Why Multi-Result Supercompilation Matters: Case Study of Reachability Problems for Transition Systems

Andrei V. Klimov*

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
klimov@keldysh.ru

Abstract. We sum up some current results of the theoretical study of the reasons of successful application of supercompilers to verification of monotonic counter systems representing the models of practical protocols and other parallel systems. Three supercompilation-like algorithms for forward analysis of counter systems and a procedure interactively invoking a supercompiler to solve the coverability problem are presented. It always terminates for monotonic counter systems. The algorithms are considered as an instance of multi-result supercompilation (proposed by I. Klyuchnikov and S. Romanenko) for special-purpose analysis of counter systems.

We speculate on the insufficiency of single-result supercompilation for solving the reachability and coverability problems and the necessity of multi-result supercompilation. Related work by Gilles Geeraerts et al. on the algorithmic schema referred to as 'Expand, Enlarge and Check' (EEC) is discussed. We regard the work on EEC as a theory of domain-specific multi-result supercompilation for well-structured transition systems. The main purpose of the theory is to prune combinatorial search for suitable residual graphs as much as possible. Based of the EEC theory and our results, several levels of the restrictions of the combinatorial search dependent on the properties of a subject transition system are revealed: some minimal restrictions when solving reachability for arbitrary transition systems; more efficient search (according to the EEC schema) in the case of well-structured transitions systems; iterative calls of a single-result supercompiler with a varying parameter of generalization for monotonic counter systems. On the other hand, no reasonable practical class of transition systems, for which the reachability or coverability is always solvable by a version of single-result supercompilation is known yet.

Keywords: multi-result supercompilation, verification, reachability, coverability, well-structured transition systems, counter systems.

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

1 Introduction

Supercompilation [27] is a forward analysis and transformation method. *Forward* means that the traces of program execution are elaborated starting from a set of initial states to a certain depth. Respectively, *backward* analysis is based on elaboration of traces backwards from a set of final states. Many other forward and backward analysis techniques have been developed for specific classes of algorithm systems, and their comparison with supercompilation and cross-fertilization is a productive research topic.[1]

Traditionally supercompilation has been developed for programs, while in this paper we deal with *transition systems* and solving the reachability problems (reachability and coverability) for them.

Programs (algorithms) and transition systems differ in some respects:

- (Essential) The result of evaluation of a program is its *final state*. (In more detail: the sense of a program is a mapping from a set of initial states to a set of final states). The outcome of a transition system comprises *all* states reachable from a set of initial states and even more up to the set of all traces. Different outcomes are considered depending on a problem under consideration. For the reachability problems the set of all reachable states is used.
- (Inessential) Programs are usually deterministic, while transition systems nondeterministic. Nondeterministic systems are somewhat technically simpler for supercompilation. (There is no need of propagating negative conditions to else branches.)

The tasks of verification of programs or transition systems allows for easy and natural comparison of methods: if some method proves more properties than another one, the former is more "powerful" than the latter. I. Klyuchnikov and S. Romanenko came to the idea of multi-result supercompilation (MRSC) [15] while dealing with proving program equivalence by supercompilation: two programs are equivalent when the respective residual terms coincide. A multi-result supercompiler, which returns a set of residual programs instead of just one in case of single-results supercompilation, allows for proving the equivalence of more programs: two programs are equivalent when the respective sets of residual programs intersect.

Solving the reachability problem (whether a given set of states is reachable from a given set of initial states by a program or a transition system) turned out to be even more sensible to the use of multi-result supercompilation instead of single-result one.

The goal of this paper is to demonstrate and argue that, and under what conditions, multi-result supercompilation is capable of solving the reachability and coverability problems for transition systems, while single-result supercompilers solve the problem only occasionally.

[1] "Backward supercompilation" is also a promising topic, as well as "multi-directed" analysis from arbitrary intermediate program points in both directions. V. Turchin initiated this research in [25,28], but much work is still required.

History and Related Work. This paper continues research [12,13,14] into why and how *supercompilers* are capable of solving the *reachability* and *coverability problems* for *counter systems* where the set of target states is upward-closed and the set of initial states has a certain form. The idea of verification by supercompilation stems from the pioneering work by V. Turchin [26]. The fact that a supercompiler can solve these problems for a lot of practically interesting counter systems has been experimentally discovered by A. Nemytykh and A. Lisitsa [19,21] with a Refal supercompiler SCP4 [24] and then the result has been reproduced with the Java supercompiler JScp [10,11] and other supercompilers.

In paper [14] two classic versions of the supercompilation algorithm are formulated for counter systems, using the notation close to the works on Petri nets and transition systems. Another paper [13] contains a simplification of one of them (namely, with the so-called lower-node generalization) by clearing out the details that have been found formally unnecessary for solving the coverability problem. The algorithms are reproduced here.

It has been found that the coverability problem for monotonic counter systems is solvable by iterative application of a supercompiler varying an integer parameter $l = 0, 1, 2, \ldots$, that controls termination of driving and generalization of integers: small non-negative integers less than l are not allowed to be generalized to a variable.[2] Papers [13,14] contain a proof that there exists such value l that the supercompiler solves the problem. (Remarkably, the proof is based on the existence of a non-computable upper estimate of l.)

In [14] (and here in Algorithms 2 and 3) this rule is used together with the traditional whistle[3] based on a well-quasi-order: the whistle is not allowed to blow if two configurations under test differ only in integers less than l. In [13] (and here in Algorithm 1) this rule is used in pure form as an imperative: every integer greater or equal to l is immediately generalized.

The iterative procedure of application a supercompiler demonstrates usefulness of *gradual specialization*: starting with a trivial result of supercompilation gradually produce more and more specialized versions until one is obtained that satisfies the needs. This differs from the traditional design of supercompilers, which try to produce the most specific residual code at once.

These algorithms may be considered as an instance of *multi-result supercompilation* [15]: to solve a problem one varies some of the degrees of freedom inherent in supercompilation, obtains several residual programs and chooses a better one w.r.t. his specific goals. They have shown the usefulness of multi-result supercompilation for proving the equivalence of expressions and in two-level supercompilation.

In our work, a problem unsolvable by single-result supercompilation is solvable by enumerating a potentially infinite set of supercompilation results parameterized by an integer.

[2] In terms of Petri net theory this phrase sounds as follows: integer values less than l are not allowed to be *accelerated*, i.e., be replaced with a special value ω.

[3] In supercompilation jargon, a *whistle* is a termination rule.

The decidability of the coverability problem for well-structured transition systems, which counter systems belong to, is well-known [1]. The iterative procedure of application of the supercompilers presented below is close to the algorithmic schema referred to as 'Expand, Enlarge and Check' (EEC) [5,6] for solving the coverability problem of *well-structured transition systems* (WSTS). We discuss their relation is Section 4.

Outline. The paper is organized as follows.

In Section 2 we recall some known notions from the theory of transition systems as well as give specific notions from supercompilation theory used in the algorithms presented in Section 3.3. The algorithms are reproduced from [13,14] to illustrate the idea of multi-result supercompilation applied to solving the coverability problem. (If you are familiar with these papers, you may go immediately to Section 4 and return back when needed.)

Section 4 contains a new contribution and answers the question in the title: it takes a more general look at the methods of solving the reachability problems for transition systems; discusses the related work by G. Geeraerts et al., which we regard as a theory of multi-result supercompilation for well-structured transition systems (WSTS); and reveal an hierarchy of properties of transition systems, which allows for pruning extra enumeration of residual graphs and making multi-result supercompilation more efficient. Monotonic counter systems turned out to be the most efficient case in this hierarchy.

In Section 5 we discuss related work and in Section 6 conclude.

2 Basic Notions

2.1 Transition Systems

We use the common notions of *transition system*, *monotonic transition system*, *well-structured transition system*, *counter system* and related ones.

A *transition system* $\mathcal{S}$ is a tuple $\langle S, \Rightarrow\rangle$ such that S is a possibly infinite set of states, $\Rightarrow \subseteq S \times S$ a transition relation.

A *transition function* $\mathsf{Post}(\mathcal{S}, s)$ is used to denote the set $\{s' \mid s \Rightarrow s'\}$ of one-step successors of s.[4]

$\mathsf{Post}^*(\mathcal{S}, s)$ denotes the set $\{s' \mid s \stackrel{*}{\Rightarrow} s'\}$ of successors of s.

$\mathsf{Reach}(\mathcal{S}, I)$ denotes the set $\bigcup_{s\in I} \mathsf{Post}^*(\mathcal{S}, s)$ of states *reachable* from a set of states I.

We say a transition system $\langle S, \Rightarrow\rangle$ is (*quasi-, partially*) *ordered* if some (quasi-,[5] partial[6]) order $\preccurlyeq$ is defined on its set of states S.

For a quasi-ordered set X, $\downarrow X$ denotes $\{x \mid \exists y \in X\colon x \preccurlyeq y\}$, the downward closure of X. $\uparrow X$ denotes $\{x \mid \exists y \in X\colon x \succcurlyeq y\}$, the upward closure of X.

[4] For effectiveness, we assume the set of one-step successors is finite.

[5] A *quasi-order* (*preorder*) is a reflexive and transitive relation.

[6] A *partial order* is an antisymmetric quasi-order.

The *covering set* of a quasi-ordered transition system $\mathcal{S}$ w.r.t. an initial set I, noted $\mathsf{Cover}(\mathcal{S}, I)$, is the set $\downarrow\mathsf{Reach}(\mathcal{S}, I)$, the downward closure of the set of states reachable from I.

The *coverability problem* for a quasi-ordered transition system $\mathcal{S}$, an initial set of states I and an upward-closed target set of states U asks a question whether U is reachable from I: $\exists s \in I, s' \in U : s \stackrel{*}{\Rightarrow} s'$.[7]

A quasi-order $\preccurlyeq$ is a *well-quasi-order* iff for every infinite sequence $\{x_i\}$ there are two positions $i < j$ such that $x_i \preccurlyeq x_j$.

A transition system $\langle S, \Rightarrow\rangle$ equipped with a quasi-order $\preccurlyeq \subseteq S \times S$ is said to be *monotonic* if for every $s_1, s_2, s_3 \in S$ such that $s_1 \Rightarrow s_2$ and $s_1 \preccurlyeq s_3$ there exists $s_4 \in S$ such that $s_3 \stackrel{*}{\Rightarrow} s_4$ and $s_2 \preccurlyeq s_4$.

A transition system is called *well-structured (WSTS)* if it is equipped with a well-quasi-order $\preccurlyeq \subseteq S \times S$ and is monotonic w.r.t. this order.

A *k-dimensional counter system* $\mathcal{S}$ is a transition system $\langle S, \Rightarrow\rangle$ with states $S = \mathbb{N}^k$, k-tuples of non-negative integers. It is equipped with the component-wise partial order $\preccurlyeq$ on k-tuples of integers:

$$\begin{aligned} s_1 \preccurlyeq s_2 \quad &\text{iff} \quad \forall i \in [1, k] : \; s_1(i) \leq s_2(i), \\ s_1 \prec s_2 \quad &\text{iff} \quad s_1 \preccurlyeq s_2 \;\wedge\; s_1 \neq s_2. \end{aligned}$$

Proposition 1. *The component-wise order $\preccurlyeq$ of k-tuples of non-negative integers is a well-quasi order. A counter system equipped with this order is a well-structured transitions system.*

2.2 Configurations

In supercompilation the term *configuration* denotes a representation of a set of states, while in Petri net and transition system theories the same term stands for a ground state. In this paper the supercompilation terminology is used. Our term *configuration* is equivalent to *ω-configuration* and *ω-marking* in Petri net theory.

The general rule of construction of the notion of a configuration in a supercompiler from that of the program state in an interpreter is as follows: add configuration variables to the data domain and allow these to occur anywhere where a ground value can occur. A configuration represents the set of states that can be obtained by replacing configuration variables with all possible values. Thus the notion of a configuration implies a set represented by some constructive means rather than an arbitrary set.

A state of a counter system is a k-tuple of integers. According to the above rule, a configuration should be a tuple of integers and configuration variables. For the purpose of this paper we use a single symbol ω for all occurrences of variables and consider each occurrence of ω a distinct configuration variable.

[7] In other words, the coverability problem asks a question whether such a state r is reachable from I that $\downarrow\{r\} \cap U \neq \emptyset$, where there is no requirement that the target set U is upward-closed.

Thus, in supercompilation of k-dimensional counter systems *configurations* are k-tuples over $\mathbb{N}\cup\{\omega\}$, and we have the set of all configurations $\mathcal{C} = (\mathbb{N}\cup\{\omega\})^k$.

A configuration $c \in \mathcal{C}$ represents a set of states noted $[\![c]\!]$:

$$[\![c]\!] = \{\langle x_1, \ldots, x_k\rangle \mid x_i \in \mathbb{N} \text{ if } c(i) = \omega,\ x_i = c(i) \text{ otherwise},\ 1 \le i \le k\}.$$

These notations agree with that used in Petri net and counter system theories. Notice that by using one symbol ω we cannot capture information about equal unknown values represented by repeated occurrences of a variable. However, when supercompiling counter systems, repeated variables do not occur in practice, and such simplified representation satisfies our needs.

We also use an extension of $[\![\cdot]\!]$ to sets of configurations to denote all states represented by the configurations from a set C: $[\![C]\!] = \bigcup_{c\in C}[\![c]\!]$.

Definition 1 (Coverability set). *A* coverability set *is a finite set of configurations C that represents the covering set in the following way: $\downarrow[\![C]\!] = \mathsf{Cover}(\mathcal{S}, I)$.*

Notice that if we could find a coverability set, we could solve the coverability problem by checking its intersection with the target set U.

2.3 Residual Graph, Tree and Set

Definition 2 (Residual graph and residual set). *Given a transition system $\mathcal{S} = \langle S, \Rightarrow\rangle$ along with an initial set $I \subseteq S$ and a set $\mathcal{C}$ of configurations,* a residual graph *is a tuple $\mathcal{T} = \langle N, B, n_0, C\rangle$, where N is a set of nodes, $B \subseteq N\times N$ a set of edges, $n_0 \in N$ a root node, $C\colon N \to \mathcal{C}$ a labeling function of the nodes by configurations, and*

1. *$[\![I]\!] \subseteq [\![C(n_0)]\!]$, and for every state $s \in S$ reachable from I there exists a node $n \in N$ such that $s \in [\![C(n)]\!]$, and*
2. *for every node $n \in N$ and states s, s' such that $s \in [\![C(n)]\!]$ and $s \Rightarrow s'$ there exists an edge $\langle n, n'\rangle \in B$ such that $s' \in [\![C(n')]\!]$.*

We call the set $\{C(n) \mid n \in N\}$ of all configurations in the graph a residual set.

Notice that a residual set is a representation of an over-approximation of the set of reachable states: $\downarrow[\![\{C(n) \mid n \in N\}]\!] \supseteq \mathsf{Reach}(\mathcal{S}, I)$.

The term *residual* is borrowed from the metacomputation terminology, where the output of a supercompiler is referred to as a *residual graph* and a *residual program*. The literature on transition systems lacks a term for what we call *residual set*. They use only the term *coverability set*, which means a specific case of a residual set, where it is a precise representation of the covering set $\mathsf{Cover}(\mathcal{S}, I) = \downarrow\mathsf{Reach}(\mathcal{S}, I)$.

The value of these notions for our purpose is as follows. To solve the coverability problem it is sufficient to find a coverability set among the residual sets: then we check whether all configurations in the coverability set are disjoint with the target set or not. Unfortunately, computing a coverability set is undecidable

for counter systems of our interest. Fortunately, this is not necessary. It is sufficient to build a sequence of residual sets that contains a coverability set. We may not know which one among the residual sets is a coverability set (this is incomputable), it is sufficient to know it exists in the sequence. This is the main idea of our algorithm and the 'Expand, Enlarge and Check' (EEC) algorithmic schema of [6].

Notice that such procedure of solving the coverability problem does not use the edges of the residual graph, and we can keep in B only those edges that are needed for the work of our versions of supercompilation algorithms. Hence the definition of a residual tree:

Definition 3 (Residual tree). *A* residual tree *is a spanning tree of a residual graph. The root of the tree is the root node n_0 of the graph.*

2.4 Operations on Configurations

To define a supercompiler we need the transition function Post on states to be extended to the corresponding function Drive on configurations. It is referred to as (one-step) *driving* in supercompilation and must meet the following properties (where $s \in S$, $c \in \mathcal{C}$):

1. $\mathsf{Drive}(\mathcal{S}, s) = \mathsf{Post}(\mathcal{S}, s)$ — a configuration with ground values represents a singleton and its successor configurations are respective singletons;
2. $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \supseteq \bigcup\{\mathsf{Post}(\mathcal{S}, s) \mid s \in [\![c]\!]\}$ — the configurations returned by Drive over-approximate the set of one-step successors. This is the *soundness* property of driving. The over-approximation suits well for applications to program optimization, but for verification the result of Drive must be more precise. Hence the next property:
3. $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \subseteq \downarrow\bigcup\{\mathsf{Post}(\mathcal{S}, s) \mid s \in [\![c]\!]\}$ — for solving the coverability problem it is sufficient to require that configurations returned by Drive are subsets of the downward closure of the set of the successors.

For the practical counter systems we experimented with, the transition function Post is defined in form of a finite set of partial functions taking the coordinates v_i of the current state to the coordinates v'_i of the next state:

$$v'_i = \textbf{ if } G_i(v_1, \ldots, v_k) \textbf{ then } E_i(v_1, \ldots, v_k), i \in [1, k],$$

where the 'guards' G_i are conjunctions of elementary predicates $v_j \geq a$ and $v_j = a$, and the arithmetic expressions E_i consist of operations $x + y$, $x + a$ and $x - a$, where x and y are variables or expressions, $a \in \mathbb{N}$ a constant.

The same partial functions define the transition function Drive on configurations, the operations on ground data being generalized to the extended domain $\mathbb{N} \cup \{\omega\}$: $\forall a \in \mathbb{N}$: $a < \omega$ and $\omega + a = \omega - a = \omega + \omega = \omega$.

2.5 Restricted Ordering of Configurations of Counter Systems

To control termination of supercompilers we use a restricted partial order on integers $\preccurlyeq_l$ parameterized by $l \in \mathbb{N}$. For $a, b \in \mathbb{N} \cup \{\omega\}$, we have:

$$a \preccurlyeq_l b \quad \text{iff} \quad l \leq a \leq b < \omega.$$

This partial order makes two integers incompatible when one of them is less than l. The order is a well-quasi-order.

Then the partial order on states and configurations of counter systems is the respective component-wise comparison: for $c_1, c_2 \in \mathcal{C} = (\mathbb{N} \cup \{\omega\})^k$,

$$c_1 \preccurlyeq_l c_2 \quad \text{iff} \quad \forall i \in [1, k]\colon\ c_1(i) \preccurlyeq_l c_2(i).$$

This order is also a well-quasi-order. It may be regarded as a specific case of the homeomorphic embedding of terms used in supercompilation to force termination. As we will see in the supercompilation algorithms below, when two configurations c_1 and c_2 are met on a path such that $c_1 \prec_l c_2$, 'a whistle blows' and generalization of c_1 and c_2 is performed. Increasing parameter l prohibits generalization of small non-negative integers and makes 'whistle' to 'blow' later. When $l = 0$, the order is the standard component-wise well-quasi-order on tuples of integers. When $l = 1$, value 0 does not compare with other positive integers and generalization of 0 is prohibited. And so on.

2.6 Generalization

When the set of states represented by a configuration c is a subset of the set represented by a configuration g, $[\![c]\!] \subseteq [\![g]\!]$, we say the configuration g is *more general* than the configuration c, or g is a *generalization* of c.[8] Let $\sqsubseteq$ denote a *generalization relation* $\sqsubseteq \in \mathcal{C} \times \mathcal{C}$: $c \sqsubseteq g$ iff $[\![c]\!] \subseteq [\![g]\!]$, $c \sqsubset g$ iff $[\![c]\!] \subsetneq [\![g]\!]$.

For termination of traditional supercompilers generalization must meet the requirement of the finiteness of the number of possible generalization for each configuration:

$$\forall c \in \mathcal{C}\colon \{g \in \mathcal{C} \mid c \sqsubseteq g\} \text{ is finite.}$$

In Section 4 we speculate on a possibility to lift this restriction for multi-result supercompilation.

We use a function $\mathsf{Generalize}\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ to find a configuration g that is more general than two given ones. Usually $\mathsf{Generalize}(c_1, c_2)$ returns the least general configuration, however this is not formally required for the soundness and termination of supercompilation, as well as for the results of this paper, although it is usually desirable for obtaining 'better' residual programs.

In the case of counter systems and the set of configurations defined above, function $\mathsf{Generalize}(c_1, c_2)$ sets to ω those coordinates where configurations c_1 and c_2 differ:

$$\mathsf{Generalize}(c_1, c_2) = c \text{ s.t. } \forall i \in [1, k]\colon\ c(i) = c_1(i) \text{ if } c_1(i) = c_2(i),\ \omega \text{ otherwise.}$$

[8] In Petri net and counter system theories generalization is referred to as *acceleration*.

Notice that the function Generalize has no parameter l. However in the supercompilation algorithms below it is called in cases where $c_1 \prec_l c_2$. Hence generalization is not performed when one of the integers is less than l.

In Algorithm 1 a generalization function $\mathsf{Generalize}_l(c)$ with an integer parameter l and one configuration argument c is used. It sets to ω those coordinates that are greater or equal than l:

$$\mathsf{Generalize}_l(c) = g \text{ s.t. } \forall i \in [1,k]\colon\ g(i) = c(i) \text{ if } c(i) < l,\ \omega \text{ otherwise.}$$

Because of the importance of the parameter l, we write it in a special position: as the index of the function names that take it as an argument.

3 Supercompilation-Based Algorithms for Solving Reachabiliy Problems

In this section we present an algorithm to solve the reachability problem for transition systems. It is applicable to an arbitrary transition system provided the basic functions Drive, $\mathsf{Generalize}_l$ and Generalize are properly defined, but has been proven to terminate only for well-structured transition systems when solving the coverability problem, provided the basic functions satisfy certain properties. In the previous section we gave an example of their definition for counter systems. The theory by G. Geeraerts et al. [5,6] explains how to define them for well-structured transition systems.

The algorithm consists of the main function $\mathsf{Reachable}(\mathcal{S}, I, U)$ (Algorithm 0 Approximate) that contains the top level loop, which invokes functions $\mathsf{Over}_l(\mathcal{S}, I)$ and $\mathsf{Under}_l(\mathcal{S}, I)$ to iteratively compute more and more precise over- and under-approximations of the set of reachable states, while increasing parameter l.

We give three versions of the definition of the approximation functions. The simplest one (Algorithm 1) fits the EEC schema. Another two are based on domain-specific versions for transition systems of classical supercompilation algorithms with lower-node and upper-node generalization (Algorithm 2 ScpL and Algorithm 3 ScpU respectively).

3.1 Main Function

Main function $\mathsf{Reachable}(\mathcal{S}, I, U)$ takes a transition system $\mathcal{S}$, an initial configuration I and a target set U and iteratively applies functions $\mathsf{Under}_l(\mathcal{S}, I)$ and $\mathsf{Over}_l(\mathcal{S}, I)$ with successive values of the parameter of generalization $l = 0, 1, 2, \ldots$ When it terminates, it returns an answer whether the set U is reachable or not. The algorithm is rather natural.[9] Reachability is determined when some under-approximation returned by $\mathsf{Under}_l(\mathcal{S}, I)$ intersects with U. Unreachability is determined when some over-approximation returned by $\mathsf{Over}_l(\mathcal{S}, I)$ is disjoint with U.

[9] The algorithm is close to but even simpler than analogous Algorithm 4.1 in [5, page 122] and Algorithm 5.2 in [5, page 141].

Algorithm 0: $\mathsf{Reachable}(\mathcal{S}, I, U)$: Solving the coverability problem for a monotonic counter system

Data: $\mathcal{S}$ a monotonic counter system
Data: I an initial configuration (representing a set of initial states)
Data: U an upward-closed target set
Result: Is U reachable from I?

```
Reachable(S, I, U)
    for l = 0, 1, 2, ... do
        if [[Under_l(S, I)]] ∩ U ≠ ∅ then
            return 'Reachable'
        if [[Over_l(S, I)]] ∩ U = ∅ then
            return 'Unreachable'
```

We regard this algorithm as a domain-specific multi-result supercompilation when the approximation functions are implemented by means of single-result supercompilation.

3.2 Simplest Approximation Function

Function $\mathsf{Approximate}_l(over, \mathcal{S}, I)$ is invoked by functions $\mathsf{Over}_l(\mathcal{S}, I)$ and $\mathsf{Under}_l(\mathcal{S}, I)$ with a Boolean parameter *over* that tells either an over- or under-approximation is to be computed. The approximation depends on parameter l.

The set of residual configurations that is a partially evaluated approximation is collected in variable R starting from the initial configuration I. Untreated configurations are kept in a set T ('to treat'). The algorithm terminates when $T = \emptyset$.

At each step an arbitrary untreated configuration c is picked up from T and *driving step* is performed: the successors of c are evaluated by function Drive and each new configuration c' is processed as follows:

1. The current configuration c' is checked whether it is covered or not by one of the configurations collected in R: $\nexists \bar{c} \in R \colon \bar{c} \sqsupseteq c'$. If covered (such $\bar{c}$ exists) there is no need to proceed c' further, as its descendants are covered by the descendants of the existing configuration $\bar{c}$.
2. In the case of under-approximation, it is checked whether the current configuration c' is not to be generalized by comparing it with $g = \mathsf{Generalize}_l(c')$. If it is to be generalized, the configuration c' is not proceeded further. In such a way, configurations reachable from the initial configuration I without generalization are collected in R. They form an under-approximation.
3. The (possibly) generalized current configuration g is added to the sets R and T. Additionally, the configurations from R and T covered by the new one are deleted from the sets.

Algorithm 1: $\mathsf{Approximate}_l(over, \mathcal{S}, I)$, $\mathsf{Over}_l(\mathcal{S}, I)$: $\mathsf{Under}_l(\mathcal{S}, I)$, Building over- and under-approximations of a set of reachable states of a transition system

Data: $\mathcal{S}$ a transition system
Data: I an initial configuration
Data: l an integer parameter of generalization
Data: $over$ = **true** an over-approximation, **false** an under-approximation
Result: R an under- or over-approximation of a set of reachable states

$\mathsf{Approximate}_l(over, \mathcal{S}, I)$
  $R \leftarrow \{I\}$
  $T \leftarrow \{I\}$
  while $T \neq \emptyset$ **do**
    select some configuration $c \in T$
    $T \leftarrow T \setminus \{c\}$
    foreach $c' \in \mathsf{Drive}(\mathcal{S}, c)$ **do**
      $g \leftarrow \mathsf{Generalize}_l(c')$
      if $\nexists \bar{c} \in R\colon \bar{c} \sqsupseteq c' \wedge (over \vee g = c')$ **then**
        $R \leftarrow R \cup \{g\} \setminus \{\bar{c} \in R \mid \bar{c} \sqsubset g\}$
        $T \leftarrow T \cup \{g\} \setminus \{\bar{c} \in T \mid \bar{c} \sqsubset g\}$
  return R

$\mathsf{Over}_l(\mathcal{S}, I) = \mathsf{Approximate}_l(\mathbf{true}, \mathcal{S}, I)$
$\mathsf{Under}_l(\mathcal{S}, I) = \mathsf{Approximate}_l(\mathbf{false}, \mathcal{S}, I)$

The algorithm $\mathsf{Reachable}(\mathcal{S}, I, U)$ with this definition of approximation functions and the Drive and $\mathsf{Generalize}_l$ functions from the previous section does not always terminate for an arbitrary counter system $\mathcal{S}$ but do terminate for a monotonic counter system and an upward-closed target set U.

3.3 Supercompilation of Transition Systems

In this section we define two classic supercompilation algorithms for well-quasi-ordered transition systems.

Supercompilation is usually understood as an equivalence transformation of programs, transition systems, etc., from a *source* one to a *residual* one. However, for the purpose of this paper the supercompilation algorithms presented here returns a part of information sufficient to extract the residual set of configurations rather than a full representation of a residual transition system.

Two algorithms, Algorithm 2 ScpL and Algorithm 3 ScpU, have very much in common. They take a transition system $\mathcal{S}$, a quasi-order $\prec$ on the set of configurations and an initial configuration I, and return a residual tree, which represents an over-approximation of the set of states reachable from the initial configuration I. The order $\prec$ is a parameter that controls the execution of the algorithms, their termination and influences resulting residual trees. If the order

Algorithm 2: ScpL: Supercompilation of a quasi-ordered transition system with lower-node generalization.

Data: $\mathcal{S}$ a transition system
Data: I an initial configuration
Data: $\prec$ a quasi-order on configurations, a binary whistle
Result: $\mathcal{T}$ a residual tree

```
ScpL(S, I, ≺)
    T ← ⟨N, B, n_0, C⟩ where N = {n_0}, B = ∅, C(n_0) = I, n_0 a new node
    T ← {n_0}
    while T ≠ ∅ do
        select some node n ∈ T
        T ← T \ {n}
        if ∃n̄ ∈ N: C(n̄) ⊒ C(n) then          — terminate the current path (1)
            do nothing
        else if ∃n̄: B⁺(n̄, n) ∧ C(n̄) ≺ C(n) ∧ C(n̄) ⋡ C(n) then
                                                — generalize on whistle (2)
            n̄ ← some node such that B⁺(n̄, n) ∧ C(n̄) ≺ C(n)
            C(n) ← Generalize(C(n̄), C(n))
            mark n as generalized
            T ← T ∪ {n}
        else                                    — unfold (drive) otherwise (3)
            foreach c ∈ Drive(S, C(n)) do
                n' ← a new node
                C(n') ← c
                N ← N ∪ {n'}
                B ← B ∪ {⟨n, n'⟩}
                T ← T ∪ {n'}
    return T
```

$\mathsf{Over}_l(\mathcal{S}, I) = \textbf{let } \langle N, B, n_0, C\rangle = \mathsf{ScpL}(\mathcal{S}, I, \prec_l) \textbf{ in } \{C(n) \mid n \in N\}$
$\mathsf{Under}_l(\mathcal{S}, I) = \textbf{let } \langle N, B, n_0, C\rangle = \mathsf{ScpL}(\mathcal{S}, I, \prec_l) \textbf{ in } \{C(n) \mid n \in N \wedge$
$\qquad \forall \bar{n} \text{ s.t. } B^*(\bar{n}, n)\colon \bar{n} \text{ is not marked as generalized}\}$

is a well-quasi-order the algorithms terminates for sure. Otherwise, in general, the algorithms sometimes terminate and sometimes do not.

The residual trees are gradually constructed from the root node n_0.

The nodes are labeled with configurations by a labeling function $C : N \to \mathcal{C}$, initially $C(n_0) = I$.

Untreated leaves are kept in a set T ('to treat') in Algorithm 2 and in a stack T in Algorithm 3. The first algorithm is non-deterministic and takes leaves from set T in arbitrary order. The second algorithm is deterministic and takes leaves from stack T in the FIFO order. The algorithms terminate when $T = \emptyset$ and $T = \epsilon$ (the empty sequence) respectively.

At each step, one of three branches is executed, marked in comments as (1), (2) and (3). Branches (1) and (3) are almost identical in the two algorithms.

Algorithm 3: ScpU: Supercompilation of a quasi-ordered transition system with upper-node generalization.

Data: $\mathcal{S}$ a transition system
Data: I an initial configuration
Data: $\prec$ a quasi-order on configurations, a binary whistle
Result: $\mathcal{T}$ a residual tree

```
ScpU(S, I, ≺)
    T ← ⟨N, B, n0, C⟩ where N = [n0], B = ∅, C(n0) = I, n0 a new node
    T ← [n0]
    while T ≠ ε do
        n ← Last(T)
        T ← T \ {n}
        if ∃n̄ ∈ N: C(n̄) ⊒ C(n) then          — terminate the current path (1)
            do nothing
        else if ∃n̄: B⁺(n̄, n) ∧ C(n̄) ≺ C(n) then   — generalize on whistle (2)
            n̄ ← the highest node such that B⁺(n̄, n) ∧ C(n̄) ≺ C(n)
            C(n̄) ← Generalize(C(n̄), C(n))
            mark n̄ as generalized
            T ← RemoveSubtreeExceptRoot(n̄, T)
            T ← T \ {n | B⁺(n̄, n)}              — drop nodes lower than n̄
            T ← Append(T, n̄)
        else                                    — unfold (drive) otherwise (3)
            foreach c ∈ Drive(S, C(n)) do
                n' ← a new node
                C(n') ← c
                N ← N ∪ {n'}
                B ← B ∪ {⟨n, n'⟩}
                T ← Append(T, n')
    return T

Over_l(S, I)  = let ⟨N, B, n0, C⟩ = ScpU(S, I, ≺_l) in {C(n) | n ∈ N}
Under_l(S, I) = let ⟨N, B, n0, C⟩ = ScpU(S, I, ≺_l) in {C(n) | n ∈ N ∧
                    ∀n̄ s.t. B*(n̄, n): n̄ is not marked as generalized}
```

- Branches (1): if a configuration $C(\bar{n})$ more general than the current one $C(n)$ exists in the already constructed tree, the current path is terminated and nothing is done.
- Branches (3): if the conditions on branches (1) and (2) do not hold, a *driving step* is performed: the successors of the current configuration $C(n)$ are evaluated by the function Drive; for each new configuration c a new node n' is created; edges from the current node n to the new ones are added to the tree and the new nodes are added to set (or respectively, stack) T of untreated nodes.
- Branches (2) check whether on the path to the current node n (call it *lower*) there exists a node $\bar{n}$ (call it *upper*) with the configuration $C(\bar{n})$ which is

less than the current one $C(n)$, generalize the two configurations and assign the generalized configuration to the lower node in Algorithm 2 ScpL and to the upper node in Algorithm 3 ScpU. In the latter case the nodes below $\bar{n}$ are deleted from the residual tree and from stack T of untreated nodes. The nodes where generalization has been performed are marked as 'generalized'. These marks are used in the Algorithm 0.

Over- and under-approximations $\mathsf{Over}_l(\mathcal{S}, I)$ and $\mathsf{Under}_l(\mathcal{S}, I)$ are extracted from the residual tree. The over-approximation is the set of all configurations in the residual tree. The under-approximation is comprised of the configurations developed from the initial configuration without generalization (where all nodes on the path are not marked as 'generalized').

The two supercompilation algorithms always terminate for a quasi-ordered transition systems with a set of configuration such that every configuration has finitely many generalizations. The algorithm $\mathsf{Reachable}(\mathcal{S}, I, U)$ with these definitions of approximation functions and the proper definition of the Drive and $\mathsf{Generalize}_l$ functions, does not terminate for an arbitrary transition system, even for an well-structured transition systems. It terminates for so called *degenerated* transition systems (see discussion in Section 4), which strongly monotonic counter systems belong too, as well as for monotonic counter systems, which may be not degenerated, but guarantee the termination as well.

4 Why Multi-Result Supercompilation Matters?

In this section we recap the ideas, on which solving the reachability problems (reachability and coverability) is based, and argue that multi-result supercompilation is powerful enough to solve the reachability problems for transition systems under certain conditions. The main problem of multi-result supercompilation is the blow-up of the exhaustive search of a suitable residual graph. An efficient multi-result supercompiler should prune as much extra search branches as possible. Possibilities to do so depend on the properties of transition system under analysis.

Papers [6,7] and the PhD thesis [5] by G. Geeraerts demonstrate that (and how) a kind of multi-result supercompilation solves the coverability problem for well-structured transition systems (WSTS) with some combinatorial search. Also they showed that several classes of monotonic counter systems belong to a "degenerated" case, which allows for limited enumeration of residual graphs. In [13,14] we strengthened this result by showing that this is true for *all* monotonic counter systems, provided the counter system and the driving function are good enough ($[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \subseteq \downarrow\mathsf{Post}(\mathcal{S}, c)$).

Let us discuss when the reachability problem for transition systems can be solved in principle, when and how multi-result supercompilation can find a solution, how the combinatorial search can be pruned based on the properties of a transition system, a set of configurations and driving and generalization functions.

But first, notice an important difference between single- and multi-result supercompilation. One of the main properties of a single-result supercompiler is its termination. The respective property of multi-result supercompilation is the fairness of enumeration of residual graphs. The notion of fairness depends on the problem under consideration: a fair multi-result supercompiler should enumerate either all residual graphs, or at least a subset of them sufficient to solve the problem.

4.1 Reachability Problem for Transition Systems

To implement a supercompiler for a transition system $\mathcal{S}$ one needs a set of configurations $\mathcal{C}$, a driving function Drive, and a generalization function $\mathsf{Generalize}$ that enumerates generalizations of a given configuration. Solving the reachability problem by supercompilation (as well as any forward analysis) is based on two ideas:

1. The supercompiler returns a residual set of configurations $R \subset \mathcal{C}$ which is a fixed point of the driving function:

$$[\![I]\!] \subseteq [\![R]\!] \wedge [\![\mathsf{Drive}(\mathcal{S}, R)]\!] \subseteq [\![R]\!].$$

2. To prove that a target set U is unreachable from I we check:

$$[\![R]\!] \cap U = \emptyset.$$

This is a necessary condition: if such R does not exist, no supercompiler can solve the reachability problem even if U is unreachable from I indeed.

This observation allows us to see the most essential limitation of the supercompilation method for solving the reachability problem: if the set of configurations $\mathcal{C}$ includes the representations of all such fixed points then we may expect of multi-result supercompilation to solve the problem when such a solution exists; if not we cannot expect. Designing such a complete set of configurations faces a conflict with the requirement that any configuration should have finitely many generalizations, which is used in traditional single-result supercompilers to guarantee termination. It is also used in the supercompilation-like algorithms presented above since they iteratively call the single-result supercompilers. In multi-result supercompilation, where the finiteness of the whole process is not expected (as it enumerates infinitely many residual graphs), this restriction of the finiteness of the number of generalizations could be lifted. However, this topic is not studied yet.

Thus, multi-result supercompilation could, in principle, solve the reachability problem when a solution is representable in form of a fixed point of Drive. The main problem is the exponential blow-up of the search space.

4.2 Pruning Enumeration of Residual Graphs

Do *all* residual sets are actually needed?

The first step of pruning the search space is based on the monotonicity of Drive (as a set function) and is applicable to any forward analysis solving the reachability problem: It is sufficient for a multi-result supercompiler to return not all fixed points R such that $R \supseteq I$, rather for every such fixed point R it should return at least one fixed point $R' \subseteq R$ such that $R' \supseteq I$.

This allows for a multi-result supercompiler to consider at each step of driving only the most specific generalizations among suitable ones. Unfortunately, there are many of them in general case, and fortunately, there is just one in our algorithms for monotonic counter systems.

This idea is utilized in the 'Expand, Enlarge and Check' (EEC) method by G. Geeraerts et al. The idea suffices to formulate the algorithmic schema and to prove that it terminates and solves the reachability problem when driving is perfect and the solution is representable in $\mathcal{C}$. (It is representable indeed for well-structured transition system.)

4.3 Ordered and Well-Structured Transition Systems

Now let us turn to transition systems with ordered sets of states (at least quasi-ordered) and to the coverability problem, which implies the target set of states is upward-closed. Two properties of transition systems may be formulated in terms of the ordering:

1. the order may be a well-quasi-order;
2. the transition system may be monotonic w.r.t. this order.

How does these properties influence the problem of the completeness of the set of configurations and the requirements of driving?

The well-quasi ordering of the set of states allows for a finite representation of all downward-closed sets. This is based on that any upward-closed subset of a well-quasi-ordered set has a finite number of *generators*, its minimal elements that uniquely determine the subset. A downward-closed set is the complement of some upward-closed set, and hence the generators of the complement determines it as well. However, such a "negative" representation may be inconvenient in practice (to implement driving), and G. Geeraerts et al. required that there exists a set of constructive objects called *limits* such that any downward closed set is representable by a finite number of limits. In terms of supercompilation, the limits are non-ground configurations.

The monotonicity of the transition system allows for using only downward-closed sets as configurations. That is, for solving the coverability problem, the Drive function may generalize configurations downwards at each step: $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \subseteq {\downarrow}\mathsf{Post}(\mathcal{S}, \{c\})$, rather than be perfect: $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] = \mathsf{Post}(\mathcal{S}, \{c\})$.

These are the main ideas the EEC algorithmic schema for solving the coverability problem for WSTS is based upon, except the last one described in the next subsection.

4.4 Coverability Problem for WSTS: EEC Schema of Algorithms

The 'Expand, Enlarge and Check' algorithmic schema (EEC) suggests to split enumeration of residual graphs in two levels. Our algorithm with the Approximate function fits well the EEC schema and is actually its simplest instance. Refer to it as an example.

To define an EEC algorithm one selects an expanding sequence of finite sets of configurations such that $\mathcal{C}_l \subseteq \mathcal{C}_{l+1}$ and $\mathcal{C} = \bigcup_l \mathcal{C}_l$. For example, for counter systems Algorithm Approximate uses $\mathcal{C}_l = \{1, \ldots, l, \omega\}^k$, sets of configurations with coordinates not greater than l or equal to ω .

An EEC algorithm consists of a top level loop and a multi-result supercompilation-like algorithm invoked in each iteration with the requirement that only configurations from $\mathcal{C}_l$ are used in construction of the set of residual graphs. Since $\mathcal{C}_l$ is finite, the set of residual graphs is finite as well, hence each iteration surely terminates.

Thus, the lower level of enumeration of residual graphs is performed in each iteration, and the upper level of enumeration is organized by the top level loop.

Notice that the other two Algorithms ScpL and ScpU do not fit the EEC schema exactly, since the sets of configurations are not fixed in advance. But they also forcedly restrict the sets of possible residual graphs explored in each iteration by making the set of graphs finite for a given transition system and a given initial configuration with the use of the well-quasi-order $\preccurlyeq_l$ parameterized by integer l.

It is an open problem for future work to device direct multi-result supercompilation algorithms that efficiently enumerate residual graphs in one process rather than in sequential calls of a single-result supercompiler. In [12] an example of such a monolithic algorithm obtained by manually fusing the top level loop with Algorithm Approximate is presented.

4.5 Coverability Problem for Monotonic Counter Systems

We saw that in the general case after each driving step the exploration of all most specific suitable generalizations is required in order not to loose residual graphs. However, there may be such a case that the most specific generalization (represented as a finite set of allowed configurations) is just one. G. Geeraerts [5,6] calls this case *degenerated*. This depends on the specifics of a transition system and the sets of configurations $\mathcal{C}_l$. In [5,6] such sets of configurations are called *perfect* and it is proved that for strongly monotonic counter systems sets of configurations $\mathcal{C}_l = \{1, \ldots, l, \omega\}^k$ are perfect.

Our proofs of termination of the above algorithms [13,14] shows that the requirement of the strong monotonicity can be weaken to the general monotonicity.

Thus, in the degenerated case of monotonic counter systems many residual graphs are produced due to the top level loop only, while in the loop body the use of a single-result supercompiler is sufficient.

5 Related Work

Supercompilation. This research originated from the experimental work by A. Nemytykh and A. Lisitsa on verification of cache-coherence protocols and other models by means of the Refal supercompiler SCP4 [18,19,20,21]. It came as a surprise that all of the considered correct models had been successfully verified rather than some of the models had been verified while others had not, as is a common situation with supercompiler applications. It was also unclear whether the evaluation of the heuristic parameter to control generalization of integers discovered by A. Nemytykh could be automated. Since then the theoretical explanation of these facts was an open problem.

In invited talk [22] A. Lisitsa and A. Nemytykh reported that supercompilation with upper-node generalization and without the restriction of generalization (i.e., with $l = 0$) was capable of solving the coverability problem for ordinary Petri nets, based on the model of supercompilation presented in their paper [21].

In this paper the problem has been solved for a larger class, and the sense of the generalization parameter has been uncovered. However the problem to formally characterize some class of non-monotonic counter systems verifiable by the same algorithm, which the other part of the successful examples belongs to, remains open.

We regard the iterative invocation of single-result supercompilation with a varying parameter as a domain-specific instance of multi-result supercompilation suggested by I. Klyuchnikov and S. Romanenko [15]. As they argue and as this paper demonstrates, multi-result supercompilation is capable of significantly extending the class of program properties provable by supercompilation.

Partial Deduction. Similar work to establish a link between algorithms in Petri net theory and program specialization has been done in [8,16,17]. Especially close is the work [17] where a simplified version of partial deduction is put into one-to-one correspondence with the Karp&Miller algorithm [9] to compute a coverability tree of a Petri net. Here a Petri net is implemented as a (non-deterministic) logic program and partial deduction is applied to produce a specialized program from which a coverability set can be directly obtained.

(Online) partial deduction and supercompilation has many things in common. The method of [17] can be transferred from partial deduction to supercompilation, and our work is a step forward in the same direction after [17].

Petri Nets and Transition Systems. Transition systems and their subclasses—Petri nets and counter systems—have been under intensive study during last decades: [1,2,4,5,6,7,9], just to name a few. Supercompilation resembles forward analysis algorithms proposed in the literature.

A recent achievement is an algorithmic schema referred to as 'Expand, Enlarge and Check' (EEC). In paper [6] and in the PhD thesis by G. Geeraerts [5] a proof is given that any algorithm that fits EEC terminates on a well-structured

transition systems (WSTS) and an upper-closed target set and solves the coverability problem.

The first of the presented algorithm fits the EEC schema, and could be proved correct by reduction to EEC. Other two Algorithms ScpL and ScpU do not fit EEC exactly, but are very close.

Algorithm 2 ScpL can be seen as a further development of the classic Karp&Miller algorithm [9] to compute a coverability set of a Petri net, and Algorithm 3 ScpU resembles the *minimal coverability tree* (MCT) algorithm by A. Finkel [4] (in which an error has been found [7]) and later attempts to fix it [7,23].[10]

6 Conclusion

We presented three versions of supercompilation-based algorithms, which solve the coverability problem for monotonic counter systems. Although the algorithms are rather short they present the main notions of supercompilation: configurations, driving and configuration analysis of two kinds—with lower-node and upper-node generalization.

The idea of multi-result supercompilation was demonstrated by these algorithms and future work to develop more powerful domain-specific multi-result supercompilers that would solve the coverability problem for well-structured transition systems as well as the reachability problem for some specific classes of non-monotonic transition systems, was discussed. This seems impossible with single-result supercompilation.

Acknowledgements. I am very grateful to Sergei Abramov, Robert Glück, Sergei Grechanik, Arkady Klimov, Yuri Klimov, Ilya Klyuchnikov, Alexei Lisitsa, Andrei Nemytykh, Anton Orlov, Sergei Romanenko, Artem Shvorin, Alexander Slesarenko and other participants of the Moscow Refal seminar for the pleasure to collaborate with them and exchange ideas on supercompilation and its applications. My work and life have been greatly influenced by Valentin Turchin whom we remember forever.

References

1. Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, July 27–30, 1996*, pages 313–321. IEEE Computer Society, 1996.
2. Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 10(5):401–424, 2008.

[10] The master thesis by K. Luttge [23] is beyond my reach. But its essence is explained in the PhD thesis by G. Geeraerts [5, pages 172–174].

3. Ed Clarke, Irina Virbitskaite, and Andrei Voronkov, editors. *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*. Springer, 2012.
4. Alain Finkel. The minimal coverability graph for Petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993, Papers from the 12th International Conference on Applications and Theory of Petri Nets, Gjern, Denmark, June 1991*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer, 1993.
5. Gilles Geeraerts. *Coverability and Expressiveness Properties of Well-Structured Transition Systems*. PhD thesis, Université Libre de Bruxelles, Belgique, May 2007.
6. Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. *Journal of Computer and System Sciences*, 72(1):180–203, 2006.
7. Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set of Petri nets. In K.S. Namjoshi et al., editor, *Proceedings of ATVA'07 – 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2007.
8. Robert Glück and Michael Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia, July 6-9, 1999. Proceedings*, volume 1755 of *Lecture Notes in Computer Science*, pages 93–100. Springer, 2000.
9. Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
10. Andrei V. Klimov. *JVer Project: Verification of Java programs by the Java Supercompiler*. Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2008. : `http://pat.keldysh.ru/jver/`.
11. Andrei V. Klimov. A Java Supercompiler and its application to verification of cache-coherence protocols. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2010.
12. Andrei V. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In R. Glück M. Bulyonkov, editor, *International Workshop on Program Understanding (PU 2011), July 2–5, 2011, Novososedovo, Russia.*, pages 25–32. Novosibirsk: Ershov Institute of Informatics Systems, 2011.
13. Andrei V. Klimov. Yet another algorithm for solving coverability problem for monotonic counter systems. In Valery Nepomnyaschy and Valery Sokolov, editors, *Second Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV'11, St. Petersburg, Russia, June 12–13, 2011*, pages 59–67. Yaroslavl State University, 2011.
14. Andrei V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In Clarke et al. [3], pages 193–209.
15. Ilya Klyuchnikov and Sergei Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In Clarke et al. [3], pages 207–223.

16. Michael Leuschel and Helko Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In John W. Lloyd et al., editor, *Computational Logic – CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2000.
17. Michael Leuschel and Helko Lehmann. Solving coverability problems of Petri nets by partial deduction. In *Proceedings of the 2nd international ACM SIGPLAN conference on principles and practice of declarative programming, September 20-23, 2000, Montreal, Canada*, pages 268–279. ACM, 2000.
18. Alexei P. Lisitsa and Andrei P. Nemytykh. Towards verification via supercompilation. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), 25-28 July 2005, Edinburgh, Scotland, UK*, pages 9–10. IEEE Computer Society, 2005.
19. Alexei P. Lisitsa and Andrei P. Nemytykh. *Experiments on verification via supercompilation.* Program Systems Instiute, Russian Academy of Sciences, 2007. : `http://refal.botik.ru/protocols/`.
20. Alexei P. Lisitsa and Andrei P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.
21. Alexei P. Lisitsa and Andrei P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.
22. Alexei P. Lisitsa and Andrei P. Nemytykh. Solving coverability problems by supercompilation. Invited talk. In *The Second Workshop on Reachability Problems in Computational Models (RP08), Liverpool, UK, September 15–17, 2008*, 2008.
23. K. Luttge. *Zustandsgraphen von Petri-Netzen.* Humboldt-Universität zu Berlin, Germany, 1995.
24. Andrei P. Nemytykh. The supercompiler SCP4: General structure. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003. Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2003.
25. Valentin F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
26. Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657. Springer, 1980.
27. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
28. Valentin F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.