A super industrial application of PSGraph*

Yuhui Lin[1], Gudmund Grov[1], Colin O'Halloran[2] and Priiya G[2]

[1] Heriot-Watt University, Edinburgh, UK {Y.Lin,G.Grov}@hw.ac.uk
[2] D-RisQ Software Systems, Malvern, UK {coh,priiya.g}@drisq.com

Abstract. The ClawZ toolset has been successful in verifying that Ada code is correctly generated from Simulink models in an industrial setting, using the Z notation. D-RisQ are now extending this technique to new domains and the C programming language, which require changes to their highly complex proof technique. In this paper, we present initial results in the technology transfer of the graphical PSGraph language to support this extension, and show feasibility of PSGraph for industrial use with strong maintainability requirements.

1 Introduction

The ClawZ toolset is used to verify that automatically generated code from (a subset of) Simulink (www.mathworks.com) into (a verifiable subset of) Ada is correct [4]. This is achieved by encoding the semantics of the two representations in the Z notation. Correctness of the generation is then ensured by a formal proof of a refinement conjecture from the Z representation of Simulink into the Z representation of Ada. This proof is supported by a very powerful proof tactic for the ProofPower theorem prover called *Supertac* [4]. This tactic has been developed over a number of years and has a very high degree of automation as it is tailor-made for these types of conjectures.

Whilst Ada is used extensively for avionics software, other sectors, such as automotive, normally uses the C programming language. The TargetLink code generation from dSPACE (www.dspace.com) is able to generate C code from a Simulink model. However, Supertac is configured for Ada and will, as will be shown in the next section, not be able to verify correctness of C generation.

A side-effect of the automation achieved by Supertac is that the code base has become highly complex and large. In fact, it consists of almost 50K lines of dense ML code[3]. Adapting it from Ada to C is therefore a non-trivial problem.

Tactic languages, such as the one used to encode Supertac, are often difficult to analyse and debug as the error may manifest itself a different place from where the problem actually lies. Often the only method to find the mistakes are to insert "`writeln`" statements in the code to print information. Needless to say, this can be a hard task for 50K LoC. Moreover, as proofs are getting larger and more commonplace, proof maintenance is going to be a problem: e.g. as time elapses any slight change to the prover will be problematic because the people who originally did the proof and tactics work, and therefore has all the deep knowledge of the work, will have moved on or retired.

* This work has been supported by EPSRC grants EP/J001058 and EP/K503915. The second author has been supported by a SICSA industrial fellowship.

[3] As far as we know, this is the largest proof tactic ever made in terms of code size.

To ease maintenance, debugging and general understanding of tactics, we have previously developed the *PSGraph* language [1] and the supporting *Tinker* tool [2,3]. Here, proof tactics are encoded as directed hierarchical graphs, where the boxes contain tactics or nested graphs, and are composed by labeled wires. The labels are called *goal types* and are predicates that describe the expected properties of sub-goals. Each sub-goal becomes a special *goal node* in the graph, which "lives" on a wire. Evaluation is handled by applying a tactic to a goal node that is placed on one of its input wires. The resulting sub-goals are sent to the out wires of the tactic node. To add a goal node to a wire, the goal type must be satisfied. Compared with traditional tactic languages, this offers better static control of sub-goal flow (via the goal types), and support for inspection and adaptations through simple graph visualization and drawing.

We will show our initial encoding of Supertac in PSGraph in §2. We will then show how this is used to analyse Supertac using PSGraph and adapt this to support C code in §3. We conclude and detail our next steps in §4.

2 PSGraph encoding of Supertac for Ada

PSGraph handles modularity and complexity through *hierarchies*, which is a box in the graph that contains a sub-graph. The architecture of Supertac consists of four sub-tactics executed in sequence. In our current encoding the first of these has been decomposed in a hierarchical node; in the future we plan to decompose the remaining three. Fig. 1 shows the top-level of our Supertac encoding:

structure_tac is used to classify the conjectures and enhancing them with meta-information. In addition, it unpacks structure surrounding conjectures, such as quantifiers.

end_proof_tac1 gets rid of Simulink and Ada vocabulary, to make it easier to reason about.

end_proof_tac2 deals with mathematical statements, and gets rid of high-level concepts (e.g. functions) to reduce it to set theoretical primitives that are easier to reason about.

end_proof_tac3 handles case statements and is mainly used as a brute-force strategy if end_proof_tac2 has not been able to discharge the conjecture.

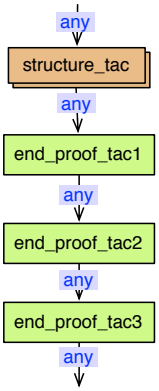

Fig. 1. Supertac

Fig. 2 shows the tactic nested by the structure_tac box in PSGraph. As a proof of concept, this encoding has been successfully applied to discharge VCs generated from a *Nose-Gear Velocity* case study [5] in ClawZ.

3 Adapting Supertac to C-code

The main difference between conjectures generated for Ada and C is that Ada specific semantics in the refinement conjecture has now been replaced by C constructs. The remaining parts, e.g. the encoding of Simulink using Z Notation, is unchanged. The key challenge is thus to replace the parts of Supertac that reduce Ada vocabulary with pure set theory into parts that reduces C vocabulary

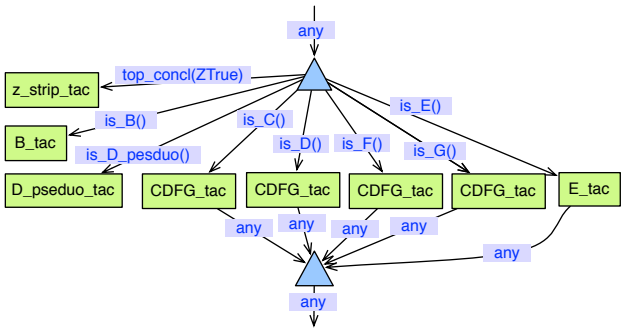

Fig. 2. The structure_tac sub-tactic of Supertac

into set theory (and develop these). This is non-trivial, because semantically, variables in Ada are easy to reason about when aliasing restrictions are in place, however the semantics of a variable in C comes with a side condition that it is disjoint from other C objects. PSGraph enables users to step through evaluation and see how the goals evolve. This is very useful when adapting a tactic in this way. To illustrate, the following VC has been generated from a simple C program:

$$\lceil (*\ ?\vdash\ *)_Z \forall\ [fa_v : VALUE_C;\ \sigma : STORE_C \mid \langle (fa_v,\ fa_l)\rangle\ AllocatedIn\ \sigma] \quad \bullet$$
$$(Test01_v! \mathrel{\widehat{=}} IntVal_C\ (mk\_signed\_int_C\ (IntOf_C\ fa_{\_} + 1)).\ fa_{\_} \mathrel{\widehat{=}}$$
$$fa_v,\ \sigma \mathrel{\widehat{=}} \sigma) \in Test01_{post} \rceil$$

By exploiting the debugging support of Tinker, we were able to interactively step through the proof of this VC in our PSGraph version of Supertac. This helped us to pinpoint where in end_proof_tac1 the target programming language constructs had to be handled. First, this sub-tactic was split into end_proof_tac1_1 and end_proof_tac1_2. The C specific parts will need to be eliminated between these parts. To illustrate, after end_proof_tac1_1 our VC looks as follows:

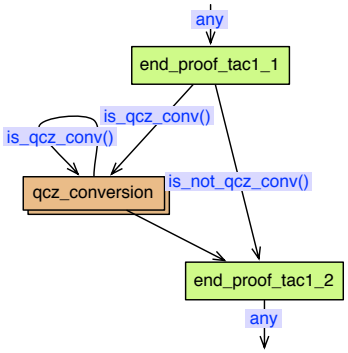

Fig. 3. New end_proof_tac1

$$\lceil (*\ \ 3\ *)_Z \sigma \in STORE_C \rceil$$
$$\lceil (*\ \ 2\ *)_Z \langle (fa_v,\ fa_l)\rangle\ AllocatedIn\ \sigma \rceil$$
$$\lceil (*\ \ 1\ *)_Z clawz\_hint1\ "Supertac{:}VC\_Origin{:}Empty\_Block\_List" \rceil$$
$$\lceil (*\ ?\vdash\ *)_Z (true \wedge \sigma \in STORE_C \wedge true)$$
$$\wedge\ IntVal_C\ \ (IntOf_C\ (IntVal_C\ (mk\_signed\_int_C\ (IntOf_C\ fa_v + 1)))$$
$$==_{CZ}\ mk\_signed\_int_C\ (IntOf_C\ fa_v + 1)) \neq IntVal_C\ 0 \rceil$$

Note that assumption *(* 1 *)* has been inserted by structure_tac to guide the rest of the proof. When the VC contains C specific parts, then these have to be reduced to set theory before end_proof_tac1_2 executes. As can be seen in Fig. 3, a new nested tactic called qcz_conversion was introduced between the two tactics discussed above. VCs containing C vocabulary are identified by the is_qcz_conv predicate, found on the wire leading to the qcz_conversion tactic.

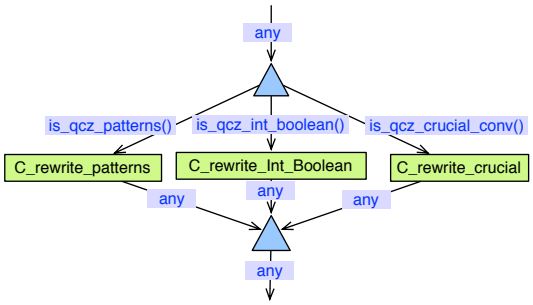

Fig. 4. Conversion types

The nested graph of qcz_conversion is shown in Fig. 4. Depending on certain properties of the goal, identified by the wire labels of the PSGraph, one of three tactics may be applied.

Each of them applies some *conversions*, which are rewrite rules proven from the semantics of the language formalisation, to simplify the goal: C_rewrite_patterns does general simplifications of C constructs; C_rewrite_Int_Boolean deals special simplifications related to C booleans (represented as integers in C); while C_rewrite_crucial does the crucial steps in fully eliminating the C specific vocabulary. Below, some of the *crucial* conversions of C_rewrite_crucial are listed:

$$\forall\ x : \mathbb{U} \bullet IntOf_C\ (IntVal_C\ x) = x$$

$$\forall\ x : \mathbb{U};\ y : \mathbb{U} \bullet IntVal_C\ x = IntVal_C\ y \Leftrightarrow x = y$$

These changes to the PSGraph was sufficient to complete running examles, and has been successfully applied to fully automate six handcrafted VCs.

4 Conclusion & future work

In this paper we have reported on a successful technology transfer project, where the PSGraph language has been used to start the adaptation of an industrial proof strategy to a new domain – using the state-based Z notation. Through an example, we have illustrated how it has been used to pinpoint and support the development of this adapted proof strategy.

In the medium term we are hoping that PSGraph can be use to remove some of the "clutter" that has been the result of updating Supertac to new applications. In the long term we would like to re-implement the overall structure from scratch, using the existing Supertac components as building blocks. This will enable us to reflect on the intuition and develop a more conscious strategy, where the overall proof plan is clear. For example, we should separate reasoning about the denotational semantics given to expressions from the operational semantics of statements, as these will be tackled in different ways.

References

1. G. Grov, A. Kissinger, and Y. Lin. A Graphical Language for Proof Strategies. In *LPAR*, pages 324–339. Springer, 2013.
2. G. Grov, A. Kissinger, and Y. Lin. Tinker, tailor, solver, proof. *UITP 2014*, 2014.
3. Y. Lin, P. Le Bras, and G. Grov. Developing & Debugging Proof Strategies by Tinkering. In *TACAS 2016*, to appear.
4. C. O'Halloran. Automated verification of code automatically generated from simulink. *ASE*, 20(2):237–264, 2013.
5. C. O'Halloran. Nose-Gear Velocity–A challenge problem for software safety. ASSC 2014, held in Melbourne 28-30 May, 2014.