International Journal of Foundations of Computer Science
Vol. 19, No. 4 (2008) 953–969

World Scientific
www.worldscientific.com

REACHABILITY ANALYSIS IN VERIFICATION VIA SUPERCOMPILATION

ALEXEI LISITSA*
Department of Computer Science, The University of Liverpool, Ashton Building, Liverpool L69 3BX, United Kingdom
A.Lisitsa@csc.liv.ac.uk

and

ANDREI P. NEMYTYKH
Program Systems Institute, Russian Academy of Sciences, Pereslavl-Zalessky, Yaroslavl region 152020, Russia
Nemytykh@math.botik.ru

Received 5 December 2007
Accepted 3 June 2008
Communicated by V. Halava and I. Potapov

We present an approach to verification of parameterized systems, which is based on program transformation technique known as supercompilation. In this approach the statements about safety properties of a system to be verified are translated into the statements about properties of the program that simulates and tests the system. Supercompilation is used then to establish the required properties of the program. In this paper we show that reachability analysis performed by supercompilation can be seen as the proof of a correctness condition by induction. We formulate suitable induction principles and proof strategies and illustrate their use by examples of verification of parameterized protocols.

Keywords: Program verification; cache coherence protocols; program specialization; supercompilation.

1. Introduction

The verification of infinite-state or parameterized problems is, in general, an undecidable problem. The research in this area is focused on finding restricted classes of problems, for which verification is decidable and the development of efficient verification procedures for practical applications. The research is active and taking different routes [1, 2, 3, 8, 10]. But still many practically interesting verification problems lie outside the scope of existing automated verification methods and further development of these methods is required.

*Corresponding author.

One of the recent interesting and promising directions for tackling infinite-state, or parameterized, verification is to apply the methods developed in the area of *program transformation* and *metaprogramming*, and in particular, *program specialization* [12, 13, 20, 21, 36].

In this paper we are interested in one particular approach in program transformation and specialization, known as supercompilation[a]. *Supercompilation* [41] has not drawn much attention yet in the context of verification, although it has been mentioned in [20, 21] as potentially applicable here. Supercompilation is a powerful semantic based program transformation technique [39, 41, 44] having a long history well back to the 1970s, when it was proposed by V. Turchin. The main idea behind a supercompiler is to observe the behavior of a functional program P running on *partially* defined input with the aim to define a program, which would be equivalent to the original one (on the domain of latter), but having improved properties. The supercompiler unfolds a potentially infinite tree of all possible computations of a parameterized program. In the process, it reduces the redundancy that could be present in the original program. It folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of the computing system. And, finally, it analyses global properties of the graph and specializes this graph with respect to these properties (without an additional unfolding). The resulting definition is constructed solely based on the meta-interpretation of the source program rather than by a (step-by-step) transformation of the program.

The result of supercompilation may be a specialized version of the original program, taking into account the properties of partially known arguments, or just a re-formulated, and sometimes more efficient, equivalent program (on the domain of the original) [16].

Turchin's ideas have been studied by a number of authors for a long time and have, to some extent, been brought to the algorithmic and implementation stage [34]. From the very beginning the development of supercompilation has been conducted mainly in the context of the Refal programming language [31, 43], another creation of V.Turchin. A number of the simplest model supercompilers for subsets of LISP-like languages were implemented as well with an aim to formalize some aspects of the supercompilation algorithms [38, 40, 39]. The most advanced supercompiler for Refal is SCP4 [29, 30, 31, 32, 33, 34]. In this paper we formulate and study a simplest theoretical model of supercompilation, which is relevant to verification of safety properties of broadcast protocols.

In [23, 24, 25, 26] we proposed to use supercompilation for verification of parameterized systems using a particular scheme of *parameterized testing*. Using this scheme we translate the statements about safety properties of a system to be verified into the statements about properties of the program that *simulates and tests* the system. Supercompilation is used then to establish the required properties of the program. We have conducted series of experiments on verification of parameterized *cache coherence protocols* and successfully verified [24] all cache coherence protocols presented in [5] and [7]. We have also verified in this way parameterized

[a]from *super*vised *compilation*

Java MetaLock algorithm and series of Petri Nets models. This work started mainly as an experiment driven one and the approach proved to be empirically successful. This left however the questions on its correctness and completeness for classes of verification problems open. In this paper we address the issue of correctness of the proposed method. We develop a formal model, which renders supercompilation process in the particular context of parameterized testing as a reachability analysis for term-rewriting systems by means of inductive proofs of safety properties. This establishes the correctness of the method.

Further we illustrate our method by verification of the parameterized MOESI protocol [5]. Interestingly enough, using supercompilation to perform parameterized testing allows not only to verify the protocol but also to discover new facts about the protocol. The facts are formulated by automatic generalization of configurations – one of the tools of supercompilation. In particular, an analysis of the supercompilation trace shows that the protocol is correct with more general assumptions on the initial state than reported in [5].

Notice, that parameterized MOESI cache coherence protocol is specified here as a broadcast protocol. It was shown in [11] that verification of safety properties of parameterized broadcast protocols is decidable and algorithms to tackle such problems by symbolic model checking based on constraint analysis have been developed e.g. in [5, 6, 7]. In this paper we confine ourselves with establishing the correctness of our approach, leaving detailed comparisons with algorithms presented in [11, 5, 6, 7] for further work.

The paper is organized as follows. In the next section we give general description of our *verification via parameterized testing* approach in language-independent terms. Section 3 presents the formal model and correctness result.

Then in section 4 we introduce some of the strategies leading to successful verification of a class of parameterized cache coherence protocols. In section 5 we present a free monoid of terms and specify the strategies of the supercompiler SCP4 in its terms. The size limit of the paper does not allow us to present a detailed verification of the MOESI protocol using these strategies. For detailed exposition we refer the reader to Appendix [27], which we put on an WWW page as a separate document.

2. Parameterized Testing

In this section we describe our general technique for the verification of parameterized systems. The technique is based on the translation of the statements about *safety* properties of a system to be verified into the statements about properties of the program that *simulates and tests* the system.

The scheme works as follows. Let S be a parameterized system (a protocol) and we would like to establish some safety property P of S. We write a program φ_S simulating execution of S for n steps, where n is an input parameter. Let the n be given in the unary system, as a string of characters. If the system is nondeterministic, we label each step with an action, whose value is assumed to be chosen at the branching point of execution, e.g. it may be a character labelling the choice. Thus, we assume that given the values of input parameter n and an initial state of

S, the program φ_S returns the state of the system S after the execution of n steps of the system, following the choices provided by the labels of the steps. Let $T_P(\_)$ be a testing program, which given a state s of S returns the result of testing the property P on s (*True* or *False*). Consider a composition $T_P \circ \varphi_S$. This program first simulates the execution of the system and then tests the property required. Assume both programs terminate. Now the statement "the safety property P holds in any possible state reachable by the execution of the system S from an initial state" is equivalent to the statement "the program $T_P(\varphi_S(n))$ never returns the value *False*, no matter what values are given to the input parameter".

In practical implementation of the scheme we use a functional programming language Refal to implement the program $T_P(\varphi_S(n))$ and an optimizer SCP4 (a supercompiler) to transform the program to a form, from which one can easily establish the required property.

The idea of using testing and supercompilation for the verification purposes is not new. In the classical paper [41] V.Turchin writes: *Proving the correctness of a program is theorem proving, so a supercompiler can be relevant. For example, if we want to check that the output of a function $F(x)$ always has the property $P(x)$, we can try to transform the function $P(F(x))$ into an identical T.* The idea has not been tried until recently for the problems interesting for verification community. Our experiments have shown that indeed, the idea is viable and can be adopted for non-trivial verifications problems for parameterized distributed systems.

3. Correctness Issue and Formal Model

One of the immediate questions posed by almost everyone seeing the approach in work for the first time is "Is this correct at all? Why should I believe your claims about verification?"

Firstly, one can argue as follows. It has been shown, in particular in [37, 38, 40] that (variants of) supercompilation is a correct transformation, in a sense it always returns (if any) the program equivalent to the input program (on the domain of latter). Then we repeat the argument from Section 2. We should note, in this respect, that SCP4 [29, 30, 34] is a large program dealing with the concrete functional language Refal, which has specific semantic assumptions, like built-in associativity of concatenation as a term forming construct.

Furthermore, the SCP4 supercompiler is the result of more than two decades of development and is a highly optimized program, implementing different strategies which can be tailored by the user to the particular cases. Proving the correctness of the whole SCP4 is far from being trivial and is, actually, not very much relevant to our experiments. Even if we accept the correctness, it does not explain *why* supercompilation works for establishing correctness properties.

We address these issues in the present paper by developing a formal model, which renders supercompilation process (in the case of verification tasks) as an inductive proof of safety properties. That establishes the correctness of the method. The formal model is a very simplified and refined theoretical version of SCP4, which, nevertheless, is sufficient for verification of a class of (parameterized) cache

coherence protocols.

In this paper we confine ourselves by the claim that supercompiler SCP4 indeed implements the formal model we present. We provide some relevant comments but detailed discussion of the claim lies outside the scope of this paper. The model is formulated in terms of term rewriting systems.

3.1. *Term rewriting systems and safety properties*

Let $\mathcal{V}$ be a denumerable set of symbols for variables and $\mathcal{F} = \cup_i \mathcal{F}_i$ be a finite set of functional symbols, here $\mathcal{F}_i$ is a set of functional symbols of arity i. Let $\mathcal{T}(\mathcal{V}, \mathcal{F})$ be a free algebra of all terms built with variables from $\mathcal{V}$ and functional symbols from $\mathcal{F}$ in a usual way. Let every $\mathcal{F}_i$ be divided into disjoint sets $\mathcal{F}_i = \mathcal{F}n_i \cup \mathcal{C}_i$. We refer to $\mathcal{F}n_i$ as function names and to $\mathcal{C}_i$ as constructor names. Let $\mathcal{C} = \cup_i \mathcal{C}_i$. A term without function names is passive. Let $\mathcal{G}(\mathcal{T}) \subset \mathcal{T}(\mathcal{V}, \mathcal{F})$ be the set of ground terms, i.e. terms without variables. Let $\mathcal{O}(\mathcal{T}) \subset \mathcal{G}(\mathcal{T})$ be the set of object terms, i.e. ground passive terms. For a term t we denote the set of all variables in t by $V(t)$.

A substitution is a mapping $\theta : \mathcal{V} \to \mathcal{T}(\mathcal{V}, \mathcal{F})$. A substitution can be extended to act on all terms homomorphically. A substitution is called *ground*, *object*, or *strict* iff its range is a subset of $\mathcal{G}(\mathcal{T})$, $\mathcal{O}(\mathcal{T})$ or $\mathcal{T}(\mathcal{V}, \mathcal{C})$ (i.e. passive terms), respectively. We use notation $s = t\theta$ for $s = \theta(t)$, call s an *instance* of t and denote this fact by $s \ll t$.

A term-rewriting system is a pair $P = \langle t, R \rangle$, where t is a term called *initial* and R is a finite set of rules of the form $f(p_1, \ldots, p_k) \to r$, where $f \in \mathcal{F}n_k$, $\forall i\ (p_i \in \mathcal{T}(\mathcal{V}, \mathcal{C}))$[b], $r \in \mathcal{T}(\mathcal{V}, \mathcal{F})$, $V(r) \subseteq V(f(p_1, \ldots, p_k))$.

Given a set of the rules R define one-step transition relation $\Rightarrow_R \subseteq \mathcal{T}(\mathcal{V}, \mathcal{F}) \times \mathcal{T}(\mathcal{V}, \mathcal{F})$ as follows: $t_1 \Rightarrow_R t_2$ holds iff there exist a strict substitution θ and a rule $(l \to r) \in R$ such that $t_1 = l\theta$ and $t_2 = r\theta$. *Reachability* relation $\Rightarrow_R^*$ is a *transitive* and *reflexive* closure of $\Rightarrow_R$. Notice, that any term reachable from a ground term is also ground.

Definition 1 *A binary relation $\Rightarrow$ on a set $\mathcal{T}$ is terminating (or well-founded) if there exists no infinite chain $t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow \ldots$ of elements of $\mathcal{T}$.*

Henceforth we assume that *transitive* closure $\Rightarrow_R^+$ of the restriction of $\Rightarrow_R$ on $\mathcal{G}(\mathcal{T}) \times \mathcal{G}(\mathcal{T})$ is terminating.

An arbitrary subset Q of $\mathcal{O}(\mathcal{T})$ is called a *property*. Let q be a finite set (*collection*) of passive terms. A property Q_q defined by q is a set of all object instances of all terms from q, that is $Q_q = \{\tau \mid \tau \in \mathcal{O}(\mathcal{T}) \wedge \exists \rho \in q\ (\tau \ll \rho)\}$

We consider the following *reachability* problem for term-rewriting systems.

Verification of safety property
Given: *A term-rewriting system $P = \langle t, R \rangle$ and a property Q_q.*

[b]For simplicity we use only such kind of term-rewriting systems.

Question: *Is it true that all passive terms reachable in P from any ground instance of t do not satisfy the property Q_q? In formal notation, is the statement*

$$\forall s \in \mathcal{O}(\mathcal{T})(\forall t' \in \mathcal{O}(\mathcal{T})((t' \ll t) \wedge (t' \Rightarrow_R^* s)) \rightarrow s \notin Q_q)$$

true?

Many interesting verification problems for parameterized systems may be reduced to the above problem. See Appendix [27] for an example. In the next subsection we present a method suitable for solving such a problem and demonstrate its correctness.

3.2. *Inductive proofs of safety properties*

Consider an instance of the above verification problem $I = (\langle t, R\rangle, Q_q)$.

The proof of the safety property for I can be established by constructing successful *proof attempt* which consists of a sequence of directed rooted trees. Vertices of the trees will be labeled by terms. For a vertex a denote by t_a the term labeling a.

We assume that we have a testing procedure, which given a vertex a checks whether *all ground instances* of the t_a *do not satisfy* the property Q_q.

Another assumption is that any vertex in a tree is labelled as *unready*, *open* or *closed* (one flag per vertex) and all generated vertices are unready until they are explicitly open. We assume also that when any vertex of the proof tree labeled by a passive term is generated it is immediately tested. If the testing produces the negative result the whole proof tree building procedure stops and returns the answer NO to the verification problem, otherwise the vertex is closed.

Given a directed tree T and its edge $(a \xrightarrow{e} b)$, we say the vertex a is the *parent* of b and b is a *child* of a. A vertex a_1 is an ancestor of a vertex a_n if there exists a sequence of edges of T such that $(a_1 \xrightarrow{e} a_2), (a_2 \xrightarrow{e} a_3), \ldots, (a_{n-1} \xrightarrow{e} a_n)$.

Definition 2 *For a given $I = (\langle t, R\rangle, Q_q)$ a* proof attempt *is a sequence of directed rooted trees $T_0, T_1, \ldots$ such that T_0 is generated by the START rule, and T_{i+1} is obtained from T_i by application of one of the following rules: UNFOLD, CLOSE I, CLOSE II, GENERALIZE.*

The proof rules are defined as follows

START Create a root of the tree, label it by the initial term t of the term rewriting system.

UNFOLD Choose any of the unready vertices a with the labeling term t_a and generate all terms $t_1, \ldots t_n$ such $t_a \Rightarrow_R t_i$. For every such t_i create a child vertex a_i for a and put $t_{a_i} = t_i$. Open the vertex a. If the parent of a is open, then close the parent.

CLOSE I Choose any of the open vertices a and check whether there is a closed vertex b, such that $t_a \ll t_b$. If yes close the vertex a and delete its children. If there is no such b do nothing.

CLOSE II Choose any of the open vertices a and check whether all its children are closed there. If yes close the vertex a.

GENERALIZE Choose any of the open vertices a and any vertex b, ancestor of the a in the tree. Generate a term τ such that both $t_b \ll \tau$ and $t_a \ll \tau$ hold.

I.e. there exist two subsitutions θ_a and θ_b such that $t_a = \tau\theta_a$, $t_b = \tau\theta_b$. Delete the subtree with the root b, except the vertex b itself. Replace the label t_b with τ. Mark the vertex b as unready. The generalization is *strict*, iff the both θ_a and θ_b are strict.

Significance of the *unready* flag is related to effectiveness issue and will be considered in the section 4.

Definition 3 *A* proof *of an I is a finite proof attempt $T_0, \ldots, T_n$ for I such that all vertices in T_n are closed. A proof using only strict generalizations is called* strict.

Let R be a term rewriting system, $t \in \mathcal{T}(\mathcal{V}, \mathcal{F})$ and t_0 be a ground instance of the term t. Let $\bar{t_0} = t_0 \Rightarrow_R t_1 \Rightarrow_R t_2 \ldots \Rightarrow_R t_l$ be an arbitrary sequence of terms derived from t_0 by application of rules from R. Denote by $\mathcal{R}(t)$ the set of all passive terms reachable in R from any ground instance of t and $\mathcal{CR}(t)$ the set of all the sequences $\bar{g_0}$ such that g_0 is a ground instance of t and $g_{|\bar{g_0}|-1} \in \mathcal{R}(t)$. Here $|\bar{g_0}|$ is the length of the $\bar{g_0}$. The following proposition is trivial.

Proposition 1 *Let t be a term and τ be a term such that $t \ll \tau$, then $\mathcal{CR}(t) \subset \mathcal{CR}(\tau)$ holds. (And hence $\mathcal{R}(t) \subset \mathcal{R}(\tau)$.)*

Theorem 1 *For an instance $I = (\langle t, R\rangle, Q_q)$ of the verification problem above if there is a proof for I such that for any vertex a from the last tree T_N of the proof closed by the application CLOSE I rule the tree T_N contains the vertex closing the a, then the answer for this I is YES.*

Proof Let $T_0, \ldots, T_N$ be a proof for I.

The termination of the *transitive* closure $\Rightarrow_R^+$ of the restriction of $\Rightarrow_R$ on $\mathcal{G}(\mathcal{T}) \times \mathcal{G}(\mathcal{T})$ is by Definition 1 of Section 3.1. The statement of the theorem follows from the following statement. Let t_0 be a ground instance of the initial term t. Let $\bar{t} = t_0 \Rightarrow_R t_1 \Rightarrow_R t_2 \ldots \Rightarrow_R t_{l-1}$ be an arbitrary sequence of terms derived from t_0 by application of rules from R. (There is no such an infinite sequence.) Then every t_i is an instance of a term t_{a_i} for some vertex a_i of T. The proof of the statement is by induction on the length of the sequence. Let $|\bar{t}| = 1$, that is $\bar{t} = t_0$ By construction of the proof attempt $T_0, \ldots, T_N$ the label τ of the initial vertex of T_N is (possibly generalized several times) term t, i.e. we have $t \ll \tau$ and therefore $t_0 \ll \tau$. Notice that once a term labelling some vertex is generated it may be generalized several times later in the proof attempt by application of GENERALIZE rule.

Consider now the step of induction. Assume the statement for all sequences of the length up to some l and let $\bar{t} = t_0 \Rightarrow_R t_1 \Rightarrow_R t_2 \ldots \Rightarrow_R t_l$. By induction hypothesis we have $t_{l-1} \ll t_a$ for some vertex $a \in T_N$. Then two cases are possible.

If there are some children $a_1, \ldots, a_k$ of a in T_N then there exists some child a_j of a such that t_{a_j} is (possibly generalization of) the term t_l. (By the semantics of UNFOLD rule.) It follows then $t_l \ll t_{a_j}$.

If there are no children of a in T_N then t_a should be active, otherwise there could not be any term t_l such that $t_{l-1} \ll t_a$ and $t_{l-1} \Rightarrow_R t_l$. Moreover in that case the vertex a is closed by the application CLOSE I rule and there should be another vertex $b \in T_N$ such that $t_a \ll t_b$. If b has some children we repeat the argument for the previous case taking vertex b instead of a. If it does not we find yet another vertex c such that $t_a \ll t_b \ll t_c$ and repeat the argument for c. Notice that there is no more than finitely many vertices in T_N and if a vertex a is being closed by another vertex b, then b has to be closed itself, so after finitely many steps this case is reduced to the previous one. Step of induction is proved.

It follows that any ground passive term derivable from t_0 is an instance of one of the passive terms in T_N. Since all reachable passive vertices in T_N are tested the statement of the theorem follows. □

Example 1 *Let* $f \in \mathcal{F}n_2$, $A, B \in \mathcal{C}_1$, $x, y, x_i, y_i \in \mathcal{V}$. *Consider* $I = (\langle t, R\rangle, Q_q)$. *Here* R *is:*
$f(B(x), y) = f(x, B(y))$;
$f(A(A(x)), y) = f(A(x), B(y))$;
$f(A(B(x)), y) = y$;
q *contains the only term* $A(x)$ *and* $t = f(B(x_1), y_1)$.

Let $[\tau]$ *be a vertex labelled with a term* τ. *We denote each closed vertex as* $[\tau]^c$, *each open vertex as* $[\tau]^o$ *and each unready vertex as* $[\tau]^u$. *The first proof attempt is successful: START rule gives the tree* T_0 *containing the only vertex* $a^u = [f(B(x_1), y_1)]^u$, *UNFOLD rule yields* $T_1 = \{a^o, b^u = [f(x_2, B(y_1))]^u, (a^o \xrightarrow{e} b^u)\}$; *after the second UNFOLD we have* $T_2 = \{a^c, b^o, d_1^u = [f(x_3, B(B(y_1)))]^u, d_2^u = [f(A(x_4), B(B(y_1)))]^u, d_3^u = [B(y_1)]^c, (a^c \xrightarrow{e} b^o), (b^o \xrightarrow{e} d_1^u), (b^o \xrightarrow{e} d_2^u), (b^o \xrightarrow{e} d_3^u)\}$; *now two applications of UNFOLD rule open* d_1^u, d_2^u *and close* b^o; *two applications of CLOSE rule close* d_1^o *and* d_2^o *with* b^c *as the witness. We have* $T_6 = \{a^c, b^c, d_1^c, d_2^c, d_3^c, (a^c \xrightarrow{e} b^c), (b^c \xrightarrow{e} d_1^c), (b^c \xrightarrow{e} d_2^c), (b^c \xrightarrow{e} d_3^c)\}$, *where all vertices are closed.*

The second proof attempt fails: T_0, T_1, T_2 *are the same as in the first attempt; GENERALIZE gives* $T_3 = \{g^u = [f(x_3, y_3)]^u\}$. *Now it is easy to see this attempt does not terminate.*

The third proof attempt fails: T_0, T_1, T_2 *are the same as in the first attempt; GENERALIZE gives* $T_3 = \{g^u = [x_3]^u\}$. *Now* g^u *can never be closed.*

Theorem 2 *For an instance* $I = (\langle t, R\rangle, Q_q)$ *of the verification problem above if there is a strict proof for* I *then the answer for this* I *is YES.*

Proof Fix a rewriting system R and let t be any term. Consider the proof $T_0, \ldots, T_n$. Let τ_i be initial term from T_i, then for any $0 \leq i < n$ $\tau_i \ll \tau_{i+1}$.

Since strict generalization does not transform active terms to passive terms we have for any $0 \leq i < n$ $\mathcal{R}(\tau_i) \subseteq \mathcal{R}(\tau_{i+1})$.

All vertices are closed in T_n. All passive vertices are closed and tested in T_n and $\mathcal{R}(\tau_0) \subseteq \mathcal{R}(\tau_n)$. That proves the statement of the theorem. □

We show now that, in fact, the proof sequence is a compact representation of the inductive proof of the correctness condition (none of the ground instances of the

reachable passive terms has the property Q_q). First, we formulate the induction scheme in general terms.

Let $\triangleright$ be a well-founded partial ordering on a set $\mathcal{K}$. Let $\mathcal{M}$ is the set of all minimal elements of $\mathcal{K}$: $\mathcal{M} = \{t \in \mathcal{K} \mid \neg\exists(\tau \in \mathcal{K}).(\tau \neq t) \wedge (t \triangleright \tau)\}$. Note that $\mathcal{M}$ is not empty. Let Q be a predicate on $\mathcal{K}$ and $\mathcal{S}$ be a subset of $\mathcal{K}$. The following induction scheme can be used then to prove that Q holds everywhere on $\mathcal{K}$ (we assume $y \triangleleft x \equiv x \triangleright y$ here):

$$\frac{(\forall t \in \mathcal{M}.Q(t)) \wedge (\forall x \in \mathcal{K}.(\forall y \in \mathcal{S}.y \triangleleft x \rightarrow Q(y)) \rightarrow Q(x))}{\forall x \in \mathcal{K}.Q(x)}$$

Retuning to our context, let $\mathcal{L}$ be the set of the terms generated by applications of GENERALIZE rule during the proof given above and t, where t is the initial term of the I. Let H_g be the following hypothesis: "none of the ground instances of $g \in \mathcal{L}$ reaches a passive term having the property Q_q".

Then the proof given by the successful proof attempt can be considered as simultaneous proofs of all hypotheses H_g, such that each of them follows the inductive scheme given above and moreover the proofs may refer one to another. Here $\mathcal{K} = \mathcal{O}(\mathcal{T})$, $\triangleright$ is $\Rightarrow_R^+$, $Q(t) = H_t$, $\mathcal{M}_g$ is the set of all passive object terms reachable from all ground instances of g, and $\mathcal{S}_g$ is the set of the ground instances of the terms closed during applications of CLOSE I rule. The subscript g indicates the concrete proof of H_g.

4. Towards Effectiveness

The proof procedure presented in the previous section is non-determinstic. That leads to necessity of development of deterministic proof strategies which would be complete and/or efficient for classes of verification problems. In this section we make first steps towards resolving these largely open issues, and present the strategies which empirically has turned out to be sufficient for (practically efficient) proofs of correctness of cache coherence protocols [5].

The second proof attempt given in the Example 1 demonstrates that crucial information may be lost during an application of GENERALIZE rule. The information guaranteed transformation of the initial term uniformly on the values of the parameters. The start vertex is not a branching point: there exists the only edge outgoing from the vertex. Terminating of $\Rightarrow_R^+$ (see section 3.1) means there cannot be an infinite sequence of such kind of vertices one after another. Thus it is desirable to exclude such vertices from generalization.

Definition 4 *An open or closed vertex b is* pivot *in a tree T_j iff b has at least two outgoing edges.*

Henceforth we impose the following restriction on the strategy of rule applications: both CLOSE I and GENERALIZE rules choose only pivot vertices.

Given two terms t_1 and t_2 there can be a number of different generalizations, see example 1 for the illustration. Aiming to preserve as much as possible the structure of the terms, we impose the next restriction on GENERALIZATION rule: *result of generalization of any two terms t_1 and t_2 should be most specific*

term τ, meaning both $t_1 \ll \tau$ and $t_2 \ll \tau$ hold and for any other term ξ such that $(t_1 \ll \xi) \wedge (t_2 \ll \xi \ll \tau)$ implies that ξ equals to τ modulo variable's names.

Further restriction is concerned with the choice of terms to be generalized. In order to preserve the structure of terms it is natural and desirable to generalize only terms, which are similar (in a sense) one to another. There is delicate trade-off here. Informally, the fewer applications of GENERALIZE rule happened during a proof attempt the less information on the terms structure is lost and more chances to close the passive vertices. On the other hand, to close active vertices one may need more applications of GENERALIZE rule. The following criteria based on well-quasi-ordering have turned out to be empirically successful.

A *quasi-ordering* is any reflexive and transitive binary relation.

Definition 5 *A quasi-ordering* $\preceq$ *on a set* T *is a* well-quasi-ordering *if every infinite sequence* $t_1, t_2, \ldots$ *of elements of* T *contains* t_i, t_j $(i < j)$ *such that* $t_i \preceq t_j$.

Given a well-quasi-ordering $\preceq$ on $\mathcal{T}(\mathcal{V}, \mathcal{F})$, we specify the strategy choosing the vertices by GENERALIZE rule as follows: *choose any of the pivot open vertices* a *and any pivot vertex* b*, ancestor of the* a *in the tree such that* $t_b \preceq t_a$*; if there exists no such* a b *do nothing.*

Further, there can be a number of such vertices b. Intuitively, the closer a vertex b to the vertex a (among all its ancestors) the closer any ground instance of the b to a passive ground term terminating evaluation of the instance by the term-rewriting system. So we add to the above generalization strategy the requirement *to choose the closest such a vertex* b.

All our experiments verifying the class parameterized protocols [5] were successful both under lazy (call by need) and under applicative (call by value) strategies developing the stack of functions. For simplicity we selected the applicative strategy to demonstrate the main example given in Appendix[27][c].

5. A Free Monoid of Terms

In this section we consider a free monoid of terms, which was actually used in our experiments. Using this data structure and concepts and strategies given above allows to obtain automatic proofs of correctness of cache coherence protocols from [5]. See also remarks in Section 6.

We construct the monoid from $\mathcal{T}(\mathcal{V}, \mathcal{F})$ by minor modification of definition. Let all the function names be unary, i.e $\mathcal{F}n = \mathcal{F}n_1$, while the constructor set be $\mathcal{C} = \mathcal{C}_2 \cup \mathcal{C}_1 \cup \mathcal{C}_0$. Let us denote terms constructed with a $f \in \mathcal{F}n_1$ as <f t>, where t is a term[d]. Let $\mathcal{C}_2$ contains the only *associative* element named as concatenation,

[c]We encode this semantic concept in syntax as follows. Given a composition

$$t = f(\ldots, g(\ldots), \ldots)$$

where $f, g \in \mathcal{F}n$, we transform the term to `Let`(x, `eq`, $g(\ldots)$, `in`, $f(\ldots, x, \ldots)$), `Let` $\in \mathcal{F}n$, `eq`, `in` $\in \mathcal{C}_0$ are auxiliary names. The term $g(\ldots)$ is transformed recursively in the same fashion. We note the semantics of both the t and the transformed term is the same. We stress that without such representation of the composition the other strategies do not lead to successful experiments with the cache coherence protocols.

[d]Here we follow Refal syntax for function calls.

used in infix notation and denoted with the blank. The associativity allows to drop the parentheses of the constructor. Let $\mathcal{C}_1$ contains the only constructor, which denoted only with its parentheses (that is without a name). $\mathcal{C}_0 = \mathcal{K} \cup \{\lambda\}$. We denote the constants from $\mathcal{K}$ with its names: that is without the parentheses. The constant λ is denoted with nothing: it is the unit of the concatenation. Let the variable set $\mathcal{V}$ be disjoined in two sets $\mathcal{V} = \mathcal{E} \cup \mathcal{S}$, where the names from $\mathcal{E}$ are prefixed with 'e.', while the names from $\mathcal{S}$ – with 's.'. For a term t we denote the set all e-variables (s-variables) in t by $\mathcal{E}(t)$ (correspondingly $\mathcal{S}(t)$). $\mathcal{V}(t) = \mathcal{E}(t) \cup \mathcal{S}(t)$. The monoid of the terms may be defined with the following grammar:

t ::= λ | c | v | <f t> | t_1 t_2 | (t)

λ ::=

where $c \in \mathcal{K}$, $v \in \mathcal{V}$, $f \in \mathcal{F}n_1$. Thus a term is a finite sequence (including the empty sequence). According to the definition, in the examples of the terms given below elements of the sequences are separated by the blanks.

A B (C D) () F

((())) () (())

e.x (*e.y* *s.z*) *e.x* A <*f* *s.z* A>

We denote the constructed free monoid as $\mathcal{A}(\mathcal{V}, \mathcal{F})$. Any substitution has to map every $v \in \mathcal{S}$ into $\mathcal{K} \cup \mathcal{S}$.

5.1. *Restrictions on Term-rewriting Systems*

Given a term-rewriting system $\langle t, R\rangle$ on the set $\mathcal{A}(\mathcal{V}, \mathcal{F})$. Associativity of the concatenation simplifies the syntax structure of the terms, but it creates a problem with the one-step transition relation $\Rightarrow_R \subseteq \mathcal{A}(\mathcal{V}, \mathcal{F}) \times \mathcal{A}(\mathcal{V}, \mathcal{F})$, namely, given a term τ and a rule $(l \to r) \in R$, then there can be several substitutions matching τ with the l. Thus we have a new kind of non-determinism here. An example is as follows:

Example 2 *τ = <f A> and l = <f $e.x$ $e.y$>, where $A \in \mathcal{K}$, $e.x, e.y \in \mathcal{E} \subset \mathcal{V}$. There exist two substitutions matching the terms: the first is $\theta_1(e.x) = \lambda, \theta_1(e.y) = A$, the second is $\theta_2(e.x) = A, \theta_1(e.y) = \lambda$.*

Multiplicity of $v \in \mathcal{V}$ in a term t is the number of occurrences of v in t. Let us denote it as $\nu_x(t)$.

A variable $x \in \mathcal{E}$ is closed in a term t iff (1) $t \in \mathcal{C}_0$; (2) $t = (t_1)$ and x is closed in t_1; (3) $t = t_1 \ldots t_n$, where there exists at most one $t_i = x$ and x is closed in t_j for all $j = 1, \ldots, n$.

We impose the following restriction on the left hand sides of the rules from R. The multiplicity $\nu_v(l)$ of any $v \in \mathcal{E}(l)$ equals 1 and v is closed in l. These restrictions exclude recursive equations that have to be solved when we are looking for the substitutions matching a given parameterized term with a left hand side of a rule. The following terms τ = <f (A $e.p$) ($e.p$ A)> and l = <f ($e.x$) ($e.x$)> is an example showing that the recursive equation $A\ e.p = e.p\ A$ arises on $e.p$; the reason of the recursion is the fact that the multiplicity $\nu_{e.x}(l)$ is 2.[e] The second

[e]The solutions of the equation are exactly $e.p_0 \in A^*$.

example $\tau = \langle f\ e.p \rangle$ and $l = \langle f\ e.x\ A\ e.y \rangle$ as well as the example 2 demonstrate the problems, which are caused by unclosed variables (here both $e.x$ and $e.y$) in the left hand sides of the rules.

Consider the example 2. Let us think of the τ as a left hand side of a rule ρ, while of the l as a term to be matched with τ with the goal to unfold. The both substitutions given in the example 2 match the term l with τ. Hence, during the application of UNFOLD rule we have to take into account both substitutions and generate two children of l from ρ. We handle such a situation with the following additional sub-rule:

SPLIT Given a term t to be unfolded (with a rule $\rho = (l \to r)$) such that $\mathcal{E}(t)$ includes unclosed variables. Take a subterm of t of the form $\xi = t_1\ e.x\ t_2\ e.y\ t_3$ (i.e. the both variables $e.x, e.y \in \mathcal{E}(t)$ are not enclosed with the parenthesis)[f] such that there exist at least two substitutions which match ξ with the corresponding subterm of l. Generate the following three substitutions $\theta_1(e.x) = s.n\ e.x_1, \theta_2(e.x) = (e.z)\ e.x_2, \theta_3(e.x) = \lambda$. Here $e.x_1, e.x_2, e.z$ are fresh variables from $\mathcal{E}$, $s.n$ is a fresh variable from $\mathcal{S}$. Unfold $t\theta_i$ with the rule ρ.

The SPLIT rule is recursive and terminates. See Appendix [27] for the examples using this rule.

The example 2 shows also another problem. The term l may be considered as a generalization of the term τ: $\tau \ll l$. The problems is: there exists a term $\xi = e.z$ such that $l \neq \xi, \tau \ll \xi$ and both $l \ll \xi$ and $\xi \ll l$ hold. Now we specify generalization. Given two terms $t_1, t_2 \in \mathcal{A}(\mathcal{V}, \mathcal{F})$ and the set G of all the most specific terms generalizing both t_1 and t_2 (see section 4). We use (as the result of generalization of t_1, t_2) $g \in G$ such that $\sum_{x \in \mathcal{E}(g)} \nu_x(g)$ is minimal over G. As a consequence, given a rule ρ the chosen g is the generalized term which generates (under unfolding by the ρ) the minimal number of its children among all possible generalizations of t_1 and t_2.

5.2. *The well-quasi-ordering on* $\mathcal{A}(\mathcal{V}, \mathcal{F})$

Given $t_1, t_2 \in \mathcal{A}(\mathcal{V}, \mathcal{F})$, there exist two elementary functions constructing a new term from the given term. The functions are $F_1(t_1, t_2) = t_1\ t_2$ and $F_2(t_1) = (t_1)$. There exists also a family of functions $F_f(t_1) = \langle f\ t_1 \rangle$, where $f \in \mathcal{F}n_1$. We consider a quasi-ordering such that: (1) with respect to it all these functions are monotone non-decreasing $t_1 \trianglelefteq F_1(t_1, t_2), t_1 \trianglelefteq F_2(t_1), t_1 \trianglelefteq F_f(t_1)$; (2) these functions are matched with the quasi-ordering: $t_1 \trianglelefteq t_2$ implies $F_2(t_1) \trianglelefteq F_2(t_2), F_f(t_1) \trianglelefteq F_f(t_2)$ and for any term t both $F_1(t, t_1) \trianglelefteq F_1(t, t_2)$ and $F_1(t_1, t) \trianglelefteq F_1(t_2, t)$ hold. The following relation is a variant of the Higman-Kruskal relation and is a well-quasi-ordering [14, 17].

Definition 6 *The homeomorphic embedding relation* $\trianglelefteq$ *is the smallest transitive relation on* $\mathcal{A}(\mathcal{V}, \mathcal{F})$ *satisfying the following properties, where* $h \in \mathcal{F}n_1$, $s \in \mathcal{A}(\mathcal{V}, \mathcal{F})$, $t, t_i \in \mathcal{A}(\mathcal{V}, \mathcal{F})$.

[f] Recall that a term t_i is a finite sequence: $t_i = \tau_{i_1} \ \ldots \ \tau_{i_k}$.

1. $\forall x, y \in \mathcal{E}.\ x \underline{\propto} y, \forall u, v \in \mathcal{S}.\ u \underline{\propto} v$;

2. $t \underline{\propto} \texttt{<}h\ t\texttt{>}, t \underline{\propto} (t), t \underline{\propto} s\ t, t \underline{\propto} t\ s$;

3. $s \underline{\propto} t$, *then* $\texttt{<}h\ s\texttt{>} \underline{\propto} \texttt{<}h\ t\texttt{>}$, $(s) \underline{\propto} (t)$, $s\ t_1 \underline{\propto} t\ t_1, t_1\ s \underline{\propto} t_1\ t$.

Corollary 1 *For any terms* $t, t_1, \ldots, t_n \in \mathcal{A}(\mathcal{V}, \mathcal{F})$

1. $\lambda \underline{\propto} t \underline{\propto} t$, *where* λ *is the empty sequence;*

2. $\exists i_1, \ldots, i_j$ *such that* $1 \leq i_1 < i_2 < \ldots < i_j \leq n$, *then* $t_{i_1} \ldots t_{i_j} \underline{\propto} t_1 \ldots t_n$.

Given an infinite sequence of terms $t_1, \ldots, t_n, \ldots$, this relation is relevant to approximation of loops increasing the syntactical structures in the sequence; or in other words to looking for the regular similar cases of mathematical induction on the structure of the terms. That is to say the cases, which allow refer one to another by a step of the induction. An additional restriction separates the basic cases of the induction from the regular ones. The restriction is:

$$\forall c \in \mathcal{K}.() \not\underline{\propto} (c) \wedge \forall v \in \mathcal{S}.() \not\underline{\propto} (v).$$

We impose this restriction on the relation $\underline{\propto}$ and denote the obtained relation as $\preceq$. It is easy to see that such a restriction does not violate the quasi-ordering property. Note that the restriction may be varied in the obvious way, but for our experiments its simplest case given above is used to control applications of GENERALIZE rule and has turned out to be sufficient.

6. Discussion

In addition to the MOESI protocol described in Appendix [27] the supercompiler SCP4 verified by our scheme the following parameterized cache coherence protocols: IEEE Futerbus+, MESI, MSI, "The University of Illions", DEC Firefly, "Berkeley", Xerox PARC Dragon [5, 6]. All these protocols are specified analogously to the description given in Appendix [27]. In the case of the MOESI protocol the time of automatic verification is 1 second (Windows XP/Service Pack 2, Intel Pentium III, 450 MHz, 256 MB of RAM); verification of the other protocols takes times, which slightly differ from the indicated.

One of the questions left open is why do we work in terms of the free monoid $\mathcal{A}(\mathcal{V}, \mathcal{F})$ and how important is such a choice? Actually, the supercompiler SCP4 is able to prove correctness of the main example considered in Appendix [27] encoded in terms of a free algebra terms too, but the proof is much more bulky as compared with the proof presented in section [*The Inductive Proof*] of [27]. Moreover, the proof in this case requires additional capabilities of the supercompiler which are not presented in our formal model. We leave detailed analysis and comparisons of different encodings to future work.

The work reported in this paper has started as mainly driven by experiments. There is still much work to be done, both theoretically and experimentally. On the theory side we would like to have the completeness results for classes of verification

problems and particular strategies. The applicability of the strategies already implemented in SCP4 is also worth to explore further. Recent experiments have shown that SCP4 strategies are quite robust with respect to order in which rewriting rules are encoded in Refal programs. For example, the above MOESI protocol can be verified with any of 120 (=5!) permutations of rewriting rules for `RandomAction`. See [26] for details to this subject.

Finally, comparisons with related work, especially with [12, 21, 22] and [36] should be done. In both these approaches transformations of logic (as opposed to our functional) programs are used to perform verification of parameterized systems. Despite the differences in programming languages, systems encodings and verifications schemes used, all three approaches have a common ground and rely on variants of unfold/fold transformations. Very recently in [13] the verification methodology based on *distillation* program transformation technique has been proposed. It is claimed that distillation is a generalization of supercompilation, but precise relationships between verification methods based on both techniques remain to be studied further.

Acknowledgements

The authors thank Andrei V. Klimov who read our paper and made meaningful remarks.

Many thanks to anonymous referees for several insightful comments that led to a substantial improvement of the paper.

The second author is supported by the Program for Basic Research of the Presidium of Russian Academy of Sciences (as a part of "Development of the basis of scientific distributed informational-computing environment on the base of GRID technologies"), and the Russian Ministry of Sciences and Education (grant 2007-4–1.4-18-02-064).

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson and M. Saksena, "A Survey of Regular Model Checking", *Proc. 15th Int. Conf. on Concurrency Theory*, LNCS **3170** (2004) 35–48.
2. K. Baukus, K. Stahl, S. Bensalem and Y. Lakhnech, "Networks of Processes with Parameterized State Space", In *Electronic Notes in Theoretical Computer Science*, Jan. 2004, vol. **50**, no.4, pp. 1–15.
3. N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. B. Sipma and T. E. Uribe, "STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems", *Proc. International Conference on Computer Aided Verification, CAV'96*, LNCS **1102** (Springer-Verlag, 1996) pp. 415-418.
4. E. M. Clarke, Grumberg and D. Peled, *Model Checking.* (MIT Press, 1999).
5. G. Delzanno, "Automatic Verification of Parameterized Cache Coherence Protocols", *Proc. of the 12th Int. Conf. on Computer Aided Verification*, LNCS **1855** (2000) pp. 53–68.
6. G. Delzanno, "Automatic Verification of Cache Coherence Protocols via Infinite-state Constraint-based Model Checking", http://www.disi.unige.it/person/DelzannoG/protocol.html.

7. G. Delzanno, "Verification of Consistency Protocols via Infinite-state Symbolic Model Checking, A Case Study", *Proc. of FORTE/PSTV*, 2000, pp: 171-186.
8. G. Delzanno, "Contsraint-based Verification of Paremeterized Cache Coherence Protocols", *Formal Methods in System Design* **23(3)** (2003) pp. 257–301.
9. A. P. Ershov, "Mixed computation in the class of recursive program schema", *Acta Cybernetica* **4(1)** (1978) pp. 19–23.
10. J. Esparza, "Decidability of model checking of infinite state concurrent systems", *Acta Informatica* **34** (1997) pp. 85–107.
11. J. Esparza, A. Finkel, R. Mayr, On the Verification of Broadcast Protocols, In Proc. of LICS 1999, pp 352-359.
12. R. Glück and M. Leuschel, "Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification", *Proc. of Systems Informatics*, LNCS **1755** (Springer-Verlag, Novosibirsk, Russia, 1999) pp. 93–100.
13. G. W. Hamilton, "Distilling Programs for Verification", *Proc. of the International Conference on Compiler Optimization Meets Compiler Verification*, Braga, Portugal, March 2007. To Appear in Electronic Notes in Theoretical Computer Science.
14. G. Higman, "Ordering by divisibility in abstract algebras", *Proc. London Math. Soc.* **2(7)** (1952) 326–336.
15. N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation,* (Prentice Hall International, 1993).
16. A. V. Korlyukov and A. P. Nemytykh, "Supercompilation of Double Interpretation. (How One Hour of the Machine's Time Can Be Turned to One Second)", 2002. http://www.refal.net/~korlukov/scp2int/Karliukou_Nemytykh.pdf.
17. J. B. Kruskal, "Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture", *Trans. Amer. Math. Society* **95** (1960) 210–225.
18. M. Leuschel, B. Martens, "Global Control for Partial Deduction through Characteristic Atoms and Global Trees", *Proc. of the PEPM'96* LNCS **1110** (1996) 263–283.
19. M. Leuschel, "On the Power of Homeomorphic Embedding for Online Termination", *Proc. of the SAS'98*, LNCS **1503** (1998) pp. 230–245.
20. M. Leuschel and H. Lehmann, "Program Specialization, Inductive Theorem Proving and Infinite State Model Checking", *Proc. of the LOPSTR'03*, Uppsala, 2003, avaialable at: www.ecs.soton.ac.uk/~mal/presentations/ITP_Lopstr03.ppt.
21. M. Leuschel and H. Lehmann, "Solving coverability problems of Petri nets by partial deduction", *Proc. 2nd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'2000)* (Montreal, Canada, 2000) pp: 268–279.
22. M. Leuschel and T. Massart, "Infinite state model checking by abstract interpretation and program specialisation", *Logic-Based Program Synthesis and Transformation. In Proc. of LOPSTR'99*, LNCS **1817** (Venice, Italy, 2000) pp: 63–82.
23. A. P. Lisitsa and A. P. Nemytykh, "Towards Verification via Supercompilation", *Proc. of COMPSAC 05, the 29th Annual International Computer Software and Applications Conf., Workshop Papers and Fast Abstracts* (IEEE, 2005) pp. 9–10.
24. A. P. Lisitsa and A. P. Nemytykh, *Verification via Supercompilation.* `http://www.csc.liv.ac.uk/~alexei/VeriSuper/`
25. A. P. Lisitsa and A. P. Nemytykh, "Verification as a Parameterized Testing (Experiments with the SCP4 Supercompiler)", *Programmirovanie* **1** (2007) pp. 22–34 (In Russian). English translation in *J. Programming and Computer Software*, Vol. **33**, No.1 (Pleiades Publishing Ltd, 2007) pp. 14–23.

26. A. P. Lisitsa and A. P. Nemytykh, "A Note on Specialization of Interpreters", *Proc. of The 2-nd International Symposium on Computer Science in Russia (CSR-2007)*, LNCS **4649** (Springer, 2007) pp. 237–248.
27. A. P. Lisitsa and A. P. Nemytykh, Appendix to "Reachability Analysis in Verification via Supercompilation", 2007. Avaialable at: http://www.botik.ru/pub/local/scp/refal5/rp07_appendix.pdf
28. A. P. Lisitsa and A. P. Nemytykh, *Experiments on verification via supercompilation.* http://refal.botik.ru/protocols/ , 2007.
29. A. P. Nemytykh, "A Note on Elimination of Simplest Recursions", *Proc. of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, (ACM Press, 2002) pp. 138–146.
30. A. P. Nemytykh, "The Supercompiler SCP4: General Structure (extended abstract)", *Proc. of the Perspectives of System Informatics*, LNCS **2890** (Springer-Verlag, 2003) pp. 162–170.
31. A. P. Nemytykh, "Playing on REFAL", *Proc. of the International Workshop on Program Understanding.* A.P. Ershov Institute of Informatics Systems, Syberian Branch of Russian Academy of Sciences, pp: 29–39, July 2003. (ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_PU03.ps.gz)
32. A. P. Nemytykh, "The Supercompiler SCP4: General Structure", (in English), *Programmnye sistemy: teoriya i prilozheniya*, Vol. **1** (Fizmatlit, Moscow, 2004) pp. 448-485. ftp://ftp.botik.ru/pub/local/scp/refal5/GenStruct.ps.gz.
33. A. P. Nemytykh, *The Supercompiler SCP4: General Structure* (URSS, Moscow, 2007). (A book in Russian)
34. A. P. Nemytykh and V. F. Turchin, *The Supercompiler SCP4: sources, on-line demonstration*, http://www.botik.ru/pub/local/scp/refal5/, (2000).
35. S. A. Romanenko, "Arity raiser and its use in program specialization", *Proc. of the ESOP'90*, LNCS **432** (1990) pp. 341–360.
36. A. Roychoudhury and C. R. Ramakrishnan, "Unfold/fold Transformations for Automated Verification of Parameterized Concurrent Systems", In *Program Development in Computational Logic*, LNCS **3049** (2004) pp. 262–291.
37. D. Sands, "Proving the correctness of recursion-based automatic program transformation", In *Theory and Practice of Software Development*, LNCS **915** (1995) pp. 681–695.
38. M. H. Sørensen and R. Glück, "An algorithm of generalization in positive supercompilation", *Logic Programming: Proceedings of the 1995 International Symposium* (MIT Press, 1995) pp. 486–479.
39. M. H. Sørensen and R. Glück, "Introduction to Supercompilation", *Partial Evaluation - Practice and Theory*, DIKU 1998 International Summer School. June 1998. http://repository.readscheme.org/ftp/papers/pe98-school/D-364.pdf
40. M. H. Sørensen, R. Glück and N. D. Jones, "A positive supercompiler", *Journal of Functional Programming* **6(6)** (1996) 811–838.
41. V. F. Turchin, "The concept of a supercompiler", *ACM Transactions on Programming Languages and Systems* **8** (1986) pp. 292–325.
42. V. F. Turchin, "The algorithm of generalization in the supercompiler", *Proc. of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation* (North-Holland Publishing Co., Amsterdam, 1988) pp. 531–549.
43. V. F. Turchin, *Refal-5, Programming Guide and Reference Manual* (New England Publishing Co., Holyoke, Massachusetts, 1989) (electronic version: http://www.botik.ru/pub/local/scp/refal5/, 2000)

44. V. F. Turchin, "Metacomputation: Metasystem transition plus supercompilation", *Proc. of the PEPM'96*, LNCS **1110** (Springer-Verlag, 1996) pp. 481–509.
45. V. F. Turchin, D. V. Turchin, A. P. Konyshev and A. P. Nemytykh, *Refal-5: sources, executable modules* http://www.botik.ru/pub/local/scp/refal5/, (2000)
46. P. Wadler, "Deforestation: Transforming programs to eliminate tree", *Theoretical Computer Science*, **73** (1990) 231–238.