# Integrating Model Checking and Theorem Proving in a Reflective Functional Language

Tom Melham

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, England
Tom.Melham@comlab.ox.ac.uk

**Abstract.** *Forte* is a formal verification system developed by Intel's Strategic CAD Labs for applications in hardware design and verification. Forte integrates model checking and theorem proving within a functional programming language, which both serves as an extensible specification language and allows the system to be scripted and customized. The latest version of this language, called *reFLect*, has quotation and antiquotation constructs that build and decompose expressions in the language itself. This provides combination of pattern-matching and reflection features tailored especially for the Forte approach to verification. This short paper is an abstract of an invited presentation given at the International Conference on Integrated Formal Methods in 2004, in which the philosophy and architecture of the Forte system are described and an account is given of the role of *reFLect* in the system.

## 1 The Forte Verification Environment

Forte [17] is a formal verification environment that has been very effective on large-scale, industrial hardware verification problems at Intel [10,11,12,15]. The Forte system combines several model checking and decision algorithms with lightweight theorem proving in higher-order logic. These reasoning tools are tightly integrated within a strongly-typed, higher-order functional programming language called FL. This allows the Forte environment to be customised and large proof efforts to be organized and scripted effectively. FL also serves as an expressive language for specifying hardware behaviour.

Model checking using symbolic trajectory evaluation ('STE') lies at the core of the Forte environment. STE [16] can be viewed as a hybrid between a symbolic simulator and a symbolic model checker. As a simulator, STE can compute symbolic expressions giving outputs as a function of arbitrary inputs. As a model checker, it can automatically check the validity of a simple temporal logic formula—computing an exact characterization of the region of disagreement if the formula is not unconditionally satisfied. These features provide a seamless connection between simulation and verification as well as excellent feedback on failed proof attempts—two key elements of an effective usage methodology for large-scale formal verification [10,17].

E. Boiten, J. Derrick, G. Smith (Eds.): IFM 2004, LNCS 2999, pp. 36–39, 2004.

STE is a particularly efficient model checking algorithm, in part because it has a very restricted temporal logic. But STE, like any model checker, still has very limited capacity. Forte therefore complements STE with a higher-order logic theorem prover of similar design to the HOL system [6]. Theorem proving bridges the gap between big, practically-important verification tasks and tractable model checking problems. The Forte philosophy is to have as thin a layer of theorem proving as possible, since using this technology is still difficult. But case studies have shown that a surprising amount of added value can be gained from even very simple (mathematically 'shallow') theorem proving.

The Forte approach is to tightly integrate model checking and theorem proving within the single framework of a functional programming language and its runtime system. A highly engineered implementation of STE is built into the core of the language, with many entry points provided as user-visible functions. Two key aspects of this architecture are that it is a 'white-box' integration of model checking and theorem proving and that functional programming plays a central role in scripting verification efforts.

## 2 The *reFLect* Functional Language

The successor to FL for future generations of Forte is a new functional language called *reFLect* [7]. The *reFLect* language is strongly typed and similar to ML [8], but has quotation and antiquotation constructs like those in LISP but in a typed setting. This provides combination of pattern-matching and reflection tailored especially for the Forte approach to verification. In what follows, a brief sketch is given of the motivation for the design of these features.

In higher-order logic theorem provers like HOL the logical 'object language' in which reasoning is done is embedded as a data-type in the (functional) meta-language used to control the reasoning. This makes the various term analysis and transformation functions required by a theorem prover straightforward to implement. But separating the object-language and meta-language also causes duplication and inefficiency. Many theorem provers, for example, need to include special code for efficient execution of object-language expressions [2,3].

In *reFLect*, the data-structure used by the underlying language implementation to represent syntax trees is made available as a data-type within the language itself. Functions on that data-structure, such as evaluation, are also made available. This approach retains all the term inspection and manipulation abilities of a conventional theorem prover while borrowing an efficient execution mechanism from the meta-language implementation.

It also builds *reflection* [9] into the logic of the theorem prover. In systems like HOL, higher order logic is constructed along the lines of Church's formulation of simple type theory [5], in which the logic is defined on top of the $\lambda$-calculus. Defining a logic on top of *reFLect* in the same way gives a higher-order logic that includes the *reFLect* reduction rules as well as certain reflection inference rules.

These reflection capabilities allow Forte to make a logically principled connection between theorems in higher order logic and the results of invoking a model

checker. A similar mechanism called *lifted-FL* [1] was available in earlier versions of Forte, but *reFLect* provides much richer possibilities. For example, one can use quantifiers to create a bookkeeping framework that cleanly separates logical content from model-checking control parameters.

In addition to serving as a meta-language for theorem proving, functional programming languages have often been used to describe the structure of hardware designs. Notable examples include work done in Haskell [4,14] and LISP [13]. A key capability exploited by such work is simulation of hardware designs by program execution. In Forte, however, we also wish to do various operations on the abstract syntax of models written in the language, as well as straight simulation. For example, we wish to implement and possibly even verify circuit design transformations [20]. *ReFLect* makes this a built-in part of the language.

The *reFLect* language can been seen as an application-specific contribution to the field of *meta-programming* [18]. Unlike most meta-programming systems, however, the target applications for *reFLect* in Forte give *intensional analysis* a primary role in the language. Its design is therefore somewhat different from staged functional languages like MetaML [21] and Template Haskell [19], which are aimed more at program generation and the control and optimization of evaluation.

**Acknowledgements.** I thank the organisers of the IFM 2004 for their kind invitation to speak at the conference in Canterbury. The Forte framework [17] and the *reFLect* language [7] are the result of many years of research and development by Intel's Strategic CAD Labs in Portland Oregon. The research reported in this paper was done in collaboration with Jim Grundy and John O'Leary at Intel, and builds on the Forte work of Carl Seger, Robert Jones, and Mark Aagaard.

## References

1. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, 'Lifted-fl: A Pragmatic Implementation of Combined Model Checking and Theorem Proving', in *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs 1999*, edited by Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, LNCS, vol. 1690 (Springer-Verlag, 1999), pp. 23–340.
2. B. Barras, 'Proving and Computing in HOL', in *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, edited by M. Aagaard and J. Harrison, LNCS, vol. 1869 (Springer-Verlag, 2000), pp. 17–37.
3. S. Berghofer and T. Nipkow, 'Executing higher order logic', in *Types for Proofs and Programs: International Workshop, TYPES 2000*, edited by P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, LNCS, vol. 2277 (Springer-Verlag, 2000), pp. 24–40.
4. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, 'Lava: Hardware design in Haskell', in *Functional Programming: International Conference, ICFP 1998*, (ACM Press, 1998), pp. 174–184.
5. A. Church, 'A Formulation of the Simple Theory of Types', *Journal of Symbolic Logic*, vol. 5 (1940), pp. 56–68.
6. M. J. C. Gordon and T. F. Melham (editors), *Introduction to HOL: A theorem proving environment for higher order logic* (Cambridge University Press, 1993).

7. J. Grundy, T. Melham, and J. O'Leary. 'A Reflective Functional Language for Hardware Design and Theorem Proving', Research Report PRG-RR-03-16, Programming Research Group, Oxford University (October, 2003).
8. R. Harper, D. MacQueen, and R. Milner, 'Standard ML', Report 86-2, University of Edinburgh, Laboratory for Foundations of Computer Science (1986).
9. J. Harrison, 'Metatheory and Reflection in Theorem Proving: A Survey and Critique', Technical Report CRC-053, SRI Cambridge (1995).
10. R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, 'Practical formal verification in microprocessor design' *IEEE Design & Test of Computers*, vol. 18, no. 4 (July/August, 2001), pp. 16–25.
11. R. Kaivola and K. R. Kohatsu, 'Proof engineering in the large: Formal verification of the Pentium-4 floating-point divider', in *Correct Hardware Design and Verification Methods: 11th Advanced Research Working Conference, CHARME 2001*, edited by T. Margaria and T. F. Melham, LNCS, vol. 2144 (Springer-Verlag, 2001), pp. 196–211.
12. R. Kaivola and N. Narasimhan, 'Formal verification of the Pentium-4 multiplier', in *High-Level Design Validation and Test: 6th International Workshop, HLDVT 2001* (IEEE Computer Society Press, 2001), pp. 115–122,
13. M. Kaufmann, P. Manolios, and J. S. Moore (editors), *Computer-Aided Reasoning: ACL2 Case Studies*, (Kluwer, 2000).
14. J. Matthews, B. Cook, and J. Launchbury, 'Microprocessor specification in Hawk' in *IEEE International Conference on Computer Languages*, (IEEE Computer Society Press, 1998), pp. 90–101.
15. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, 'Formally Verifying IEEE Compliance of Floating-Point Hardware', *Intel Technical Journal* (First quarter, 1999). Available at `developer.intel.com/technology/itj/`.
16. C.-J. H. Seger and R. E. Bryant, 'Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories', *Formal Methods in System Design*, vol. 6, no. 2 (March 1995), pp. 147–189.
17. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, M. D. Aagaard, C. Barrett, and D. Syme, 'An Industrially Effective Environment for Formal Hardware Verification'. Submitted for publication.
18. T. Sheard, 'Accomplishments and Research Challenges in Meta-Programming', in *Semantics, Applications, and Implementation of Program Generation: 2nd International Workshop, SAIG 2001*, edited by W. Taha, LNCS, vol. 2196 (Springer-Verlag, 2001), pp. 2–44.
19. T. Sheard and S. Peyton Jones, 'Template Meta-Programming for Haskell', in *ACM SIGPLAN Haskell Workshop*, edited by M. T. M. Chakravarty, (ACM Press, 2002), pp. 1–16.
20. G. Spirakis, 'Leading-edge and future design challenges: Is the classical EDA ready?', in *Design Automation: 40th ACM/IEEE Conference, DAC 2003* (ACM Press, 2003), p. 416.
21. W. Taha and T. Sheard, 'Multi-stage programming with explicit annotations', *SIGPLAN Notices*, vol. 32, no. 12 (2002), pp. 203–217.