# Towards verification via supercompilation

Alexei Lisitsa
Department of Computer Science
The University of Liverpool
Peach Street, Liverpool, U.K.
A.Lisitsa@csc.liv.ac.uk

Andrei Nemytykh*
Program Systems Institute
Pereslavl-Zalessky
152020, Russia
Nemytykh@math.botik.ru

## Abstract

*Supercompilation, or Supervised Compilation is a technique for program specialization, optimization and, more generally, program transformation. We present an idea to use supercompilation for verification of parameterized programs and protocols, present a case study and report on our initial experiments.*

## 1 Supercompilation and Verification

Verification of infinite-state or parameterized systems is challenging and, in general, an undecidable problem. The research in this area is still very active and taking various routes, including development of deductive methods and parameterized model checking. This abstract describes our ongoing work on developing a new alternative approach for reasoning about and verification of parameterized systems. The approach uses the program transformation methods known as *supercompilation*.

The supercompilation (from *super*vised *compilation*) is a powerful semantic based program transformation technique [6, 8] having a long history well back to the 1960-70s, when it was proposed by V. Turchin. The main idea of supercompilation may be explained as follows.

Let $P$ be a functional program implementing a partial function $f_P : D^k \to D$, where $D$ is some set of data. A supercompiler observes the behaviour of $P$ running on (partially defined) input with the aim to define, based solely on the observations, a program $\bar{P}$, which would be equivalent to the original one (on the domain of latter), but having improved properties. The result may be a specialized version of the original program, taking into account the properties of known arguments, or just a reformulation, and sometimes more efficient equivalent program. The resulting program $\bar{P}$ is build as the result of metainterpretation of $P$, so $\bar{P}$ and $P$ may have nothing common in syntax. As to the semantics, the result of supercompilation $\bar{P}$ implements, in general, an extension[1] of $f_P$, that is a function $f_{\bar{P}} : D^k \to D$ such that $\forall d \in dom(f_P)\ \ f_P(d) = f_{\bar{P}}(d)$.

From the very beginning the development of supercompilation has been conducted mainly in the context of the Refal programming language [7]. A few supercompilers for Refal were implemented with the most advanced being SCP4 [5].

Our idea here, to use supercompilation for the purpose of verification, came from the numerous observations on the results of supercompilation. It is often the case that transformed program is much simpler than original one, and moreover, the attempts to execute a parameterized program[2], undertaken by a supercompiler reveal deep properties of the computation of this program, otherwise hidden in the original syntax. Bearing this in mind we came up with the following idea for verification via supercompilation:

> *Take a program/protocol/system to be verified and encode it as a parameterized functional program. Then apply supercompilation in order to get a transformed program for which the correctness condition can be easily checked.*

In our experiments we have used a particular scheme, which essentially implements a *parameterized testing*. That is, the above parameterized functional program

- first *models the execution of a system (protocol) for $n$ steps* (where $n$ is an input parameter), and
- then *checks the correctnes condition*.

For any concrete value of $n$ the program tests the correctness condition on the $n$-th step of the execution of the system.

* visiting Department of Computer Science, the University of Liverpool, supported by the Research Development Fund, grant RDF 4416

[1] this distinguishes it from other popular program transformation techniques, such as partial evaluation [3]

[2] we use somewhat ambiguous, but shorter term *parameterized program* for denoting a program with the parameterized entry point

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

Supercompilation works on programs with parameterized entry points and analyses behaviour of programs for *all* possible values of input parameters (in our case the parameter is an *arbitrary* value of the $n$). The output of supercompilation can be used then to verify safety conditions.

Using the programs with parameterized entry points allows also to deal with non-deterministic systems. We model non-determinism by providing another input parameter, taking strings of characters (possible choices) as values.

## 2 Case study

As an example we consider here the verification of the parameterized cache coherence protocol MESI. Following [2], it is formally modelled as a parameterized family of identical finite state automata (with four states in this case: *Modified*, *Exclusive*, *Shared* and *Invalid*)). All transitions of the automata are labelled either by action labels *R* (for *Read*), or *W* (for *Write*), or by compelementary reaction labels $\bar{R}$ or $\bar{W}$. The process of the protocol execution may be described as follows. At every step one automaton is chosen nondeterministically to perform some action $a$, all other automata make a complementary transition (reaction) $\bar{a}$.

To verify the parameterized version of the protocol one has to prove the following two correctness conditions (safety for data consistency). No matter how many automata is in the system, if all of them are in the state *Invalid* at the initial moment of time then

- two automata cannot be in the states *Modifed* and *Shared*, respectively, at the same time;
- no two automata cannot both be in the *Modified* state at the same time.

We encode this protocol as a Refal program [4], implementing parameterized testing of the above safety conditions. The prorgam has two input parameters $e.A$ and $e.I$. We assume non-deterministic execution of the protocol and the characters of the string value of $e.A$ represent possible choices at branching points. So the length of a string value of $e.A$ is the number of steps of the protocol the program is going to execute. The characters of a string value of $e.I$ represent automata and its length is the number of automata in the system. The program consists of definitions of four functions: *Go* (entry point of the program), *Loop* (the main loop of the protocol execution), *RandomAction* (modelling one-step execution) and *Result* (testing correctness conditions). See the text of the program with further explanations in [4].

The program may return either **False** or **True** depending on the results of testing. Correctness of the protocol, however, is equivalent to the fact that *the program never returns* **False**, no matter what values are given to the input parameters.

The result of supercompilation of the above program is a Refal program which *does not contain (syntactically)* the value **False**. This implies the resulting program never returns **False**, which in turn, implies the required property for the original program. Correctness of the protocol follows immediately. The technical details may be found in [4].

## 3 Experiments and Discussion

We have verified in this way all parameterized cache coherence protocols from [1, 2]. These include *SynapseN+1*, *MSI*, *MESI*, *Illinois*, *Berkley*, *Dragon*, *Firefly* and *Futurebus+* protocols. For *Futurebus+* and *Dragon* it takes around 3 seconds, and for all others less than 1 second to verify on the Pentium III 1.2 Ghz, running Windows XP and supercompiler SCP4.

All technical details of these experiments as well as the information about further developments are available at [4].

The experiments have revealed the power of this non-traditional application of supercompilation for verification purposes. We plan to continue development of corresponding techniques both experimentally and theoretically, providing in particular a full formal account of the correctness of the program transformations involved.

## References


[1] G. Delzanno. Verification of consistency protocols via infinite-state symbolic model checking, a case study. In *Proceedings of FORTE/PSTV*, pages 171–188, 200.

[2] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):279–295, 2003.

[3] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[4] A. Lisitsa and A. Nemytykh. Verification via supercompilation. http://www.csc.liv.ac.uk/~alexei/VeriSuper/, 2005.

[5] A. Nemytykh and V. Turchin. The supercompiler scp4: sources, on-line demonstration. http://www.botik.ru/pub/local/scp/refal5/~2000.

[6] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8:292–325, 1986.

[7] V. Turchin. *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, 1989. Electronic version: http://www.botik.ru/pub/local/scp/refal5/~2000.

[8] V. Turchin. Supercompilation: Techniques and results. In *Proceedings of Perspectives of System Informatics*, volume 1181 of *LNCS*, pages 227–248. Springer-Verlag, 1996.