A Program Certification Assistant Based on Fully Automated Theorem Provers

Ewen Denney and Bernd Fischer

USRA/RIACS, NASA Ames Research Center
Moffett Field, CA 94035, USA
{edenney,fisch}@email.arc.nasa.gov

Abstract

We describe a *certification assistant* to support formal safety proofs for programs. It is based on a graphical user interface that hides the low-level details of first-order automated theorem provers while supporting limited interactivity: it allows users to customize and control the proof process on a high level, manages the auxiliary artifacts produced during this process, and provides traceability between the proof obligations and the relevant parts of the program. The certification assistant is part of a larger program synthesis system and is intended to support the deployment of automatically generated code in safety-critical applications.

1 Introduction

Program verification remains one of the most promising applications of theorem proving, and both fully automatic and interactive provers have been used in verification projects. However, program verification has not lived up to its early promises and is not yet applied routinely in software development. This has a variety of reasons, ranging from the technical difficulties the task still poses for theorem provers, to problems in designing appropriate interfaces, and the cultural changes that are necessary in the software development process itself.

In this paper, we describe a user interface we have developed for the application of fully automatic first-order theorem provers (ATPs) in formal *program certification*. Here we employ a limited and thus more tractable variant of full program verification that uses the same basic technology but is concerned only with safety-relevant aspects of a software system rather than the complete system behavior. Formal program certification is based on the idea that mathematical proofs of the individual safety properties can be regarded as certificates which can be subjected to external scrutiny and related to the relevant safety-critical parts of the software system. This particular purpose of the proofs requires a dedicated application-oriented interface rather than a proof-oriented interface—in effect, we need a *certification assistant* rather than a proof assistant.

The role as certification assistant puts the user interface under the influence of two competing design principles. On the one hand, it has to "hide" the theorem provers from unsuspecting software engineers. On the other, it has to be open in order to provide coarse-grained control of the certification process, to maintain traceability between the different artifacts (especially source code and verification conditions), and to ensure trust in the entire certification process. As a consequence of this duality, the interface cannot completely separate the ATP from the rest of the certification environment.

Our work on certification emerged from an ongoing project on *automated program synthesis*. We have developed two synthesis systems for the domains of scientific data analysis [12] and state estimation [36], which can generate code for safety-critical application areas like spacecraft guidance, navigation, and control. Process standards such as DO-178B [23], however, require that all safety-critical software be certified to a high degree of confidence. Our goal is thus to integrate our synthesis tools with a dedicated certification environment so that end-users can trust the generated code more easily. We adopt a browser paradigm so that users can inspect the code and interact with the underlying prover, while being shielded from the low-level minutiae. We believe that our work, while still in progress, offers potential for increasing acceptance of code generators in safety-critical domains at NASA.

In Section 2, we introduce our automated synthesis and verification systems, as well as the underlying theorem provers, and describe the certification problem we address. The system architecture has a direct bearing on the certification interface, which is described in Section 3. Although the system is fully automatic, users have the option of controlling the proof process by selecting and parameterizing different provers, and inspecting the logs of prover sessions, including the proofs themselves. This is described in Sections 3.1 and 3.2, respectively. In Section 3.3, we describe the verification condition browser, which is used to relate proof obligations to the synthesized code. Section 4 describes the user model our system is aimed at. Section 5 describes related work on interfaces for prover-based verification, and Section 6 outlines our future plans.

2 Background and System Architecture

Figure 1 shows the overall architecture of our extended program synthesis system, which comprises three classes of components: the original synthesis system, the certification extensions, and corresponding document generation

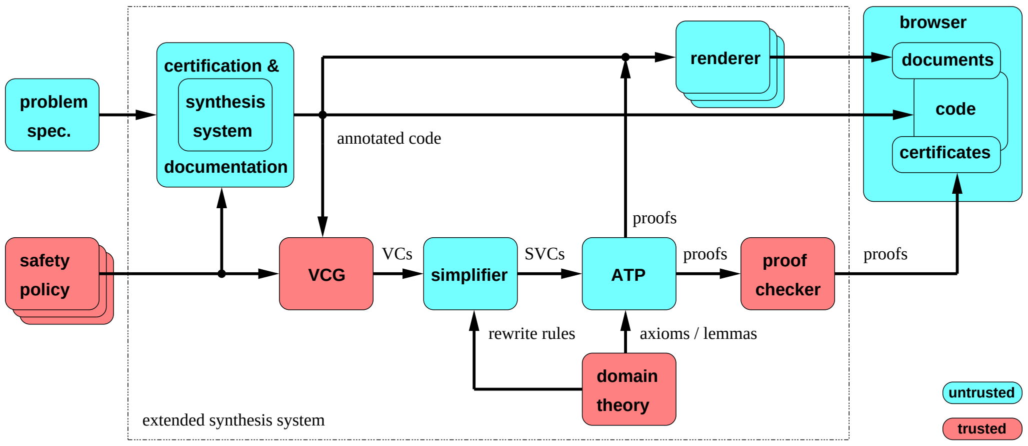

Fig. 1. Certifiable program synthesis: System architecture

extensions. We describe these components in more detail in the following sections.

2.1 Program Synthesis

Traditionally, program synthesis has followed the proofs-as-programs paradigm: the problem is specified as a conjecture in a suitable logic, an interactive theorem prover like Coq, Isabelle, or NuPRL is used to construct a proof, and a functional program is extracted from that proof and then translated into the target environment. However, this traditional, purely deductive approach to program synthesis is notoriously difficult to scale up to large problems (cf. [3]) and full automation has remained elusive. We thus follow a *schema-based* synthesis approach that combines deductive reasoning with techniques from generative programming. Most of the components described in this section are hidden inside the synthesis system box in Figure 1.

Problem Specifications. Schema-based synthesis does not necessarily require a logical conjecture as starting point for a proof. Code derivation, therefore, can begin with a specification in a more application-oriented *domain-specific language*. Our specification languages combine some target language constructs (e.g., declarations) with established scientific and engineering notations (e.g., differential equations). This allows a concise and fully declarative formulation of the problem together with some details of the desired configuration and architecture of the code to be generated.

Schemas. A *schema* is a parameterized code fragment (i.e., template) together with a set of constraints that determine whether the schema is applicable and how the parameters can be instantiated. The constraints are formulated as conditions on a problem model, which allows the problem structure to directly guide the application of the schemas and thus constrains the search space. The parameters are instantiated by the synthesis engine, either directly on schema application or by recursive calls with a modified problem. The schemas are organized hierarchically into a *schema library* which further constrains the search space. Schemas represent both fundamental building blocks (i.e., algorithms) and solution methods (i.e., transformations) of the domain; they are thus similar to the lemmas used in interactive systems but they can contain explicit calls to a meta-programming kernel in order to construct the code fragments.

Symbolic Computations. Symbolic computations are used to support schema instantiation and code optimization. The core of the symbolic subsystem is a small rewrite engine which supports associative-commutative operators and explicit contexts. It thus allows contextual rules as for example $x/x \rightarrow_{C \vdash x \neq 0} 1$ where $\rightarrow_{C \vdash x \neq 0}$ means "rewrites to, provided $x \neq 0$ can be proven from the current context C." Expression simplification and symbolic differentiation, similar to those in Mathematica, are implemented on top of the rewrite engine. The basic rules are straightforward; however, vectors and matrices require careful formalizations, and some rules also require explicit meta-programming, e.g., when bound variables are involved.

Intermediate Code. The code fragments in the schemas are formulated in an imperative intermediate language. This is essentially a "sanitized" variant of C (i.e., no pointers, no side effects in expressions etc.); however, it also contains a number of domain-specific constructs like vector/matrix operations, finite sums, and convergence-loops.

Optimization. Straightforward schema instantiation and composition produces suboptimal code; worse, many of the suboptimalities cannot be removed completely using a separate, after-the-fact optimization phase. Schemas can thus explicitly trigger large-scale optimizations which take into account information from the synthesis process. For example, all numeric routines restructure the goal expression using code motion, common sub-expression elimination, and memoization; since the schemas know the goal variables, no dataflow analysis is required to identify invariant sub-expressions, and code can be moved around aggressively, even across procedure borders.

Code Generation. In a final step, the optimized intermediate code is translated into code tailored for a specific run-time environment. We currently have code generators for the Octave and Matlab environments, and can also produce standalone Ada, C, and Modula-2 code. Each code generator employs one rewrite system to eliminate the constructs of the intermediate language which are not supported by the target environment ("desugaring") and a second rewrite system to clean up the desugared code; most rules are shared between the different code generators.

AutoBayes and AutoFilter. So far we have built two domain-specific synthesis systems following the schema-based approach outlined above. AutoBayes [12] works in the scientific data analysis domain and generates parameter learning programs, while AutoFilter [36] generates state estimation code based on variants of the Kalman filter algorithm. Both systems share a large common core (e.g., symbolic subsystem, certification subsystem, and code generators) but have their individual schema libraries. They are implemented in SWI-Prolog and together comprise approximately 100 kLoC. Both systems work fully automatically and can generate code of considerable size and complexity (approximately 1500 LoC with deeply nested loops) within a few seconds.

2.2 Certification

Unlike purely deductive approaches, schema-based synthesis cannot ensure correctness by construction. Since formally verifying the entire system is infeasible, we instead validate each generated program individually; furthermore, we concentrate on specific aspects of program safety (e.g., memory safety). The core idea is that the schemas can be extended to simultaneously generate code *and all required annotations* such that a verification condition generator can produce proof obligations which are then discharged using an automated theorem prover. The resulting proofs, which can be validated by an automated proof checker or prepared for human inspection, then serve as *certificates*. The synthesis system can generate the appropriate annotations because it has full knowledge about the form the generated code will take and the specific safety aspect that is to be certified. However, our certification approach is not necessarily tied to synthesis and the annotations could in principle also be added to code that has been implemented manually.

Safety Policies. A *safety policy* is a set of Hoare-style proof rules and auxiliary definitions which are designed to show that "a program does not go wrong," i.e., satisfies the safety property of interest [6,35]. Safety policies can be used to enforce both *language-specific* properties which can be expressed in terms of the constructs of the underlying programming language itself, and are thus sensible for any program in the language, as well as *domain-specific* properties, which typically relate to high-level concepts outside the language (e.g., matrix multiplication).

We currently support five different safety policies. Array-bounds safety requires each access to an array element to be within the declared bounds of the array. Variable initialization-before-use ensures that each variable or individual array element has been assigned a defined value before it is used. Both are typical examples of language-specific properties. For the data analysis domain, we can guarantee vector-norm safety (i.e., probability vectors add up to one). For the state estimation domain we can check proper sensor input usage (i.e., all input variables are used in the computation of the filter output) and matrix symmetry (i.e., covariance matrices are not skewed).

Annotated Code. The annotations are part of the schemas and thus are instantiated in parallel with the code fragments; further annotations are introduced by the desugaring steps of the code generation phase. The annotations contain local information in the form of logical pre- and post-conditions and loop invariants, which is then propagated through the code.

VCG. The fully annotated code is then processed by a weakest precondition verification condition generator (VCG), which applies the Hoare-rules of the safety policy in order to generate verification conditions (VCs). The VCG has been designed to be "correct-by-inspection", i.e., to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic. Hence, it does not implement any optimizations, such as structure sharing on the VCs or even apply any simplifications. As usual, the VCG works backwards through the code and verification conditions are generated at each line that can potentially violate the safety policy.

Simplification. By design of the VCG, the generated VCs are quite complex; hence, they need to be simplified before they can be discharged by an ATP. The certification extension thus re-uses the rewrite engine of the synthesizer together with a dedicated set of rewrite rules. Details can be found in [7].

ATP. For our purposes, an ATP is a search procedure which applies the inference rules of its calculus until it either finds a proof or fails because none of the rules are applicable. In order to handle extra-logical operations (as, for example, arithmetic functions), the ATP needs an additional *domain theory* that specifies their intended meaning as axioms. The provers use a set of core axioms, together with a collection of dynamically generated axioms, depending on the particular proof task.

Proof Checking. As an alternative to formally verifying the ATPs, they can be extended to generate sufficiently detailed proofs which can then be independently checked by a small, verifiable algorithm. However, due to the lack of a standardized format and various other reasons [8], there are almost no proof checkers for high-performance ATPs, in contrast to the situation for tactic-based higher-order provers. We have linked our system to the only exceptions we are aware of: the Ivy system [17], which is based on Otter, and the GDV verifier [27].

Trusted Components. Similarly to proof carrying code [20], we distinguish between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. Components are called *trusted*—and must thus be correct—if 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 correctness of the overall system does not depend on the correctness of its two largest components: the synthesizer, and the theorem prover; instead, we need only trust the safety policy, the VCG, the domain theory, and the proof checker.

2.3 Theorem Proving

We have deliberately limited our attention to fully automated theorem provers for first-order logic. This is primarily motivated by our user model which assumes no expertise in theorem proving (cf. Section 4) and thus mandates a fully automated proof process. While tactic-based provers can also provide some degree of automation (e.g., Coq's `Auto`, PVS's `grind`, or Isabelle's `Blast_tac`), their automatic modes are too weak for the emerging proof tasks and their interactive modes are incompatible with our current framework. An escape from failed proof tasks directly into the prover would need to be carefully integrated into the interface, and would only be useful for theorem proving experts, who are not our intended user class.

Moreover, first-order logic suffices for the formulation of safety policies, so using higher-order logic or constructive type theory would (from some perspectives) be introducing an unnecessary complication.

Finally, first-order ATPs benefit from a standardized syntax for proof tasks and an associated tool suite that provides translations into different input formats, preprocessing steps like clausification, and prover control. Although it would probably not be too hard to convert our output into the various tactic-based prover formats, we believe this is best left to a more general approach, perhaps by leveraging work done on Proof General [2].

2.4 Document Generation

The basic idea behind our certification approach can also be extended to human-readable documentation. The schemas contain text templates that are instantiated and composed together with the code fragments; this auto-generated text explains the relevant parts of the algorithm, give detailed derivations of mathematical formulas, and relate program constructs and variable names back to the specification.

We can also generate a standardized software design document that contains interface descriptions, administrative information (names of files, versions, etc.), specific input and output constraints, and synthesis and compiler warnings. The document is hyperlinked to the input specification, the code, and any other intermediate artifacts generated during synthesis. Since the design document is generated at synthesis time, it can refer back to the original specification and also include design details which would be difficult, if not impossible, to infer from the generated code alone.

The actual documents are generated from the commented code and the auxiliary files by different *renderers* that are specific for the targeted output formats. This also allows an easy customization of the document styles.

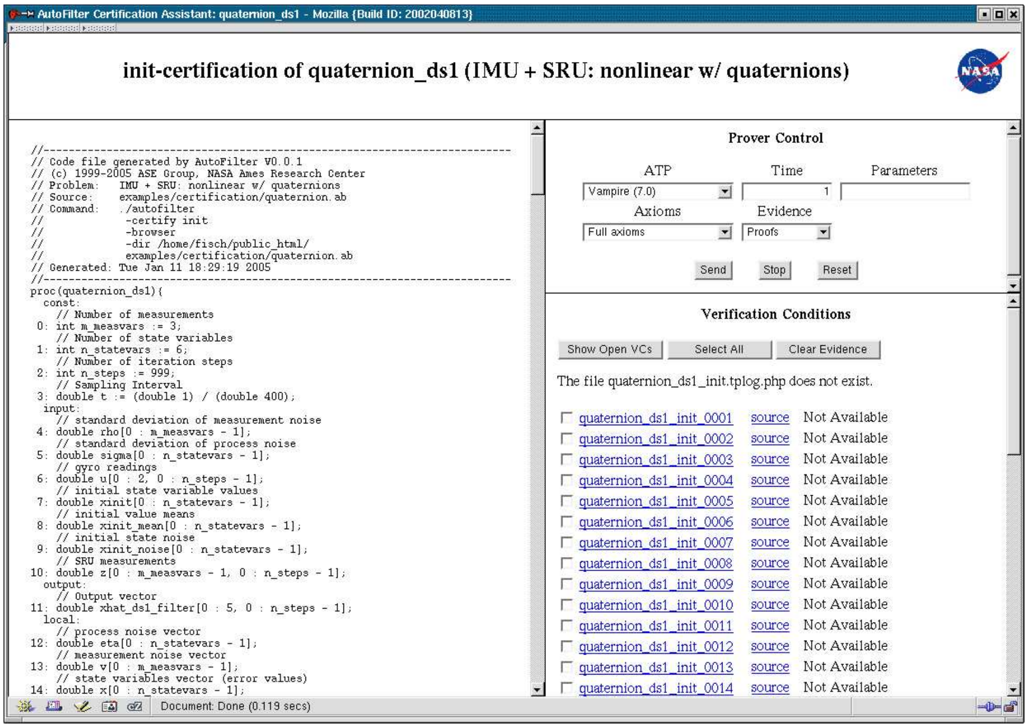

Fig. 2. Certification assistant: start-up view

3 The Certification Assistant

The certification assistant serves three main purposes. First, it allows users to customize and control the proof process on a high level. Second, it manages the auxiliary artifacts produced by this process, such as the dynamically generated axioms or clausified formulas generated by the ATPs. Third, it provides traceability between the VCs and the relevant parts of the program.

The interface mainly uses straightforward HTML as underlying technology. This is augmented with some scripting code to support the VC-browsing described in Section 3.3. The certification assistant consists of a few supporting shell scripts to control the provers, some boilerplate CGI and HTML code and a number of PHP files that are auto-generated together with the target code; in total, this amounts to approximately 2000 lines. The generation of the PHP files and the customization of the certification assistant itself is triggered by a simple command line option of the synthesis system.

Figure 2 shows the startup view of the certification assistant, after the code and supporting files have been generated. The window is split into three different areas. The left half contains a hyperlinked version of the generated code; the line numbers are used as labels by the VC linker. The right half contains a simple prover control panel and the list of verification conditions below that. Initially, no information is available about the proof status of any of the verification conditions.

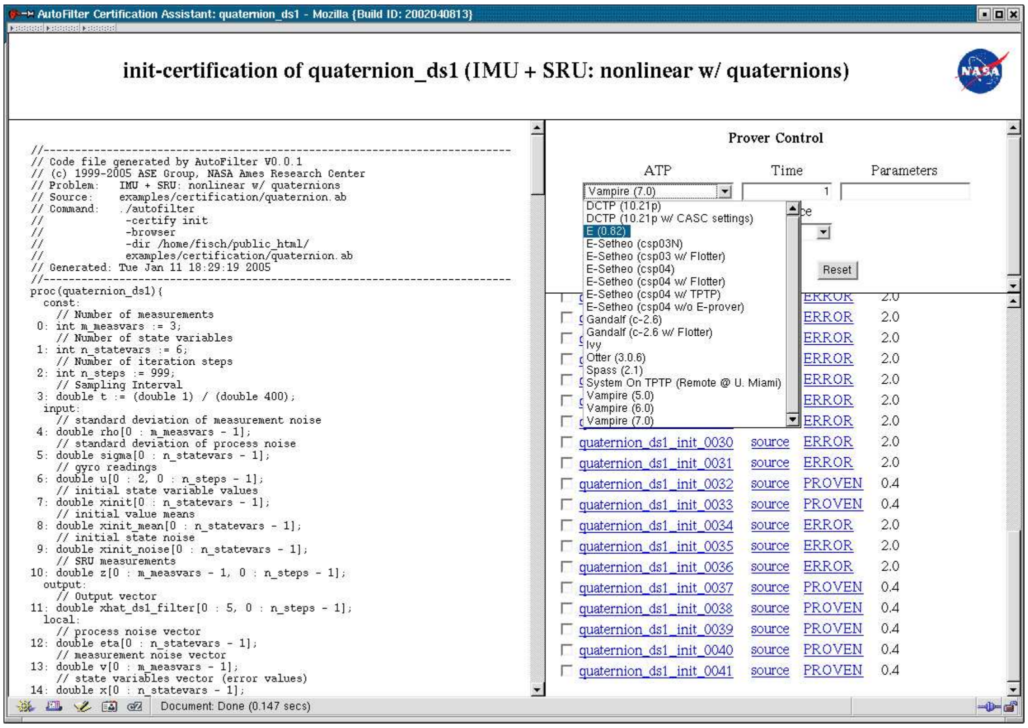

Fig. 3. Certification assistant: updated view with prover results

3.1 Prover Control

In contrast to interactive proof systems like Coq, Isabelle, or NuPRL, the user interaction here is only concerned with the parameterization of the proof process, but not with the application of individual tactics, and the prover control panel reflects this restricted interaction style. Different drop-down menus allow the user to select the theorem prover, choose between various predefined axiomatizations of the domain theory that are to be used for the proof attempts, and to select the level of evidence (i.e., proof status, prover logs, or full proofs) that the prover is required to supply. In addition, the user can specify the time limit for an individual task and any prover-specific parameters that will be passed along unchecked (e.g., term orderings). The certification process is then started by selecting any or all of the verification conditions from the list and sending the request, which the certification assistant passes to the selected prover.

Figure 3 shows the startup view updated with the prover results. For each verification condition, a link to its location in the source code (cf. Section 3.3) is displayed, together with the current proof status and the elapsed proof time. We abstract away from the varying responses produced by different theorem provers and currently the proof status is either *proven*, *error*, or *not available* (if no proof has been attempted yet). Future versions will use the more refined TSTP ontology [29]. If evidence has been produced, the status contains a link to the evidence (cf. Section 3.2). If some proof attempts fail (e.g., the verification condition *quaternion_ds1_init_0034*), the user can resend a request with different settings or with a different ATP.

The certification assistant serves as common interface to different first-order ATPs in a similar spirit to how Proof General [2] serves as common interface for different interactive higher-order provers. However, due to the limited interaction and the black-box style integration, the protocol requirements to link to a prover are much simpler than for Proof General. In particular, we can use the TPTP syntax [28] as common notation which is understood by most targeted ATPs and can be easily translated for the others. Hence, only straightforward control scripts are required for the integration; we are currently also in the process of replacing these by the control scripts used in the annual ATP system competition (CASC) [25]. The certification assistant currently integrates eight different ATPs (DCTP [16], E [24], E-Setheo [19], Gandalf [30], Ivy [17], Otter [18], Spass [34], and Vampire [22]), most of them in different versions, that run on the local server. In addition, it also provides a remote link to the SystemOnTPTP proof server at the University of Miami [26], which acts as a trusted prover component repository. The integration is simplified by the fact that SystemOnTPTP is also based on the TPTP tool suite and thus returns results in the same output format as the local installations of the ATP.

3.2 Certificate Inspection and Visualization

The certification assistant also provides access to the auxiliary artifacts that are produced during the certification. This includes the intermediate stages in the processing chain (generated axioms, clausal normal form etc.), prover log files, and actual proofs, depending on the required level of evidence. These artifacts can support, or in the absence of a proof collectively serve as, the certificate, and can be inspected as raw text files, or using third-party tools, e.g., the GDV derivation verifier [27] and the proof visualizer from the TPTP tool suite [28].

3.3 VC Linker and Browser

A VC can fail to be proven for a number of reasons. First, there may of course be an actual safety violation in the code. Second, the (generated) annotations may be insufficient or wrong. Annotation errors can come from any part of the schema, or from the propagation phase: an annotation might not be propagated far enough, or it might be propagated out of scope. Third, the theorem prover may time-out, either due to the size and complexity of the VC, or due to an incomplete domain theory. For certification purposes, however, it is important to distinguish between unsafe programs and any other reasons for failure, and in the case of genuine safety violations, to locate the unsafe parts of the program.

However, manually tracing the VCs back to their source is quite difficult as the verification process is inherently complex. The VCs can become very large and go through substantial structural simplifications, after which they are typically [7] of the form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow conc$. Here, a hypothesis, *hyp*, stems either from a loop invariant, an index bound, or a propagated annotation, and the conclusion, *conc*, is either derived from an annotated assertion or from a generated safety condition, any of which can be substituted into. Hence, a single VC can depend on a variety of information distributed throughout the program.

In order to support tracing between the VCs and the source code, the VCG adds the appropriate location information to the formulas it constructs as it processes a statement at a given source code location. We currently use simple line numbers as locations rather than individual subterm positions [14].

The source locations need to be threaded through all stages of our certification architecture (cf. Figure 1), and, in particular, through the simplifier. We have thus extended the rewrite rules used for simplification to preserve the associated labels through the rewrite process, similar to the labeled rewrite rules used in the Simplify prover [10]. This approach requires careful "rule engineering" to maintain the relevant location information while minimizing the scope of the labels and thus preventing the introduction of too much noise into the linking process. However, since each VC is generally linked to multiple statements, the location information for the entire program needs to be maintained, even if we just want to know whether a single line in the code satisfies a given safety property.

Figure 4 shows how the tracing information can be used to support the certification process. A click on the source link associated with each verification condition 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, 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 the violation can be resolved. This traceability is also mandated by relevant standards such as DO-178B [23].

In practice, safety checks are often carried out during code reviews [21], 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 (i.e., which are labeled with its line number). Figure 5 shows the result of clicking on the label for line 220; the unproven verification condition indicates that this line of code has not been completely cleared yet.

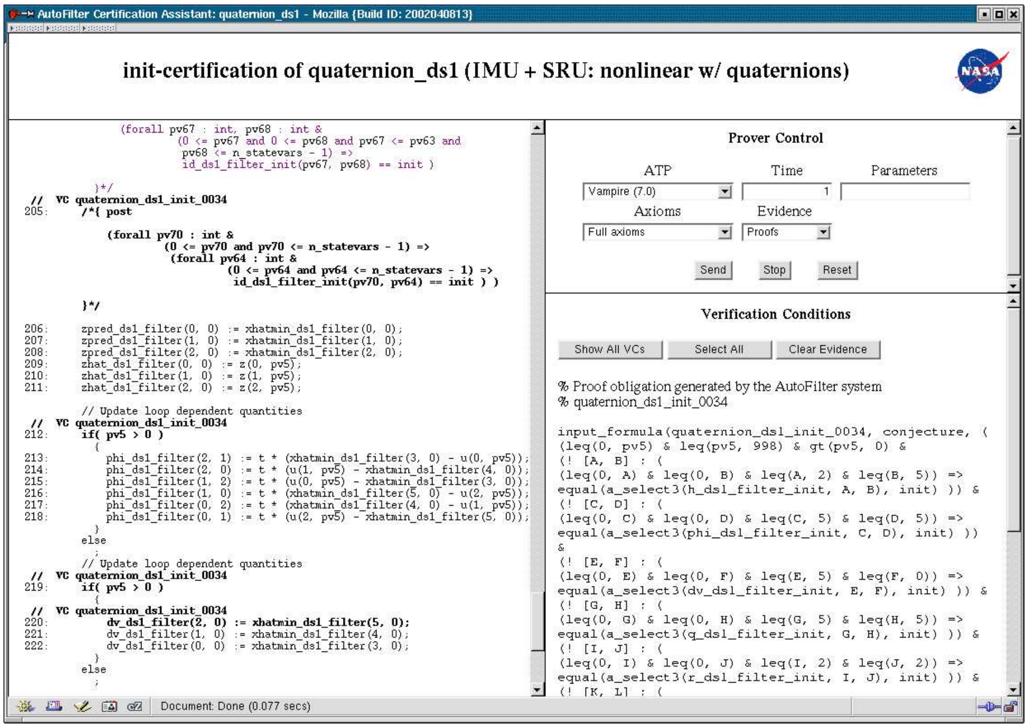

Fig. 4. Certification assistant: linking from VC

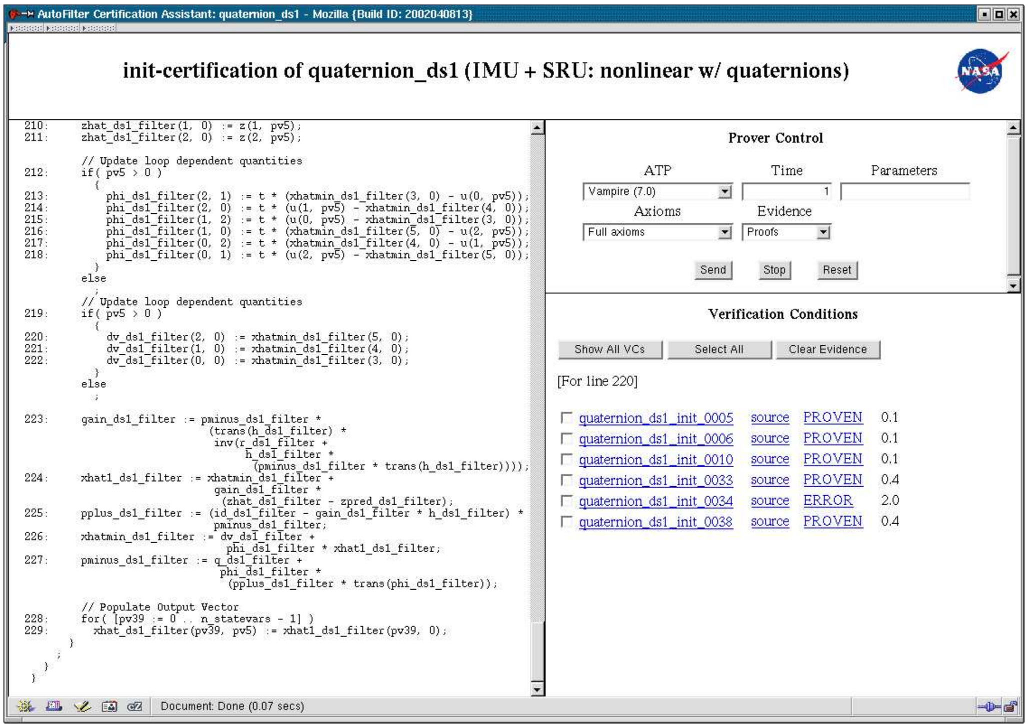

Fig. 5. Certification assistant: linking from code

4 User Model

Our system is primarily targeted towards end-user domain experts, whose knowledge is in the application domain (i.e., data analysis or state estimation). It is expected that they develop the specifications or algorithm schemas (cf. Section 2.1, but not discussed further in this paper) but not the certification machinery. Their goals are automatically certifying the code, gaining incremental assurance through redundant checks, and supporting code reviews. We assume general familiarity with the synthesis and certification process, in particular with the principle that annotated code generates a number of verification conditions for a given safety policy, on a line by line basis. However, we do *not* assume any familiarity with the logical process by which the obligations are obtained or subsequently discharged. Hence, we support a black box use of ATPs to discharge the VCs. The ability to vary the prover and to invoke an external authority both increase the level of assurance. The ability to trace in both directions between code and VC is essential to support code reviews.

A secondary user class is the system developers. Their main aim is debugging various components of the underlying synthesis systems, in particular the generated code and annotations, the safety policies, the provers and domain theories (including the dynamically generated axioms), and their installation (including the scripts). Hence, being able to vary as many components and parameters as possible, and inspecting evidence is important in isolating bugs and debugging prover scripts and installations. Likewise, tracing VCs to source code locations is essential for debugging the generated code and the annotations, as errors in the synthesizer can manifest themselves in unprovable VCs.

5 Related work

There are a number of program verification systems which use theorem proving, both automated and interactive, as the underlying technology. However, the provers are hidden to varying degrees. Fully automated provers are typically used as black-box components in a tool chain that controls the verification process and determines the form of the interface. Often, the interfaces follow a compiler style, with command line parameters as inputs and error messages as outputs (e.g., ESC/Java [13,10]), although an integration with a graphical software development environment is also possible [1]. Interactive systems, on the other hand, typically use expressive higher-order logics to model the entire software process within the system, relying on the built-in prover interface to directly control and interpret the verification process.

Perfect Developer [5] is a combined software development and verification environment. The user writes annotated programs in an intermediate language, where the annotations express a correctness specification in the design-by-contract methodology. The programs can then be translated into a variety of target languages, or analyzed using Hoare logic and an automated theorem prover.

Caveat [4] offers similar analysis capabilities, although it operates directly on source code (as opposed to an intermediate language), with the advantage that bugs can then be located directly. The tool tries to verify that the code is free of the usual range of safety violations (division by zero, null dereferencing, array out of bounds), based on Hoare logic. If the automated prover fails, the user can start an interactive proof process using Caml as scripting language. The developers' stated aim is to increase the scope for interactivity and control of the proof process. Users are also able to manually insert code annotations as required. For debugging purposes, Caveat lets users interpret failed proof obligations and relate them to their origin. It can also generate counter-examples.

Perfect Developer and Caveat are closest to our tool in terms of their approach to verification, but there are a number of key differences. Both systems require the users to supply annotations, and use fixed provers, while we automatically generate the annotations and allow the choice of many different provers, or even an external prover resource. Caveat, in particular, allows some quite sophisticated interactions with the proof process, while we have consciously adopted a more course-grained approach.

There are very few other approaches to providing explicit traceability for a program synthesis system. Van Baalen and others [32] use origin tracking [33] to indicate how statements in synthesized code relate to the initial problem specification and domain theory. They later built on this to present a documentation generator and XML-based browser interface that generates an explanation for every executable statement in the synthesized program [37].

The Proof General [2] generic prover interface aims to shield users from the low-level details of using a theorem prover. It offers a customizable user interface, while adding functionality on top of that provided by the underlying prover. Our aims are similar, although we target a different abstraction level. There are also several developing higher-order proof networks (e.g., Mathweb [15]), but it is not yet clear what role they should play in a certification assistant. The SystemOnTPTP proof service [26], which we use, can be seen as a first-order automated equivalent of these.

6 Future Work

We are currently extending the markup the VCG adds to the proof to allow a more precise and meaningful tracing. The VCG will then augment the VCs with semantic information concerning the interpretation of the different parts of the formula (e.g., “loop invariant asserted in line X”). This simplifies the interpretation of the VCs for debugging purposes and can also be used to generate high-level descriptions of the VCs. However, since this extension changes the structure of the location information, we now need to extend the interface to make use of this information.

We are also extending the interface to include specifications and design documents, thus combining our work on certification with automated safety and design document generation [9]. Eventually, we aim to provide full and seamless traceability between specification, design documentation, code, and certificates. However, the tool does not yet provide the level of integration which we would like. In particular, since the synthesis system, which generates the annotations, is not yet integrated into the interface, users can not yet use the interface to choose the safety policy.

Another line of future work is to replace some of the smaller trusted components by correct-by-construction implementations. Promising candidates are the domain theory, including the dynamically generated axioms, which could be replaced with a conservatively developed domain theory, and the rewrite engine, which takes the proof tasks through a sequence of normal forms. This combination of control and logic could be provided by tactics. In the long run, we might also look at generating tactics (instead of annotations) along with the program, especially as we move towards certifying more complicated policies.

Our broader aim is to develop a certificate management system along the lines of the Programatica project [31]. This will enable a wide range of additional capabilities, such as support for manual sign-offs on code fragments that violate stated safety policies.

Acknowledgments. Amber Telfer implemented a first version of the VC linking and browsing. Phil Oh helped extend this into the current version of the certification assistant.

References

[1] Ahrendt, W., T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager and P. H. Schmitt, *The KeY tool*, Technical Report TR 2003-05, Göteborg University (2003).

[2] Aspinall, D., *Proof general: A generic tool for proof development*, in: S. Graf and M. I. Schwartzbach, editors, *Proc. 6th Intl. Workshop TACAS*, LNCS **1785** (2000), pp. 38–42.

[3] Ayari, A. and D. Basin, *A higher-order interpretation of deductive tableau*, JSC **31** (2001), pp. 487–520.

[4] Baudin, P., G. Canet, A. Pacalet and D. Schoen, *Caveat: a tool for analysis and formal proof of C programs*, in: *Tool Exhibition Notes, FM 2003: 12th International FME Symposium*, Pisa, Italy, 2003, pp. 6–10.

[5] Crocker, D., *Perfect developer: a tool for object-oriented formal specification and refinement*, in: *Tool Exhibition Notes, FM 2003: 12th International FME Symposium*, Pisa, Italy, 2003, pp. 37–41.

[6] Denney, E. and B. Fischer, *Correctness of source-level safety policies*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *Proc. FM 2003: Formal Methods*, LNCS **2805** (2003), pp. 894–913.

[7] Denney, E., B. Fischer and J. Schumann, *An empirical evaluation of automated theorem provers in software certification*, in: *Proc. IJCAR 2004 Workshop on Empirically Successful First Order Reasoning (ESFOR)*, 2004.

[8] Denney, E., B. Fischer and J. Schumann, *An empirical evaluation of automated theorem provers in software certification*, International Journal of AI Tools (2005), to appear.

[9] Denney, E. and R. P. Venkatesan, *A generic software safety document generator*, in: C. Rattray, S. Maharaj and C. Shankland, editors, *Proc. 10th Intl. Conf. on Algebraic Methodology and Software Technology*, LNCS **3097** (2004), pp. 102–116.

[10] Detlefs, D. L., G. Nelson and J. B. Saxe, *Simplify: A theorem prover for program checking*, Technical Report HPL-2003-148, HP Labs (2003).

[11] Feather, M. S. and M. Goedicke, editors, "Proc. 16th ASE," IEEE Comp. Soc. Press, San Diego, CA, 2001.

[12] Fischer, B. and J. Schumann, *AutoBayes: A system for generating data analysis programs from statistical models*, J. Functional Programming **13** (2003), pp. 483–508.

[13] Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, *Extended static checking for Java*, in: L. J. Hendren, editor, *Proc. PLDI 2002* (2002), pp. 234–245, published as SIGPLAN Notices 37(5).

[14] Fraer, R., *Tracing the origins of verification conditions*, in: M. Wirsing and M. Nivat, editors, *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology*, LNCS **1101** (1996), pp. 241–255.

[15] Franke, A., S. M. Hess, C. G. Jung, M. Kohlhase and V. Sorge, *Agent-oriented integration of distributed mathematical services*, Journal of Universal Computer Science **5** (1999), pp. 156–187.

[16] Letz, R. and G. Stenz, *DCTP: A Disconnection Calculus Theorem Prover*, in: R. Gore, A. Leitsch and T. Nipkow, editors, *Proc. First Intl. Joint Conf. Automated Reasoning*, LNAI **2083** (2001), pp. 381–385.

[17] McCune, W. and O. Shumsky, *System description: Ivy*, in: D. McAllester, editor, *Proc. 17th CADE*, LNAI **1831** (2000), pp. 401–405.

[18] McCune, W. and L. Wos, *Otter—the CADE-13 competition incarnations*, JAR **18** (1997), pp. 211–220.

[19] Moser, M., O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann and K. Mayr, *The model elimination provers SETHEO and E-SETHEO*, JAR **18** (1997), pp. 237–246.

[20] Necula, G. C. and P. Lee, *The design and implementation of a certifying compiler*, in: K. D. Cooper, editor, *Proc. PLDI 1998* (1998), pp. 333–344, published as SIGPLAN Notices 33(5).

[21] Nelson, S. and J. Schumann, *What makes a code review trustworthy?*, in: *Proc. Thirty-Seventh Annual Hawaii International Conference on System Sciences (HICSS-37)* (2004).

[22] Riazanov, A. and A. Voronkov, *The design and implementation of Vampire*, AI Communications **15** (2002), pp. 91–110.

[23] RTCA Special Committee 167, *Software considerations in airborne systems and equipment certification*, Technical report, RTCA, Inc. (1992).

[24] Schulz, S., *E – A Brainiac Theorem Prover*, Journal of AI Communications **15** (2002), pp. 111–126.

[25] Sutcliffe, G., *The CADE-J2 ATP System Competition* (2004), www.tptp.org/CASC/J2/.

[26] Sutcliffe, G., *System on TPTP* (2005), www.tptp.org/cgi-bin/SystemOnTPTPFormMaker.

[27] Sutcliffe, G. and D. Belfiore, *Semantic Derivation Verification*, in: I. Russell and Z. Markov, editors, *Proceedings of the 18th Florida Artificial Intelligence Research Symposium* (2005), to appear.

[28] Sutcliffe, G. and C. Suttner, *TPTP home page* (2003), www.tptp.org.

[29] Sutcliffe, G., J. Zimmer and S. Schulz, *TSTP Data-Exchange Formats for Automated Theorem Proving Tools*, in: W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, IOS Press, 2004 pp. 201–215.

[30] Tammet, T., *Gandalf*, JAR **18** (1997), pp. 199–204.

[31] The Programatica Team, *Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell*, in: *Proc. High Confidence Software and Systems Conference*, Baltimore, MD, 2003. Available at www.cse.ogi.edu/PacSoft/projects/programatica.

[32] Van Baalen, J., P. Robinson, M. Lowry and T. Pressburger, *Explaining synthesized software*, in: Feather and Goedicke [11], pp. 240–248.

[33] van Deursen, A., P. Klint and F. Tip, *Origin tracking*, JSC **15** (1993), pp. 523–545.

[34] Weidenbach, C., *Spass home page* (2003), `http://spass.mpi-sb.mpg.de`.

[35] Whalen, M., J. Schumann and B. Fischer, *Synthesizing certified code*, in: L.-H. Eriksson and P. A. Lindsay, editors, *Proc. FME 2002: Formal Methods—Getting IT Right*, LNCS **2391** (2002), pp. 431–450.

[36] Whittle, J. and J. Schumann, *Automating the implementation of Kalman filter algorithms*, ACM Transactions on Mathematical Software **30** (2004), pp. 434–453.

[37] Whittle, J., J. Van Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry and G. Brat, *Amphion/NAV: Deductive synthesis of state estimation software*, in: Feather and Goedicke [11], pp. 395–399.