). All code analyzed in this case study was generated using the Simulink setting “fixed step, discrete (no continuous states)” and the template file⁴ `ert.tlc`, i.e., using the RTW Embedded Coder without a specific configuration. In addition to the selected target architecture and platform, both the model structure and the specific blocks used in the model can influence the amount and type of code that is generated. For example, a model in which updates are computed with different frequencies induces the generation of a rate-monotonic scheduler and changes the calling convention by introducing individual step functions for the different frequencies. Similarly, using RTW’s “absolute value” library block introduces the function `rtw_FABS`. However, both aspects are (mostly) ignored in the case study.

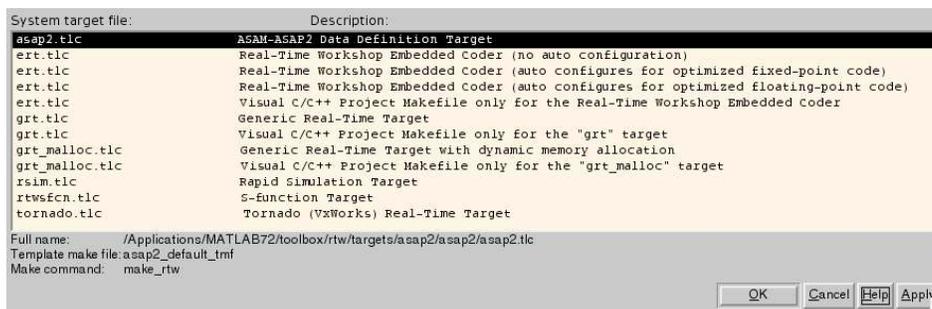


Figure 9: RTW Target Browser

5.1.1 Overall Code Structure

Real-Time Workshop produces six files for a Simulink model *model*:

model.h This file contains type definitions for the block states and external (i.e., root model) inputs and outputs, structure declarations for block block parameters and the model data structure, and `extern` declarations for the global data structures and model functions `model_initialize`, `model_step`, and `model_terminate`. It is `#included` in `model.c` and `model_data.c`. There are small differences between V6.1 and V6.4, but the overall structure and content remain the same.

model_private.h This file contains some configuration constants and declarations for imported external block signals, if the model contains these. It is `#included` in `model.c` and `model_data.c`.

⁴Template files are just configuration files that contain command line parameter settings.

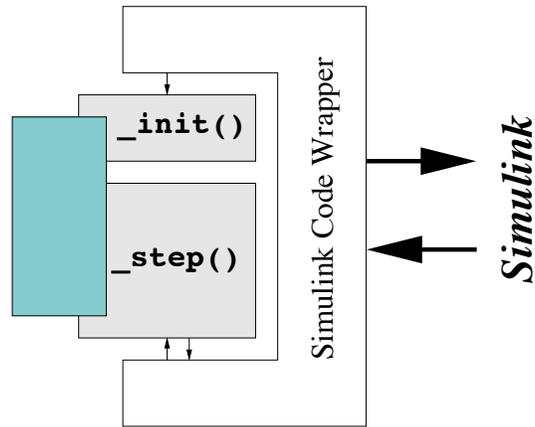


Figure 10: RTW Code Interaction

model_types.h This file contains the (forward) type declaration for `rtModel` and (if specified in the model) additional type definitions. It is `#included` in *model.c*.

model.c This file contains the implementations of the model functions `model_initialize`, `model_step`, and `model_terminate`. The body of `model_step` contains the code generated for all blocks in the model.

model_data.c This file contains initial values for block parameters and constants used by the model. It is `#included` in *model.c*.

ert_main.c This file contains wrapper code, including a main function, which then calls the top-level functions of the generated code (i.e., `model_initialize` and `model_step`). This is boiler-plate code with only a few system-specific details (i.e., calls to the generated functions). In most cases, this file is not actually used; instead, its functionality would be provided by the control module (process), which calls the generated code. This file will mostly be ignored for the analysis.

In the simplest case, the client code has a structure similar to the as following (also shown in Figure 10):

```

model_initialize(1);
...
while(!(done)){

    // get "input" data and store it in the model
    get_input_data(...);

```

```

    model_U = ...;

    // update the model
    model_step();

    // get "output" data from the model and process it
    ... = model_Y;
    use_output_data(...);

};

```

Instead of using this simple “polling” architecture, the model update *model_step* could also be called within a separate process that is activated whenever an update must be performed (e.g., at a scheduled time, or on receiving data).

5.1.2 C Pre-Processor Directives

Macro Definitions

Real-Time Workshop uses `#define` to introduce compilation and configuration control tokens, named integer constants, and boiler-plate (i.e., model-independent) parameterized access routines for the underlying model data structures. `#define` can occur in all files, with the model data structure access macros defined in *model.h*.

`#include`

The files *model.c* and *model_data.c* include the *model.h* and *model_private.h* files. *model.h* includes *model_types.h*, some RTW runtime files (*rtwtypes.h* and, in V6.1, *rtlibsrc.h*), and a number of C standard files; the latter change between V6.1 (*float.h*, *string.h*, *math.h*) and V6.4 (*stdlib.h*, *stddef.h*, *string.h*, *math.h*). Finally, *model_private.h* includes *rtwtypes.h* and, in V6.4, *rtlibsrc.h*.

Conditional Compilation

RTW uses `#ifndef` to control the definition of tokens, to prevent repeated macro definitions, and to implement conditional `#includes` but not for (proper) conditional compilation on the statement level. It is used in all header files.

5.1.3 Types

Type Usage

The RTW files use the standard types `char` and `void`; numeric types are used only in derived versions (i.e., `boolean_T`, `int_T`, `int32_T`, `uint8_T`, `uint32_T`, `real_T`, and `real32_T`; other bit-lengths might occur, depending on the the types used in the model). In addition, the defined `struct` types as well as array- (i.e., `T[]`) and pointer types (i.e., `T*`) are also used in declarations. Other types might get included, depending on the model structure.

Type Definitions

The files `model.h` and `model_types.h` contain several `struct` type definitions, which can be nested (i.e., can contain another `struct` as field). The field names of the different `structs` appear to be disjoint in the given test corpus. The definitions use both the `typedef` mechanism (i.e., `typedef struct { ... } name;`) and the basic `struct` mechanism (i.e., `struct name { ... };`); however, in the latter case, additional `typedefs` are used to introduce aliases for the *struct* types.

5.1.4 Expressions

Arithmetic

RTW uses the normal arithmetic operators, including division, and the if-then-else operator (i.e., `_?_:`). Some—but not all—numeric literals use trailing `U` and `F` as markers to denote their type (e.g., `1U`, `2.0F`). Expressions contain explicit type casts where necessary.

Hexadecimal Arithmetic

RTW does not use hexadecimal arithmetic but uses negation for booleans.

Strings

RTW does not use strings.

Side-effects

The code for `rate_monotonic_scheduler` uses a pre-increment operation (i.e., `--x`); the remaining code is free of side-effects in expressions.

5.1.5 Statements

RTW uses both simple and operator assignments (i.e., += and *=), for loops, if-then, if-then-else, switch, and increment statements. All for loops are simple, i.e., of the form `for(i=0; i<N; i++) {...}`. The switch statements all have a default branch and use a break at the end of each branch.

5.1.6 Memory Management and Pointers

RTW does not explicitly allocate / deallocate memory, but the function `model_initialize` uses both `memset` and explicit pointer aliases to initialize several data structures. The pointer aliases are used in the idiomatic form

```
void *p = (void *)&field;
for(i=0; i < N; i++) {p[i] = 0.0;};
```

where `field` is the first of `N` consecutive fields in a structure. (If `N` is small, the loop is also unrolled.) The file `model.c` also uses the address-of operator to create a pointer to the model data structure. This is accessed using the `->`-operator by the rate-monotonic scheduler.

5.1.7 Variable Declarations and Scoping

Globally visible variables are defined as `extern` in `model.h`, and declared (without `static`) on the top-level of `model.c`. The `model_step` function introduces local block I/O variables. Local blocks are used to introduce auxiliary variables, e.g., loop variables. These are not necessarily renamed apart, i.e., their names can be reused in different blocks, but declarations are never overwritten.

5.1.8 Functions and Procedures

Function and Procedure Calls

Except for the rate-monotonic scheduler, all calls are to library functions (i.e., with non-void return types), not to procedures. All calls are with value-arguments only. The functions called fall into the following two categories:

C library functions: this includes the standard mathematical functions `sqrt`, `exp`, `fabs`, and `floor`.

Real-Time library functions: this includes RTW-functions corresponding to “mathematical” blocks (e.g., `rt_ABS` or `rt_FSGN`), or to other functionalities (e.g., `rt_lookup32` or `rt_SATURATE`).

The actually occurring function calls depend on the specific blocks used in the model. The function calls are free of aliasing (i.e., no global variables passed as arguments and no repeated arguments).

Function and Procedure Declarations

RTW generates only proper procedures, i.e., functions with `void` as return type. There are no nested procedure declarations. It generates only non-recursive procedures with call-by-value parameters. The generated procedures read from and write to global (scalar and array) variables. The only formal parameter (in `model_initialize`) is of type `boolean_T`.

5.2 Intermediate Code Extensions

5.2.1 System Structure

The generated code should only be pre-processed *moderately*; running the top-level file through the pre-processor (i.e., `cpp` or `gcc -E`) leads to the inclusion of too many details. In particular:

- the relevant files (`model.h`, `model_private.h`, `model_types.h`, `model_data.c`, and `model.c`) should be combined into a single file; and
- macros should be expanded.

This can probably be achieved by forcing the pre-processor to use (empty or reduced) “dummy” files instead of the included C standard and RTW runtime files. Corresponding declarations can then be provided in the form of a “hyperscope” (i.e., standard environment) built into the certification engine.

The declarations can be mapped onto the current `system`-structure of the intermediate language:

- `static` directives can be ignored because they only appear on the top-level; and
- forward-declarations of functions (i.e., function prototypes) can be ignored because all files are merged.

5.2.2 Type Structure

The type structure of the generated programs can be simplified:

- the two different definition styles for `structs` (i.e., directly using the `typedef` mechanism, and using the basic `struct` mechanism with additional `typedefs` to introduce aliases) can be unified, preferably by mapping the latter onto the former;

- the derived numeric datatypes (e.g., `uint32_T`) can be mapped back to their “idealized” counterparts already provided by the intermediate language (e.g., `int`); and
- type casts can be ignored because they are only used to map between different versions of the same numeric base datatypes (e.g., from `int8_T` to `int_t`).

The last two simplifications assume that the generated programs are “essentially” type correct and that no type-based safety properties are shown.

The intermediate language needs to be extended by

- `typedef` statements,
- `struct` types, and
- `struct` field selection expressions.

Variant records (`unions`) are not required to handle the given code base. Pointer types are required to handle RTW’s initialization code and its aliasing of the model data structure. However, their introduction should be delayed until the required extensions of the logical framework (Luckham/Suzuki’s pointer representation or separation logic) are evaluated.

5.2.3 Statement Structure

Loops can be mapped onto the current `for` loops in the intermediate code; the front-end or the VCG should double-check that the loop index variable is not modified by the loop body (although that does not happen in the analyzed code base).

The `switch` statements could be mapped onto deeply nested `if-then-else` statements, since all branches end on a `break`. However, this would complicate tracing, and it might be easier to extend the intermediate language and the VCG by a dedicated `case` statement. The VCG would be significantly simpler if the `case` statement were based on Pascal’s semantics (i.e., implicit `break`s at the end of each branch, to prevent fall-through from one branch into the next).

5.2.4 Expression Structure

The current expression structure of the intermediate language need only be extended by `struct` field access (see above) and explicit function calls (see below).

The domain theory for the theorem provers must be extended by the hexadecimal operators only. `struct` field can be considered as atomic names for the provers, and function calls will be eliminated by the VCG.

However, depending on the representation of pointer types, it may become necessary to add further operators for dereferencing (`*`) and address-of (`&`).

5.2.5 Function and Procedure Representation

Function and procedure calls should be represented by a dedicated `fcall` operator and `pcall` statement, respectively. These only need to represent the function/procedure name and the actual parameters. Actual parameters on var-parameter positions can be represented without the explicit use of the address-of operator (`&`).

Function and procedure declarations must contain the list of parameters (marked as input, output, and inout, respectively), and the read- and write-frames (i.e., set of global variables that are accessed and modified by the function/procedure); the read-frames are only required to check that the non-aliasing assumptions built into the verification rules are satisfied. The frames can be approximated safely and easily by the set of all globally declared variables.

The declarations must allow pre- and postconditions; if no precondition is given, the VCG can either assume *true* as precondition, or an approximate precondition can be derived by computing the WSPC for the body wrt. postcondition *true*.

5.3 Code Patterns

We used initialization safety as a driving example of safety property for the case study. Real-Time Workshop generates code that uses the following idiomatic code structures to initialize array variables:

1. a non-empty sequence of assignments to the individual array elements where the indexes are the integer literals running from the array's lower to upper bounds;
2. a `for` loop running from the array's lower to upper bounds, where the body contains an assignment to an array element indexed with the loop variable;
3. a combination of the above two patterns, i.e., a `for` loop followed by additional assignments to individual elements;
4. a sequence of `for` loops, where the bodies each contain an assignment to an array element, indexed with the loop variable but with different offsets.

The first two patterns are already support by the existing prototype, while the last two patterns are new. The following code fragment shows an example for the last case:

```
for(i=0; i < 20; i++) { a[i]    = b[i]; };
for(i=0; i < 20; i++) { a[i+20] = c[i]; };
for(i=0; i < 20; i++) { a[i+40] = d[i]; };
```

6 Conclusions

We conclude that a safety certification plug-in for Real-Time Workshop is feasible, using the technology we have developed as a basis. We have three principle conclusions:

1. The use of a tightly-coupled generation/analysis tool can allow system engineers to concentrate on the modeling and design, rather than worrying about low-level software details. The developers of the VMS software have expressed their belief that tools such as we propose can help achieve this goal.
2. The Matlab environment, in general, and Real-Time Workshop, in particular, are amenable to integration with our tool. Moreover, we have had encouraging and fruitful discussions with individuals from the MathWorks.
3. The underlying logical basis for our analysis tool can be extended to deal with the output from Real-Time Workshop, and a three-phase analysis process consisting of annotation inference, verification condition generation, and verification condition discharge is suitable for proving safety properties automatically. In particular, the core idea of pattern-based annotation inference is feasible.

Our recommended design consists of:

- a core analysis engine, adapting from the existing Prolog engine (although restructuring and reimplementing will likely prove necessary at some point),
- a JavaScript backend, to convert verification and tracing artifacts from the internal Prolog representation into a form suitable for rendering in the Matlab environment, and
- an XSL-based transformation script, to insert function calls and modify the existing RTW-generated C.html file (rather than regenerate it).

Work has begun on the adaptation to the inference engine, and parts of the backend have already been implemented.

There have been two publications so far, based on work carried out within the project, both at leading software engineering conferences [DF06a, DF06b]. As the tool matures and we carry out further case studies, we will present the work at aerospace conferences. We also intend to submit a NASA technology disclosure form.

In addition to implementing the preliminary design (including any necessary extensions to the pattern library and the domain theory), future development of the tool in Phase 2 will be in several areas. First, we will extend our analysis to the Embedded Coder, to see which patterns occur in the optimized code it produces. Second, we will incorporate a safety explanation mechanism. This capability will be based on an existing prototype

which turns VCs into natural language explanations. Third, we will extend coverage of the tool to a wider range of models (that is, more blocks), though this depends on getting access to appropriate models. We also plan to look at other RTW targets, insofar as this is appropriate for our target studies, and will encode more high-level properties as safety policies. Finally, we will investigate other ways in which the analysis can provide insight into generated code. One possibility is that by computing the WPC of (the code generated by) a block/submodel, the tool can automatically determine its interface requirements. The user could also request that a submodel be certified (i.e., the code corresponding to that submodel). Integration with the Matlab Report Generator might also be useful, especially after we have integrated the safety explanation capability. We anticipate working closely with the MathWorks developers.

We are currently investigating the application of the technology to the the GReAT code generator from Vanderbilt University, in a government funded collaboration with NASA Ames. We are looking into other NASA projects which we could target for case studies. Possibilities include projects with teams at NASA Dryden and Johnston.

A Logical Framework

The analysis proceeds by first translating the parsed C code into a simple intermediate language. The logical inference is carried out on this language. Here we give examples of the rules which are used to verify initialization safety of annotated code.

$$\begin{array}{l}
(\text{assign}) \quad \frac{}{Q[e/x, \text{INIT}/x_{\text{init}}] \wedge \text{safe}_{\text{init}}(e) \{x := e\} Q} \\
(\text{update}) \quad \frac{}{Q[\text{upd}(x, e_1, e_2)/x, \text{upd}(x_{\text{init}}, e_1, \text{INIT})/x_{\text{init}}] \wedge \text{safe}_{\text{init}}(e_1) \wedge \text{safe}_{\text{init}}(e_2) \{x[e_1] := e_2\} Q} \\
(\text{if}) \quad \frac{P_1 \{c_1\} Q \quad P_2 \{c_2\} Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2) \wedge \text{safe}_{\text{init}}(b) \{\text{if } b \text{ then } c_1 \text{ else } c_2\} Q} \\
(\text{while}) \quad \frac{P \{c\} I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I \wedge \text{safe}_{\text{init}}(b) \{\text{while } b \text{ inv } I \text{ do } c\} Q} \\
(\text{for}) \quad \frac{P \{c\} I[i + 1/i] \quad I[\text{INIT}/i_{\text{init}}] \wedge e_1 \leq i \leq e_2 \Rightarrow P \quad I[e_2 + 1/i] \Rightarrow Q}{I[e_1/i] \wedge e_1 \leq e_2 \wedge \text{safe}_{\text{init}}(e_1) \wedge \text{safe}_{\text{init}}(e_2) \{\text{for } i := e_1 \text{ to } e_2 \text{ inv } I \text{ do } c\} Q} \\
(\text{comp}) \quad \frac{P \{c_1\} R \quad R \{c_2\} Q}{P \{c_1 ; c_2\} Q} \quad (\text{skip}) \quad \frac{}{Q \{\text{skip}\} Q} \quad (\text{assert}) \quad \frac{P' \Rightarrow P \quad P \{c\} Q' \quad Q' \Rightarrow Q}{P' \{\text{pre } P' c \text{ post } Q'\} Q}
\end{array}$$

Formally, a *safety property* is an exact characterization of these conditions based on the operational semantics of the language. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. Appendix A shows the

rules of the initialization safety policy as an example. The rules are formalized using the usual Hoare triples $P \{c\} Q$, i.e., if the condition P holds before and the command c terminates, then Q holds afterwards. For example, the *assert* rule says that given an arbitrary incoming postcondition Q , we must first prove that the asserted postcondition Q' implies this. We then compute the *weakest precondition* (WPC) of Q' for c and show that the asserted precondition P' implies this. The asserted precondition is then passed on as the WPC of the annotated statement. See [Mit96] for more information about Hoare-style program proofs.

The safety environment consists of shadow variables x_{init} that contain the value INIT after the variable x has been assigned a value. Arrays are represented by shadow arrays to capture the status of the individual elements. The rules can be read backwards to compute the WSPCs. For example, the *for*-rule says that for an arbitrary postcondition, Q , if c has WSPC P for the postcondition $I[i + 1/i]$, and if the two intermediate VCs are true, then the WSPC of the loop is as shown. Only statements assigning a value to a location affect the value of a shadow variable (i.e., the *assign*-, *update*-, and *for*-rules). However, all rules also produce the appropriate safety conditions $\text{safe}_{\text{init}}(e)$ for all immediate subexpressions e of the statements. Since the safety property defines an expression to be safe if all corresponding shadow variables have the value INIT, $\text{safe}_{\text{init}}(x[i])$ for example simply translates to $i_{\text{init}} = \text{INIT} \wedge (x_{\text{init}}[i]) = \text{INIT}$.

B Acronyms

ACG	Automated Code Generator
ATP	Automated Theorem Prover
CEV	Crew Exploration Vehicle
MCU	Mode Control Unit
RTW	Real-Time Workshop
RTW/EC	Real-Time Workshop with Embedded Coder
VC	Verification Condition
VCG	Verification Condition Generator
VME	Versamodule Eurocard Bus
VMS	Vertical Motion Simulator
WPC	Weakest Precondition
WSPC	Weakest Safety Precondition
XSL	Extensible Stylesheet Language

References

- [DF03] Ewen Denney and Bernd Fischer. Correctness of source-level safety policies. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proc. FM 2003: Formal Methods*, volume 2805 of *LNCS*, pages 894–913, Pisa, Italy, September 2003. Springer.
- [DF06a] Ewen Denney and Bernd Fischer. Annotation inference for the safety certification of automatically generated code. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE '06)*, pages 265–268, Tokyo, Japan, September 2006. IEEE.
- [DF06b] Ewen Denney and Bernd Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings of the Conference on Generative Programming and Component Engineering*, Portland, Oregon, October 2006. ACM Press.
- [DFS06] Ewen Denney, Bernd Fischer, and Johann Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal of AI Tools*, 15(1):81–107, February 2006.
- [Erk04] Tom Erkkinen. Production code generation for safety-critical systems. Technical report, MathWorks, 2004.
- [FS03] Bernd Fischer and Johann Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, May 2003.
- [Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
- [Nec97] George C. Necula. Proof-carrying code. In *Proc. 24th POPL*, pages 106–19, Paris, France, January 15–17 1997. ACM Press.
- [NS04] Stacey Nelson and Johann Schumann. What makes a code review trustworthy? In *Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Sciences (HICSS-37)*. IEEE, 2004.
- [RTC92] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical report, RTCA, Inc., December 1992.
- [SC03] Ingo Stürmer and Mirko Conrad. Test suite design for code generation tools. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 286–290. IEEE, October 2003.

- [SD06] Johann Schumann and Ewen Denney. Customer survey on code generators in safety-critical applications. Technical report, Robust Software Engineering Group, NASA Ames, 2006. 6G Project, Exploration Systems Architecture Studies (ESAS) Report D0306.
- [SWC05] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. *SIGSOFT Software Engineering Notes*, 30(4):1–6, July 2005.
- [WS04] Jon Whittle and Johann Schumann. Automating the implementation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, 30(4):434–453, December 2004.