A Super Industrial Application of PSGraph

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 is now extending this technique to new domains of the C programming language, which requires 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[3] into (a verifiable subset of) Ada is correct [7]. 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* [7]. 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 use the C programming language. The TargetLink code generator from dSPACE[4] 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[5]. 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. This is further complicated by: (1) the *non-deterministic nature* of tactics, meaning multiple branches can be generated;

* This work has been supported by EPSRC grants EP/J001058, EP/K503915, EP/M018407 and EP/N014758. The second author is supported by a SICSA Industrial Fellowship.

[3] See www.mathworks.com.
[4] See www.dspace.com.
[5] As far as we know, this is the largest proof tactic ever made in terms of code size.

and (2) that they tend to be *untyped* in the sense that subgoals cannot be differentiated. This non-deterministic nature makes it more difficult to debug than most software, and often the only method to find the mistakes is to insert "`writeln`" statements in the code to print information. Needless to say, this can be a hard task for 50K LoC. As proofs are getting larger and more commonplace, proof maintenance is going to be a problem. Improved debugging features for ML would be beneficial and a step in the right direction, but this alone will not be sufficient: the non-determinstic nature of tactics and the non-trivial flow of sub-goals will require their own solutions.

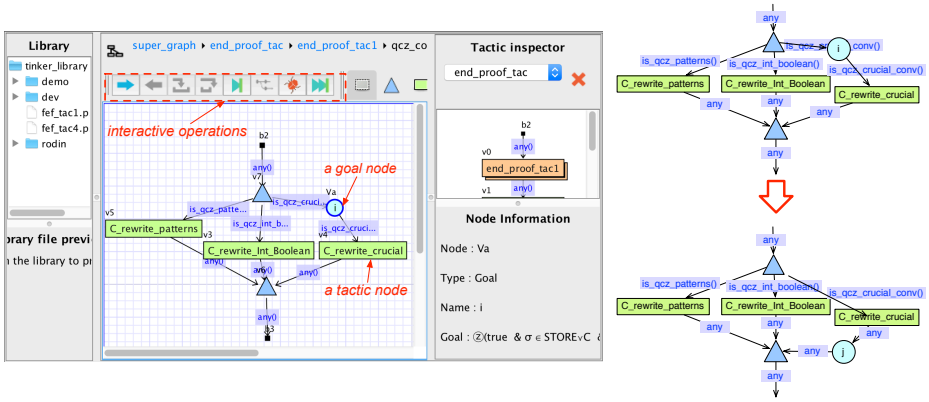

Fig. 1. The GUI of Tinker (left) & an illustrative evaluation step (right)

To ease maintenance, debugging and general understanding of tactics, we have previously developed the *PSGraph* language [1] and the supporting *Tinker* tool [2,5], as can be seen in Fig. 1 (left). Here, proof tactics are encoded as directed hierarchical graphs, where the boxes contain tactics or nested graphs, and are composed by labelled 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. Fig. 1 (top right) illustrates a single evaluation step where tactic C_rewrite_crucial, which is discussed in §3, is applied to a goal labelled i:

$$\begin{aligned}&\ldots\ (*\ ?\vdash\ *)^{\lceil}_{Z}(true \wedge \sigma \in STORE_C \wedge true)\\ &\qquad \wedge\ IntVal_C\ \ (\ IntOf_C\ (IntVal_C(\ mk\_signed\_int_C\ (IntOf_C\ fa_v\ +\ 1)\ ))\\ &\qquad\qquad ==_{CZ}\ mk\_signed\_int_C\ (IntOf_C\ fa_v\ +\ 1)) \neq IntVal_C\ 0^{\rceil}\end{aligned}$$

The tactic produces a goal labelled j on its output wire (bottom right of Fig. 1)

$$\begin{aligned}&\ldots\ (*\ ?\vdash\ *)^{\lceil}_{Z}(true \wedge \sigma \in STORE_C \wedge true)\\ &\qquad \wedge\ IntVal_C\ (mk\_signed\_int_C\ (IntOf_C\ fa_v\ +\ 1)\\ &\qquad\qquad ==_{CZ}\ mk\_signed\_int_C\ (IntOf_C\ fa_v\ +\ 1)) \neq IntVal_C\ 0^{\rceil}\end{aligned}$$

The goal types introduce a notion of 'types' to the tactic language, which improves upon the static properties for composition and evaluation found in the tactic language currently used to encode Supertac[6]. As can be seen in Fig. 1 (left), Tinker provides support for tactic developers to step through the proofs using 'interactive operations' to e.g. 'step over' or 'step into' a tactic. A novel breakpoint feature has recently been introduced [5], where sub-goals are evaluated automatically until a sub-goal reaches the breakpoint. At this point evaluation will stop, and the user can guide the evaluation step-by-step from this point onwards. This is particularly useful for large and complex tactics such as Supertac. We will return to how we have exploited these special Tinker features for this work in §3. We believe that this support for inspection and adaptations through simple graph visualization is novel.

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*, represented by boxes in the graph that contains sub-graphs. The architecture of Supertac consists of four sub-tactics executed in sequence, where the first of these has been decomposed in a hierarchical node. This is done by unfolding and breaking it, and its sub-tactics, down, and then re-compositing the sub-tactics in PSGraph with proper goal-types. In the future we plan to decompose the remaining three. Fig. 2 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.

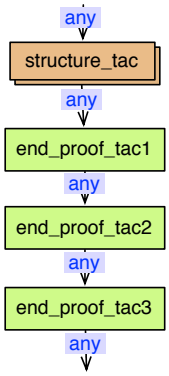

Fig. 2. Supertac

Fig. 3 shows the tactic nested by the structure_tac node in PSGraph. As a proof of concept, this encoding has been successfully applied to discharge VCs generated from a *Nose-Gear Velocity* case study [8] in ClawZ.

[6] Recently, several *typed* tactic language, e.g. Mtac [9], have been developed. They will have comparable static properties to PSGraph, albeit they do not have the dynamic inspection features Tinker provides.

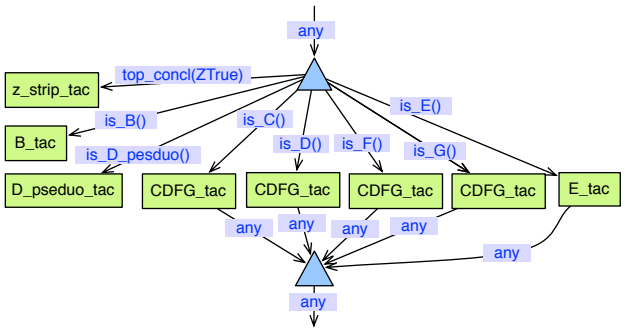

Fig. 3. The structure_tac sub-tactic of Supertac

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[7]. The remaining parts, e.g. the encoding of Simulink using Z Notation, are unchanged. The key challenge is thus to replace the parts of Supertac that reduce the Ada vocabulary with pure set theory into parts that reduce the C vocabulary into set theory (and develop these). This is non-trivial, because semantically variables in Ada can be treated simply when aliasing restrictions are in place, however the semantics of a variable in C come with a side condition stating that it is disjoint from other C objects. PSGraph enables users to step through evaluation and see how the goals evolve. This is a very useful when adapting a tactic in this way. To illustrate, the following VC has been generated from a simple C program:

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

With the debugging support of Tinker to interactively step through our PSGraph version of Supertac, we were able to pinpoint where in end_proof_tac1 it was assumed that the target programming language constructs need to be eliminated. First we split this sub-tactic into end_proof_tac1_1 and end_proof_tac1_2. The C specific parts will need to be eliminated between these two sub-tactics. To illustrate, after end_proof_tac1_1 our VC looks as follows:

$$\begin{array}{l} \lceil (*\ \ 3\ *)^{\lceil}_{\mathrm{Z}} \sigma \in STORE_C \rceil \\ \lceil (*\ \ 2\ *)^{\lceil}_{\mathrm{Z}} \langle (fa_v, fa_l) \rangle\ AllocatedIn\ \sigma \rceil \\ \lceil (*\ \ 1\ *)^{\lceil}_{\mathrm{Z}} clawz\_hint1\ \texttt{"}Supertac{:}VC\_Origin{:}Empty\_Block\_List\texttt{"} \rceil \end{array}$$

[7] A subset of C called C♭ is used. It has been designed with safety critical applications in mind. C♭'s formalisation in ProofPower is based on work by Norrish for the HOL system [6]. The formalisation is comparable to Frama-C, however it has not been designed to act as a framework for other analysis tools.

$$(* \ ?\vdash \ *)^{\sqsubset}_{Z}(true \land \sigma \in STORE_C \land true)$$
$$\land \ IntVal_C \ \ (IntOf_C \ (IntVal_C \ (mk\_signed\_int_C \ (IntOf_C \ fa_v \ + \ 1))))$$
$$==_{CZ} \ mk\_signed\_int_C \ (IntOf_C \ fa_v \ + \ 1)) \neq IntVal_C \ 0^{\urcorner}$$

where assumption *(* 1 *)* has been inserted by structure_tac to guide the rest of the proof. The intuition behind this VC is that the C value of comparing $IntOf_C(IntVal_C(mk\_signed\_int_C(IntOf_C fa_v + 1)))$ with $mk\_signed\_int_C(IntOf_C fa_v + 1)$, using the C version of integer comparison operator $==_{CZ}$, is not equal to the C value of 0. 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. 4 (left), we introduce a new nested tactic called qcz_conversion, and insert it 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.

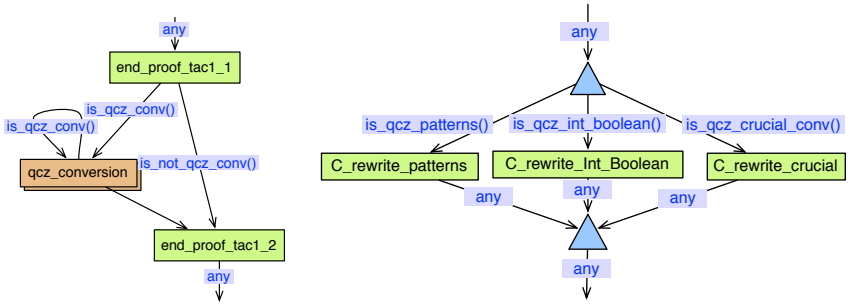

Fig. 4. New end_proof_tac1 (left) & codeonversion types (right)

The nested graph of this tactic is shown in Fig. 4 (right). 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 with special simplifications related to C booleans (represented as integer in C); while C_rewrite_crucial does the crucial steps in eliminating C vocabulary. To illustrate, the following are some of the *crucial* conversions of C_rewrite_crucial:

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

These changes were sufficient to complete the running example, 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.

Proofs are becoming more commonplace and increasingly complex. Like with software maintainence, *proof maintainence* is going to be a problem, as people who did the proof originally will have moved on or retired. For example, the substantial proof effort on the sel4 kernel [4] will be around for decades, and even small changes to the underlying kernel is likely to break many proofs. We have shown the advantages of PSGraph for the maintainence of Supertac; this experience has given us confidence that it has potential to play a supporting role in *proof engineering* [3] other large proof-based developments.

So far we have only decomposed structure_tac. In the medium term we aim to complete this work by decomposing the remaining tactics and we are hoping that PSGraph can be used to remove some of the "clutter" that has been the result of updating Supertac to new applications. Particular challenges will be to discover new goal types and find suitable graphical represention of some domain specific and non-trivial combinators used in Supertac.

In the long term we would like to re-implement the overall structure of Supertac from scratch, using the existing Supertac components as building blocks. Ideally, we will support both Ada and C. 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. We believe that this will give a much cleaner proof strategy that would be easier to analyse and adapt for future applications and changes to the modelling language or the target programming language.

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. In *UITP 2014*, volume 167 of *ENTCS*, pages 23–34. Open Publishing Association, 2014.
3. G Klein. Proof engineering considered essential. In *FM 2014: Formal Methods*, pages 16–21. Springer, 2014.
4. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.
5. Y. Lin, P. Le Bras, and G. Grov. Developing & debugging proof strategies by tinkering. In *TACAS 2016*, to appear.
6. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1999.
7. C. O'Halloran. Automated verification of code automatically generated from Simulink. *ASE*, 20(2):237–264, 2013.
8. C. O'Halloran. Nose-Gear Velocity–A challenge problem for software safety. Australian System Safety Conference (ASSC 2014), held in Melbourne 28-30 May, 2014.
9. B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25:e12, 2015.