# Automatic Verification of Counter Systems via Domain-Specific Multi-Result Supercompilation*

Andrei V. Klimov, Ilya G. Klyuchnikov, and Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

**Abstract.** We consider an application of supercompilation to the analysis of counter systems. Multi-result supercompilation enables us to find the best versions of the analysis by generating a set of possible results that are then filtered according to some criteria. Unfortunately, the search space may be rather large. However, the search can be drastically reduced by taking into account the specifics of the domain. Thus, we argue that a combination of domain-specific and multi-result supercompilation may produce a synergistic effect. Low-cost implementations of domain-specific supercompilers can be produced by using prefabricated components provided by the MRSC toolkit.

## 1 Introduction

Supercompilation is a program manipulation technique that was originally introduced by V. Turchin in terms of the programming language Refal (a first-order applicative functional language) [37], for which reason the first supercompilers were designed and developed for the language Refal [35,39,29].

Further development of supercompilation led to a more abstract reformulation of supercompilation and to a better understanding of which details of the original formulation were Refal-specific and which ones were universal and applicable to other programming languages [32,33,3]. It particular, it was shown that supercompilation is as well applicable to non-functional programming languages (imperative and object-oriented ones) [6].

Also, despite the fact that from the very beginning supercompilation was regarded as a tool for both program optimization and program analysis [36], the research in supercompilation, for a long time, was primarily focused only on program optimization. Recently, however, we have seen a revival of interest in the application of supercompilation to inferring and proving properties of programs [25,12,10].

Multi-result supercompilation is a technique of constructing supercompilers that, given an input program, are able to produce a set of residual programs, rather than just a single one [13,8].

* 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.

The purpose of the present work is to show, by presenting a concrete example, that multi-result, domain-specific supercompilation is not a theoretical curiosity, but rather a workhorse having certain advantages over general-purpose, single-result (deterministic) supercompilation. Some of the reasons are the following.

- Tautologically speaking, a general-purpose supercompiler should deal with programs written in a general-purpose subject language that, by definition, is not dedicated to a particular problem domain. Thus, for a given domain, the subject language may be too sophisticated, but, on the other hand, lacking in certain features.
- In cases where supercompilation is used for the purposes of analysis and verification, the problem of reliability and correctness of the supercompiler itself becomes rather actual. Can we trust the results produced by a (large and intricate) general-purpose supercompiler?
- On the other hand, it is only natural for a domain-specific supercompiler to accept programs in a domain-specific language (DSL) that provides domain-specific operations and control constructs whose mathematical properties may be known in advance. This domain knowledge can be hard-coded into the supercompiler, thereby increasing its power and enabling it to achieve better results at program analysis and transformation, as compared to "pure" supercompilation.
- The subject language of a domain-specific supercompiler may be very limited in its means of expression, in which case some parts of the supercompiler can be drastically simplified. For example, in some areas there is no need to deal with nested function calls in configurations. The simplifications of that kind increase the reliability of the supercompiler and make it easier to prove its correctness by formal methods (as was shown by Krustev [15]).
- The implementation of a domain-specific supercompiler may be very cheap if it is done on the basis of prefabricated components (for example, by means of the MRSC toolkit [13,14]), so that the costs of implementation can be reduced by an order of magnitude, as compared to implementations of general-purpose supercompilers.

## 2 Analyzing the behavior of systems by means of supercompilation

One of the approaches to the analysis of systems consists in representing systems by programs. Thus the task of analyzing the behavior of a system is reduced to the task of inferring and analyzing the properties of a program $p$.

The program $p$, modeling the original system, may in turn be analyzed using *the transformational approach*, in which case $p$ is transformed into another program $p'$ (equivalent to $p$), so that some non-obvious properties of $p$ become evident in the program $p'$.

For example, suppose that the original program $p$ is complicated in structure and contains statements `return False`. Can this program return `False`? This question is not easy to answer. Suppose, however, that by transforming $p$ we get a trivial program $p'$ whose body consists of a single statement `return True`. Then we can immediately conclude that $p'$ can never return `False`. Since $p'$ is equivalent to $p$, it implies that $p$ also can never return `False`.

Initial states:

$$(i, 0, 0, 0)$$

Transitions:

$$\begin{aligned}
&(i, e, s, m) \,|\, i \geq 1 \longrightarrow (i - 1, 0, s + e + m + 1, 0)\\
&(i, e, s, m) \,|\, e \geq 1 \longrightarrow (i, e - 1, s, m + 1)\\
&(i, e, s, m) \,|\, s \geq 1 \longrightarrow (i + e + s + m - 1, 1, 0, 0)\\
&(i, e, s, m) \,|\, i \geq 1 \longrightarrow (i + e + s + m - 1, 1, 0, 0)
\end{aligned}$$

Unsafe states:

$$\begin{aligned}
&(i, e, s, m) \,|\, m \geq 2\\
&(i, e, s, m) \,|\, s \geq 1 \wedge m \geq 1
\end{aligned}$$

Fig. 1: MESI protocol: a protocol model in form of a counter system

One of the applications of the transformational approach is the verification of communication protocols modeled by counter systems [1]. For instance, let us consider a model of the MESI protocol in form of a counter system that is informally described in Fig. 1.

The states of the system are represented by quadruples of natural numbers. The specification of the system includes the description of a set of *initial states* and a set of *transition rules* of the form

$$(i, e, s, m) \,|\, p \longrightarrow (i', e', s', m')$$

where $i$, $e$, $s$, $m$ are variables, $p$ is a condition on the variables which must be fulfilled for the transition to be taken, and $i'$, $e'$, $s'$, $m'$ are expressions that may contain the variables $i$, $e$, $s$, $m$.

The system is non-deterministic, as several rules may be applicable to the same state.

The specification of a protocol model also includes the description of a set of *unsafe states*. The analysis of such a protocol model is performed for the purpose of solving the *reachability problem*: in order to prove that unsafe states are not reachable from the initial states.

As was shown by Leuschel and Lehmann [21,18,19,16], reachability problems for transition systems of that kind can be solved by program specialization techniques. The system to be analyzed can be specified by a program in a domain-specific language (DSL) [21]. The DSL program is then transformed into a Prolog program by means of a classic partial evaluator LOGEN [5,17] by using the first Futamura projection [2]. The Prolog program thus obtained is then transformed by means of ECCE [22,20], a more sophisticated specializer, whose internal workings are similar to those of supercompilers.

Lisitsa and Nemytykh [24,25,26] succeeded in verification of a number of communication protocols by means of the supercompiler SCP4 [29,28,27]. The input language of SCP4 is Refal, a first-order functional language developed by Turchin [36]. SCP4 is a descendant of earlier supercompilers for Refal [35,36,39,37,38].

According to the approach by Lisitsa and Nemytykh, protocol models are represented as Refal programs. For instance, the MESI protocol [1,27,23] is modeled by the Refal program in Fig. 2. The program is written in such a way that, if an unsafe state is reached, it returns the symbol `False` and terminates.

The supercompiler SCP4 takes this program as input and produces the residual program shown in Fig. 3, which contains no occurrences of the symbol `False`. This suggests the conclusion that the residual program is unable to return `False`. However, strictly speaking, this argument is not sufficient in the case of a dynamically typed language (like Lisp, Scheme and Refal): a program can still return `False`, even if `False` does not appear in the text of the program. Namely, the program may receive `False` via its input data and then transfer it to the output. And, indeed, "engineering solutions" of that kind are extremely popular with hackers as a means of attacking web-applications [34]. Fortunately, there exist relatively simple data flow analysis techniques that are able to compute an upper approximation to the set of results that can be produced by a function, even for dynamically-typed languages [4], and which are able to cope with Refal programs like that in Fig. 3.

```
*$MST_FROM_ENTRY;
*$STRATEGY Applicative;
*$LENGTH 0;

$ENTRY Go {e.A (e.I) =
  <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}

Loop {
  () (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Result (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>;
  (s.A e.A) (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Loop (e.A)
    <RandomAction s.A
    (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>>;
}

RandomAction {
* rh Trivial
* rm
A (Invalid s.1 e.1) (Modified e.2) (Shared e.3) (Exclusive e.4) =
   (Invalid e.1)  (Modified )  (Shared s.1 e.2 e.3 e.4 ) (Exclusive );
* wh1 Trivial
*wh2
B (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive s.4 e.4) =
  (Invalid e.1)(Modified s.4  e.2)(Shared e.3)(Exclusive e.4);
* wh3
C (Invalid e.1)(Modified e.2)(Shared s.3 e.3)(Exclusive e.4) =
  (Invalid e.4 e.3 e.2 e.1)(Modified  )(Shared )(Exclusive s.3);
* wm
D (Invalid s.1 e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
  (Invalid e.4 e.3 e.2 e.1)(Modified  )(Shared )(Exclusive s.1);
}

Result{
(Invalid e.1)(Modified s.2 e.2)(Shared s.3 e.3)(Exclusive e.4) = False;
(Invalid e.1)(Modified s.21 s.22 e.2)(Shared e.3)(Exclusive e.4) = False;

(Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) = True;
}
```

Fig. 2: MESI protocol: a protocol model in form of a Refal program

```
* InputFormat: <Go e.41 >
$ENTRY Go {
 (e.101 )  = True ;
 A e.41 (s.103 e.101 )  = <F24 (e.41 ) (e.101 ) s.103 > ;
 D e.41 (s.104 e.101 )  = <F35 (e.41 ) (e.101 ) s.104 > ;
}

* InputFormat: <F35 (e.109 ) (e.110 ) s.111 e.112 >
F35 {
 () (e.110 ) s.111 e.112  = True ;
 (A e.109 ) (e.110 ) s.111 s.118 e.112  =
    <F24 (e.109 ) (e.112 e.110 ) s.118 s.111 > ;
 (A e.109 ) (s.119 e.110 ) s.111  = <F24 (e.109 ) (e.110 ) s.119 s.111 >;
 (B ) (e.110 ) s.111 e.112  = True ;
 (B A e.109 ) (e.110 ) s.111 s.125 e.112  =
    <F24 (e.109 ) (e.112 e.110 ) s.125 s.111 > ;
 (B A e.109 ) (s.126 e.110 ) s.111  =
    <F24 (e.109 ) (e.110 ) s.126 s.111> ;
 (B D e.109 ) (e.110 ) s.111 s.127 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.127 > ;
 (B D e.109 ) (s.128 e.110 ) s.111  =
    <F35 (e.109 ) (s.111 e.110 ) s.128> ;
 (D e.109 ) (e.110 ) s.111 s.120 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.120 > ;
 (D e.109 ) (s.121 e.110 ) s.111  = <F35 (e.109 ) (s.111 e.110 ) s.121 >;
}

* InputFormat: <F24 (e.109 ) (e.110 ) s.111 e.112 >
F24 {
 () (e.110 ) s.111 e.112  = True ;
 (A e.109 ) (s.114 e.110 ) s.111 e.112  =
    <F24 (e.109 ) (e.110 ) s.114 s.111 e.112 > ;
 (C e.109 ) (e.110 ) s.111 e.112  =
    <F35 (e.109 ) (e.110 ) s.111 e.112 >;
 (D e.109 ) (s.115 e.110 ) s.111 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.115 > ;
}
```

Fig. 3: MESI protocol: the residual Refal program.

# 3 Domain-specific supercompilation as a means of analysis

## 3.1 Drawbacks of general-purpose supercompilation

An obvious advantage of general-purpose supercompilation is just its being general-purpose. Upon having designed and implemented a general-purpose supercompiler, we can apply it to various problems again and again, in theory, without any extra effort. However, there are some disadvantages associated with general-purpose supercompilation. As an example, let us consider the use of the specializer SCP4 for the analysis of counter systems [28,24,25,26], in which case the tasks for the supercompiler are formulated as Refal programs [27,23]. This causes the following inconveniences.

- Natural numbers in input programs are represented by strings of star symbols, and their addition by string concatenation. This representation is used in order to take into account the behavior of some general-purpose algorithms embedded in SCP4 (the whistle, the generalization), which know nothing about the specifics of counter systems. Thus, the representation of data has to conform to subtle details of the internal machinery of SCP4, rather than comply with the problem domain.
- The programs modeling counter systems have to be supplemented with some directions (in form of comments) for SCP4, which control some aspects of its behavior. In this way SCP4 is given certain information about the problem domain, and without such directions, residual programs produced by SCP4 would not possess desirable properties. Unfortunately, in order to be able to give right directions to SCP4, the user needs to understand its internals.
- There remains the following question: to what extent can we trust the results of the verification of counter systems, obtained with the aid of SCP4? The internals of SCP4 are complicated and the source code is big. Thus the problem of verifying SCP4 itself seems to be intractable.

## 3.2 Domain-specific algorithms of supercompilation

Which techniques and devices embedded into SCP4 are really essential for the analysis of counter systems? This question was investigated by Klimov who has developed a specialized supercompilation algorithm that was proven to be correct, always terminating, and able to solve reachability problems for a certain class of counter systems [6,7,10,8].

It was found that, in the case of counter systems, supercompilation can be simplified in the following ways.

- The structure of configurations is simpler, as compared to the case of classic supercompilation for functional languages.
  - There are no nested function calls.
  - There are no multiple occurrences of variables.

  - A configuration is a tuple, all configurations consisting of a fixed number of components.
  - A component of a configuration is either a natural number $n$, or the symbol $\omega$ (a wildcard, representing an arbitrary natural number).
- The termination of the supercompilation algorithm is ensured by means of a very simple generalization algorithm: if a component of a configuration is a natural number $n$, and $n \geq l$, where $l$ is a constant given to the supercompiler as one of its input parameters, then $n$ must be replaced with $\omega$ (and in this way the configuration is generalized). It can be easily seen that, given an $l$, the set of all possible configurations is finite.

### 3.3 Domain-specific supercompilers for domain-specific languages

The domain-specific supercompilation algorithm developed by Klimov [6,7,10,8] turned out to be easy to implement with the aid of the MRSC toolkit [13,14]. The simplicity of the implementation is due to the following.

- We have only to implement a simplified supercompilation algorithm for a domain-specific language, rather than a sophisticated general-purpose algorithm for a general-purpose language.
- The MRSC toolkit is based on the language Scala that provides powerful means for implementing embedded DSLs. The implementations can be based either on interpretation (shallow embedding) or on compilation (deep embedding).
- The MRSC toolkit provides prefabricated components for the construction of graphs of configurations (by adding/removing graph nodes), for manipulating sets of graphs and pretty-printing graphs. When implementing a supercompiler, it is only necessary to implement the parts that depend on the subject language and on the structure of configurations.

When we develop a domain-specific supercompiler, it seems logical for its subject language also to be domain-specific, rather than general-purpose.

In this case the formulations of problems that are submitted to the supercompiler can be concise and natural, since the programs written in the subject DSL may be very close to the informal formulations of these problems. For instance, consider the 3 specifications of the MESI protocol: the informal one (Fig. 1), the one in form of a Refal program (Fig. 2), and the one written in a domain-specific language (Fig. 4).

A protocol model encoded as a DSL program is, in terms of Scala, an object implementing the trait `Protocol` (Fig. 5). Thus this program is not a first-order value (as is implicitly assumed in the classic formulation of the Futamura projections [2]), but rather is a mixture of first-order values (numbers, lists) and higher-order values (functions). This approach is close to the DSL implementation technique known as "shallow embedding".

By supercompiling the model of the MESI protocol, we obtain the graph of configurations shown in Fig. 6.

```
object MESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0)
  val rules: List[TransitionRule] = List(
    {case List(i, e, s, m) if i>=1 => List(i-1, 0, s+e+m+1, 0)},
    {case List(i, e, s, m) if e>=1 => List(i, e-1, s, m+1)},
    {case List(i, e, s, m) if s>=1 => List(i+e+s+m-1, 1, 0, 0)},
    {case List(i, e, s, m) if i>=1 => List(i+e+s+m-1, 1, 0, 0)})
  def unsafe(c: Conf) = c match {
    case List(i, e, s, m) if m>=2 => true
    case List(i, e, s, m) if s>=1 && m>=1 => true
    case _ => false
  }
}
```

Fig. 4: MESI protocol: a protocol model in form of a DSL program

```
package object counters {
  type Conf = List[Expr]
  type TransitionRule = PartialFunction[Conf, Conf]
  ...
}

sealed trait Expr { ... }

trait Protocol {
  val start: Conf
  val rules: List[TransitionRule]
  def unsafe(c: Conf): Boolean
}
```

Fig. 5: DSL for specifying counter systems: the skeleton of its implementation in Scala

## 4 Using multi-result supercompilation for finding short proofs

When analyzing a transition system, a graph of configurations produced by supercompilation describes an upper approximation of the set of reachable states. This graph can be transformed in a human-readable proof that any reachable state satisfy some requirements (or, in other words, cannot be “unsafe”).

The smaller the graph the easier it is to understand, and the shorter is the proof that can be extracted from this graph. However, a traditional single-result supercompiler returns a single graph that may not be the smallest one.

However, a multi-result supercompiler returns a set of graphs, rather than a single graph. Thus the set of graphs can be filtered, in order to select “the best”

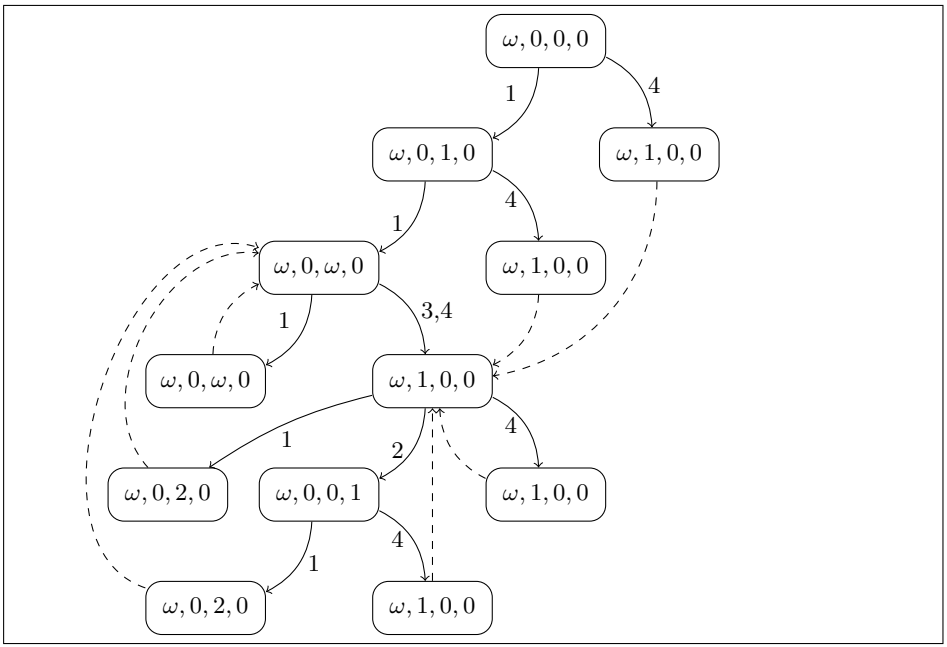


Fig. 6: MESI protocol: the graph of configurations (single-result supercompilation)

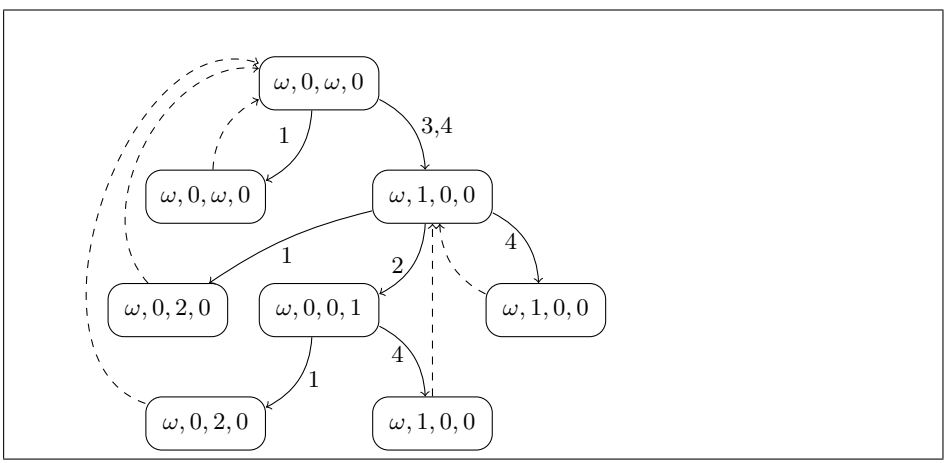


Fig. 7: MESI protocol: the minimal graph of configurations (multi-result supercompilation)

ones. In the simplest case, “the best” means “the smallest”, although the graphs can be filtered according to other criteria (for example, we may select the graphs that are, in a sense, “well-structured”, to transform them into “well-structured” proofs).

For example, the graph produced for the MESI protocol (see Fig. 6) by single-result positive supercompilation [32,33] contains 12 nodes, while, by filtering the set of graphs produced by multi-result supercompilation, we can find the graph shown in Fig. 7, which only contains 8 nodes.

The point is that single-result supercompilers, especially those meant for program optimization try to avoid the generalization of configurations by all means. This strategy is reasonable and natural in the case of optimizing supercompilation, but it is unlikely to produce minimal graphs. In the above example, single-result supercompilation starts from the configuration $(\omega, 0, 0, 0)$ and, after a while, comes to the configuration $(\omega, 0, \omega, 0)$, which is more general than $(\omega, 0, 0, 0)$.

However, multi-result supercompilation, by "a sudden flash of inspiration", starts with generalizing the initial configuration. From the viewpoint of optimizing supercompilation, this action appears to be strange and pointless. But it leads to producing the graph shown in Fig. 7, which is a *subgraph* of the graph in Fig. 6

Another interesting point is that in the case of single-result supercompilation the whistle and the generalization algorithm are tightly coupled, since generalization is performed at the moments when the whistle blows, in order to ensure termination of supercompilation. For this reason, the whistle and the generalization algorithm, to be consistent, have to be developed together. In the case of multi-result supercompilation, however, the whistle and generalization are completely decoupled. In particular, configurations can be generalized at any moment, even if the whistle does not regard the situation as dangerous. As a result, a multi-result supercompiler can find graphs that are not discovered by single-result supercompilation.

As an example, let us consider the verification of the MOESI protocol (Fig. 8). The graph produced by single-result supercompilation (Fig. 9) contains 20 nodes, while the graph discovered by multi-result supercompilation (Fig. 10) contains 8 nodes only.

This is achieved due to a "brilliant insight" of multi-result supercompilation that the initial configuration $(\omega, 0, 0, 0, 0)$ can be immediately generalized to the configuration $(\omega, 0, \omega, 0, \omega)$. This leads to an 8-node graph that is *not contained* as a subgraph in the 20-node graph produced by single-result supercompilation. Note that the configuration $(\omega, 0, \omega, 0, \omega)$ *does not appear* in the 20-node graph, and, in general, the structure of the graphs in Fig. 9 and Fig.10 is completely different.

It should be noted that there exists a domain-specific supercompilation algorithm for counter systems (developed by Klimov [10]) that, in some cases, is able to reduce the number of nodes in the graphs, because, as compared to general-purpose optimizing supercompilers, it generalizes configurations more energetically. For instance, for the MESI protocol, it generates the same graph (Fig. 7), as that produced by multi-result supercompilation.

The idea of Klimov's algorithm [10] is the following. Suppose, in the process of supercompilation there appears a configuration $c$, such that $c$ is an instance

```
case object MOESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0, 0)
  val rules: List[TransitionRule] =
    List({ // rm
      case List(i, m, s, e, o) if i>=1 =>
        List(i-1, 0, s+e+1, 0, o+m)
    }, { // wh2
      case List(i, m, s, e, o) if e>=1 =>
        List(i, m+1, s, e-1, o)
    }, { // wh3
      case List(i, m, s, e, o) if s+o>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    }, { // wm
      case List(i, m, s, e, o) if i>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    })

  def unsafe(c: Conf) = c match {
    case List(i, m, s, e, o) if m>=1 && e+s+o>=1 => true
    case List(i, m, s, e, o) if m>=2 => true
    case List(i, m, s, e, o) if e>=2 => true
    case _ => false
  }
}
```

Fig. 8: MOESI protocol: a protocol model as a DSL program

of a configuration $c'$ that is already present in the graph. Then $c$ has to be generalized to $c'$.

Unfortunately, this algorithm is not always successful in generating minimal graphs. For example, in the case of the MOESI protocol, multi-result supercompilation finds such configurations that are not appear in the process of classic positive supercompilation [32,33].

The table in Fig. 11 compares the results of verifying 13 communication protocols. The column SC shows the number of nodes in the graphs produced by classic single-result positive supercompilation, and the column MRSC shows the number of nodes in the graphs produced by straightforward multi-result supercompilation derived from positive supercompilation [32,33] according to the scheme described in [14]. It is evident that, practically always, multi-result supercompilation is able to find graphs of smaller size than those produced by single-result supercompilation.

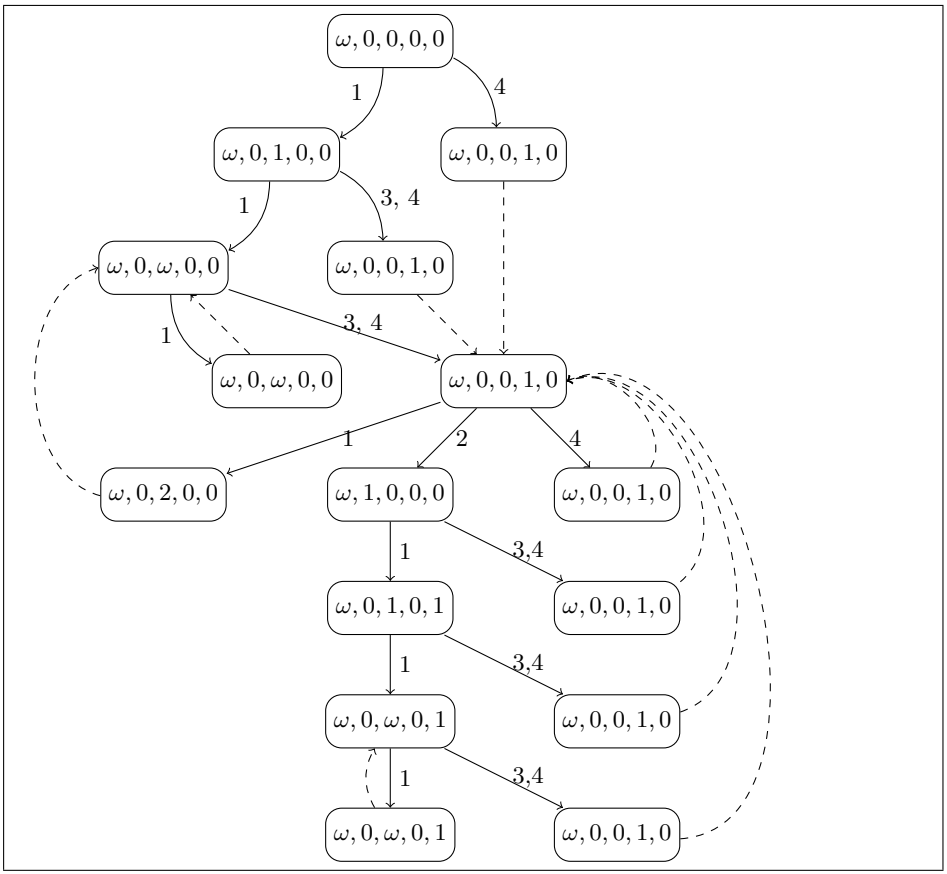


Fig. 9: MOESI protocol: the graph of configurations (single-result supercompilation)

## 5 Domain-specific residualization of the graphs of configurations

Traditionally, general-purpose supercompilation is performed in two steps. At the first step, there is produced a finite graph of configurations. At the second step, this graph is "residualized", i.e. transformed into a residual program. For example, the supercompiler SCP4 generates residual programs in the language Refal (see Fig. 3)

When the purpose of supercompilation is the analysis of counter systems, residual programs are not executed, but analyzed to see whether they possess some desirable properties. For example, as regards the program in Fig. 3, all that matters is whether it can return `False`, or not? This can be determined either by asking a human's opinion, or, in a more rigorous way, by submitting the program to a data flow analysis algorithm [4].

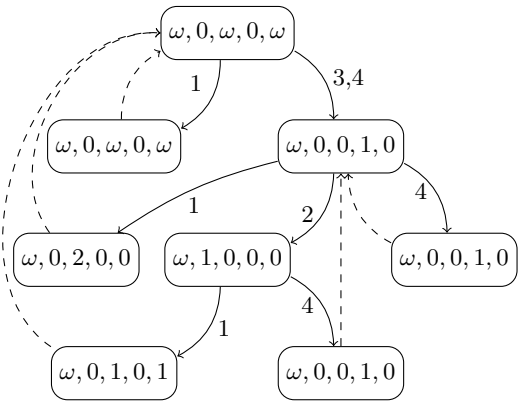


Fig. 10: MOESI protocol: the minimal graph of configurations (multi-result supercompilation)

| | SC | MRSC |
|---|---|---|
| Synapse | 11 | 6 |
| MSI | 8 | 6 |
| MOSI | 26 | 14 |
| MESI | 14 | 9 |
| MOESI | 20 | 9 |
| Illinois | 15 | 13 |
| Berkley | 17 | 6 |
| Firefly | 12 | 10 |
| Futurebus | 45 | 24 |
| Xerox | 22 | 13 |
| Java | 35 | 25 |
| ReaderWriter | 48 | 9 |
| DataRace | 9 | 5 |

Fig. 11: Single-result vs. multi-result supercompilation: the size of proofs (represented by graphs of configurations)

However, when using supercompilation for the analysis of counter systems, we can take an easier way: it turns out that graphs of configurations are easier to analyze, than residual programs. Thus, we can dispense with the generation of residual programs for the purposes of separating the good outcomes of supercompilation from the bad ones. Moreover, upon selecting a graph with desirable properties, instead of generating a residual program in a programming language, we can transform the graph into a script for a well-known proof assistant [16], in order to verify the results obtained by supercompilation.

In particular, we have implemented a domain-specific supercompiler that transforms graphs of configurations into scripts for the proof assistant Isabelle [30]. A script thus produced specifies the reachability problem for a communi-

```
theory mesi
imports Main
begin

inductive mesi :: "(nat * nat * nat * nat) => bool" where
  "mesi (i, 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i, 0, Suc (s + e + m), 0)" |
  "mesi (i, Suc e, s, m) ==> mesi (i, e, s, Suc m)" |
  "mesi (i, e, Suc s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)"

inductive unsafe :: "(nat * nat * nat * nat) => bool" where
  "unsafe (i, e, s, Suc (Suc m))" |
  "unsafe (i, e, Suc s, Suc m)"


inductive mesi' :: "(nat * nat * nat * nat) => bool" where
  "mesi'(_, Suc 0, 0, 0)" |
  "mesi'(_, 0, 0, Suc 0)" |
  "mesi'(_, 0, _, 0)"

lemma inclusion: "mesi c ==> mesi' c"
  apply(erule mesi.induct)
  apply(erule mesi'.cases | simp add: mesi'.intros)+
done

lemma safety: "mesi' c ==> ~unsafe c"
  apply(erule mesi'.cases)
  apply(erule unsafe.cases | auto)+
done

theorem valid: "mesi c ==> ~unsafe c"
  apply(insert inclusion safety, simp)
done

end
```

Fig. 12: MESI protocol: the script for the proof assistant Isabelle produced by the domain-specific supercompiler.

cation protocol and, in addition, a number of tactics that instruct Isabelle how to formally prove that all reachable states are safe.

For example, in the case of the MESI protocol, there is produced the script shown in Fig. 12. The script comprises the following parts.

- Inductive definitions of the predicate `mesi`, specifying the set of reachable states, and the predicate `unsafe`, specifying the set of unsafe states (`unsafe`), which are the same (modulo notation) as in the source DSL program.
- An inductive definition of the predicate `mesi'`, specifying a set of states that is an upper approximation to the set specified by `mesi`. This definition (modulo notation) enumerates configurations appearing in the graph in Fig. 7. In order to reduce the size of the script, there is applied a simple optimization: if the graph contains two configurations $c'$ and $c$, where $c'$ is an instance of $c$, then $c'$ is not included into the definition of the predicate `mesi'`. The definition of `mesi'` is the most important (and non-trivial) part of the script.
- The lemma `inclusion`, asserting that any reachable state belongs to the set specified by `mesi'`, or, in other words, for any state $c$, `mesi` $c$ implies `mesi'` $c$.
- The lemma `safety`, asserting that all states in the set specified by `mesi'` are safe, or, in other words, for any state $c$, `mesi'` $c$ implies ¬`unsafe` $c$.
- The main theorem: any reachable state is safe. In other words, for all states $c$, `mesi` $c$ implies ¬`unsafe` $c$. This trivially follows from the lemmas `inclusion` and `safety`).

The fundamental difference between the definitions of `mesi` and `mesi'` is that `mesi` is defined inductively, while the definition of `mesi'` is just an enumeration of a finite number of cases. For this reason, the lemma `safety` can be proven by tedious, yet trivial case analysis.

Thus the rôle of supercompilation in the analysis of counter systems amounts to generalizing the description of the set of reachable states in such a way that proving the safety of reachable states becomes trivial. Therefore, supercompilation can be regarded as a useful supplement to other theorem-proving and verification techniques.

# 6 Improving the efficiency of supercompilation by taking into account the specifics of the domain

## 6.1 Exploiting the mathematical properties of domain-specific operations

As was shown by Klimov [6,7,10,8], in the case of supercompilation for counter systems it is sufficient to deal with configuration of the form $(a_1, \ldots, a_n)$, whose each component $a_i$ is either a natural number $N$, or the symbol $\omega$. As regards driving, it is sufficient to deal with tests of the form either $e = N$, or $e \geq N$, where $N$ is a natural number and $e$ is an arithmetic expression that can only contain the operators $+$, $-$, natural numbers and $\omega$. All necessary operations over such expressions are easy to implement in terms of the language Scala (see Fig.13).

But, if we use a general-purpose supercompiler, dealing with programs in a general-purpose language, the supercompiler does not have any knowledge about the problem domain and the operations over domain-specific data structures. For example, when the supercompiler SCP4 is used for the verification of protocols,

```
package object counters {
  ...
  implicit def intToExpr(i: Int): Expr = Num(i)
}

sealed trait Expr {
  def +(comp: Expr): Expr
  def -(comp: Expr): Expr
  def >=(i: Int): Boolean
  def ===(i: Int): Boolean
}

case class Num(i: Int) extends Expr {
  def +(comp: Expr) = comp match {
    case Omega  => Omega
    case Num(j) => Num(i + j)
  }
  def -(comp: Expr) = comp match {
    case Omega  => Omega
    case Num(j) => Num(i - j)
  }
  def ===(j: Int) = i == j
  def >=(j: Int) = i >= j
}

case object Omega extends Expr {
  def +(comp: Expr) = Omega
  def -(comp: Expr) = Omega
  def >=(comp: Int) = true
  def ===(j: Int) = true
}
```

Fig. 13: Counter systems: operations over components of configurations implemented in Scala.

natural numbers have to be encoded as strings of the star symbol, and addition of natural numbers as concatenation of strings (see Fig. 2 and 3). As a consequence, it becomes difficult (both for humans and for supercompilers) to see that a program operates on natural numbers.

### 6.2 Handling non-determinism in a direct way

When a supercompiler is a general-purpose one, its subject language is usually designed for writing deterministic programs. This causes some inconveniences in cases where supercompilation is used for the analysis of non-deterministic systems. If a model of a non-deterministic system has to be encoded as a deterministic program, there arises the need for using various tricks and artificial

workarounds, which, certainly, complicates the program and obscures its meaning.

Consider, for example, the model of the MESI protocol in Fig. 2, encoded as a Refal program. The entry point of the program is the function `Go` which takes 2 parameters: `e.A` and `e.I` [27,23].

```
$ENTRY Go {e.A (e.I) =
    <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}
```

The parameter `e.I` is used for building the initial state, while the parameter `e.A` has been artificially introduced in order to simulate non-determinism. Since the rules describing the transition system are not mutually exclusive, more than one rule can be applicable at the same time, and the value of the parameter `e.A` is a sequence of rule names, prescribing which rule must be applied at each step.

Unfortunately, this additional parameter, pollutes not only the source program, but also the configurations emerging during supercompilation and, finally, the residual program (see Fig. 3), thereby obscuring its meaning.

However, if a model of a non-deterministic system is encoded as a program in a non-deterministic language (see Fig. 4), then there disappears the need for using tricks and workarounds related to non-determinism. Also note that non-determinism, by itself, does not create additional problems for supercompilation, as, unlike an ordinary interpreter, a supercompiler has to consider *all* possible ways of executing a program (for a given set of initial states) [18,19].

### 6.3 Filtering graphs of configurations, rather than residual programs

As has been discussed in Section 2, multi-result supercompilation can be used for finding residual programs satisfying some criteria. Since a multi-result supercompiler may produce hundreds, or even thousands of residual programs, there is a need for automatic filtering of residual programs.

For example, when applying a general-purpose supercompiler for the analysis of counter systems, we need a filter for selecting residual problems that are certain not to return `False`, and such a filter can be constructed on the basis of well-known data-flow analysis algorithms [4].

In the case of domain-specific supercompilation, however, "residual programs" may not be programs in traditional sense of the word. For instance, the result produced by analyzing a counter system can be presented as a script for an automatic proof assistant (see Section 5). So the filtering of programs should be replaced with the filtering of something else.

Fortunately, it turns out that filtering of the final results of supercompilation can be replaced with filtering of graphs of configurations. Moreover, taking into account the specifics of the domain allows the process of filtering to be optimized by discarding some graphs that are in construction, without waiting for them to be completed. This can considerably reduce the amount of work performed by a multi-result supercompiler, because discarding an incomplete graph prunes the whole set of graphs that would be generated by completing the discarded graph.

| | | SC1 | SC2 | SC3 | SC4 | SC5 |
|---|---|---|---|---|---|---|
| Synapse | completed | 48 | 37 | 3 | 3 | 1 |
| | pruned | 0 | 0 | 0 | 0 | 2 |
| | commands | 321 | 252 | 25 | 25 | 15 |
| MSI | completed | 22 | 18 | 2 | 2 | 1 |
| | pruned | 0 | 0 | 0 | 0 | 1 |
| | commands | 122 | 102 | 15 | 15 | 12 |
| MOSI | completed | 1233 | 699 | 6 | 6 | 1 |
| | pruned | 0 | 0 | 0 | 0 | 5 |
| | commands | 19925 | 11476 | 109 | 109 | 35 |
| MESI | completed | 1627 | 899 | 6 | 3 | 1 |
| | pruned | 0 | 0 | 27 | 20 | 21 |
| | commands | 16329 | 9265 | 211 | 70 | 56 |
| MOESI | completed | 179380 | 60724 | 81 | 30 | 2 |
| | pruned | 0 | 0 | 0 | 24 | 36 |
| | commands | 2001708 | 711784 | 922 | 384 | 126 |
| Illinois | completed | 2346 | 1237 | 2 | 2 | 1 |
| | pruned | 0 | 0 | 21 | 17 | 18 |
| | commands | 48364 | 26636 | 224 | 74 | 61 |
| Berkley | completed | 3405 | 1463 | 30 | 30 | 2 |
| | pruned | 0 | 0 | 0 | 0 | 14 |
| | commands | 26618 | 12023 | 282 | 282 | 56 |
| Firefly | completed | 2503 | 1450 | 2 | 2 | 1 |
| | pruned | 0 | 0 | 2 | 2 | 3 |
| | commands | 39924 | 24572 | 47 | 25 | 21 |
| Futurebus | completed | - | - | - | - | 4 |
| | pruned | - | - | - | - | 148328 |
| | commands | - | - | - | - | 516457 |
| Xerox | completed | 317569 | 111122 | 29 | 29 | 2 |
| | pruned | 0 | 0 | 0 | 0 | 1 |
| | commands | 5718691 | 2031754 | 482 | 482 | 72 |
| Java | completed | - | - | - | - | 10 |
| | pruned | - | - | - | - | 329886 |
| | commands | - | - | - | - | 1043563 |
| ReaderWriter | completed | 892371 | 402136 | 898 | 898 | 6 |
| | pruned | 0 | 0 | 19033 | 19033 | 1170 |
| | commands | 24963661 | 11872211 | 123371 | 45411 | 3213 |
| DataRace | completed | 51 | 39 | 8 | 8 | 3 |
| | pruned | 0 | 0 | 0 | 0 | 4 |
| | commands | 360 | 279 | 57 | 57 | 31 |

Fig. 14: Resources consumed by different versions of the multi-result supercompiler

As regards counter systems, the specifics of the domain are the following. The predicate `unsafe` must be monotonic with respect to configurations: for all configurations $c$ and $c'$, such that $c$ is an instance of $c'$, `unsafe` $c$ implies `unsafe` $c'$. Another point is that if a configuration $c$ has appeared in a graph of configurations, it can be removed by supercompilation only by replacing $c$ with a more general configuration $c'$ (such that $c$ is an instance of $c'$). Thus, if $c$ is unsafe, it can only be replaced with an unsafe configuration (due to the monotonicity of the predicate `unsafe`). Therefore, if a graph contains an unsafe configuration, it can be discarded immediately, since all graphs produced by completing that graph would also contain unsafe configurations.

The detection of unsafe configurations can be performed at various places in the supercompilation algorithm, and the choice of such places bears great influence on the efficiency of multi-result supercompilation.

The next optimization, depending on the specifics of the domain, takes into account the properties of the set of all possible generalizations of a given configuration $c$.

Namely, all generalizations of $c$ can be obtained by replacing some numeric components of $c$ with $\omega$. Thus, the configuration $(0, 0)$ can be generalized in 3 ways, to produce $(\omega, 0)$, $(0, \omega)$ and $(\omega, \omega)$. Note that $(\omega, \omega)$ is a generalization with respect to $(\omega, 0)$ and $(0, \omega)$.

A naïve multi-result supercompilation algorithm, when trying to rebuild a configuration $c$ by replacing it with a more general configuration $c'$, considers all possible generalizations of $c$ immediately. If a generalization $c'$ is not a maximal one, after a while, it will be, in turn, generalized. For instance, $(\omega, 0)$, and $(0, \omega)$ will be generalized to $(\omega, \omega)$. Thus the same graph of configurations will be produced 3 times: by immediately generalizing $(0, 0)$ to $(\omega, \omega)$, and by generalizing $(0, 0)$ to $(\omega, \omega)$ in two steps, via $(\omega, 0)$, and $(0, \omega)$.

The number of graphs, considered during multi-result supercompilation, can be significantly reduced, by allowing only minimal generalization of a configuration, which can be obtained by replacing a single numeric component in a configuration with $\omega$.

We have studied the performance of 5 variations of a supercompilation algorithm for counter systems: SC1, SC2, SC3, SC4 and SC5. Each variant differs from the previous one in that it introduces an additional optimization.

- SC1. Filtering and generation of graphs are completely decoupled. A graph is examined by the filter only after having been completed. Thus, no use is made of the knowledge about domain-specific properties of generalization (its decomposability into elementary steps) and the predicate `unsafe` (its monotonicity). This design is modular, but inefficient.
- SC2. The difference from SC1 is that, when rebuilding a configuration $c$, SC2 only considers the set of "minimal" generalizations (produced by replacing a single component of $c$ with $\omega$).
- SC3. The difference from SC2 is that the configurations produced by generalization are checked for being safe, and the unsafe ones are immediately discarded.

- SC4. The difference from SC3 is that the configurations that could be produced by driving a configuration $c$ are checked for being safe. If one or more of the new configurations turn out to be unsafe, driving is not performed for $c$.
- SC5. The difference from SC4 is that the graphs that are too large are discarded, without completing them. Namely, the current graph is discarded if there is a complete graph that has been constructed earlier, and whose size is smaller than that of the current graph. (Note that, due to the optimizations introduced in SC2, SC3 and SC4, all configurations in completed graphs are guaranteed to be safe.)

The optimization introduced in SC5 is typical for algorithms in the field of artificial intelligence, where it is known as "pruning" [31].

The table in Fig. 14 shows the resources consumed by the 5 versions of the supercompiler while verifying 13 communication protocols. For each protocol, the row *completed* shows the number of completed graphs that have been produced (with possible repetitions), the row *pruned* shows the number of discarded incomplete graphs, and the row *commands* shows the number of graph building steps that have been performed during supercompilation.

In the case of the protocols Futurebus and Java, the data are only given for the version SC5, as the resource consumption by the other versions of the supercompiler turned out to be too high, for which reason data were not obtained.

The data demonstrate that the amount of resources consumed by multi-result supercompilation can be drastically reduced by taking into account the specifics of the problem domain.

## 7 Conclusions

Multi-result supercompilation is not a theoretical curiosity, but rather a workhorse that, when exploited in a reasonable way, is able to produce results of practical value.

- The use of multi-result supercompilation in the field of the analysis and verification of transition systems improves the understandability of the results, by considering various versions of the analysis and selecting the best ones.
- The use of multi-result supercompilation allows the whistle and the algorithm of generalization to be completely decoupled, thereby simplifying the structure of the supercompiler. This, in turn, makes it easier to ensure the correctness of the supercompiler.

The usefulness of domain-specific supercompilation is due to the following.

- The tasks for a domain-specific supercompiler can be written in a domain-specific language that is better at taking into account the specifics of the problem domain, than a general-purpose language. (For example, this DSL may be non-deterministic, or provide domain-specific data types and operations.)

- In the case of a domain-specific supercompiler, the machinery of supercompilation can be simplified, since, in a particular domain, some complexities of general-purpose supercompilation may be of little usefulness.
- The efficiency of multi-result supercompilation can be improved by early discarding of unsatisfactory variants of supercompilation.
- The MRSC toolkit allows domain-specific multi-result supercompilers to be manufactured at low cost, making them a budget solution, rather than a luxury.

Thus, the combination of domain-specific and multi-result supercompilation produces a synergistic effect: generating multiple results gives the opportunity to select the best solutions to a problem, while taking into account the specifics of the problem domain reduces the amount of resources consumed by multi-result supercompilation.

## Acknowledgements

The authors express their gratitude to the participants of the Refal seminar at Keldysh Institute for useful comments and fruitful discussions.

## References

1. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Form. Methods Syst. Des.*, 23:257–301, November 2003.
2. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
3. N. D. Jones. The essence of program transformation by partial evaluation and driving. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, PSI '99, pages 62–79, London, UK, UK, 2000. Springer-Verlag.
4. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3):120–136, 2007.
5. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 238–262. Springer, 1996.
6. A. Klimov. An approach to supercompilation for object-oriented languages: the Java supercompiler case study. In *First International Workshop on Metacomputation in Russia*, 2008.
7. A. V. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2009.
8. A. V. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In *International Workshop on Program Understanding, PU 2011, Novososedovo, Russia, July 2–5, 2011*, pages 25–32. Ershov Institute of Informatics Systems, Novosibirsk, 2011.

9. A. V. Klimov. Yet another algorithm for solving coverability problem for monotonic counter systems. In V. Nepomnyaschy and V. Sokolov, editors, *Second Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV 2011, St. Petersburg, Russia, June 12–13, 2011*, pages 59–67. Yaroslavl State University, 2011.
10. A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In E. Clarke, I. Virbitskaite, and A. 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*, pages 193–209. Springer, 2012.
11. A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit. Preprint 24, Keldysh Institute of Applied Mathematics, 2012.
12. I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.
13. I. Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In *Ershov Informatics Conference*, volume 7162 of *LNCS*, pages 210–226, 2012.
14. I. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011.
15. D. Krustev. A simple supercompiler formally verified in Coq. In *Second International Workshop on Metacomputation in Russia*, 2010.
16. H. Lehmann and M. Leuschel. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce. In M. Bruynooghe, editor, *LOPSTR*, volume 3018 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.
17. M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Electr. Notes Theor. Comput. Sci.*, 30(2):157–162, 1999.
18. M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 101–115, London, UK, 2000. Springer-Verlag.
19. M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 268–279, New York, NY, USA, 2000. ACM.
20. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst.*, 20:208–258, January 1998.
21. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Selected papers from the 9th International Workshop on Logic Programming Synthesis and Transformation*, pages 62–81, London, UK, 2000. Springer-Verlag.
22. M. Leuschel and D. D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. D. Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 1996.
23. A. Lisitsa. Verification of MESI cache coherence protocol. `http://www.csc.liv.ac.uk/~alexei/VeriSuper/node5.html`.

24. A. Lisitsa and A. P. Nemytykh. Towards verification via supercompilation. *Computer Software and Applications Conference, Annual International*, 2:9–10, 2005.

25. A. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

26. A. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.

27. A. Nemytykh. SCP4 : Verification of protocols. `http://refal.botik.ru/protocols/`.

28. A. P. Nemytykh. The supercompiler SCP4: General structure. In M. Broy and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2003.

29. A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 249–260, 1996.

30. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

31. D. Poole and A. K. Mackworth. *Artificial Intelligence - Foundations of Computational Agents*. Cambridge University Press, 2010.

32. M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Dept. of Computer Science, University of Copenhagen, 1994.

33. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

34. Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

35. V. F. Turchin. A supercompiler system based on the language refal. *SIGPLAN Not.*, 14(2):46–54, 1979.

36. V. F. Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis*. Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

37. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

38. V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.

39. V. F. Turchin, R. M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 47–55, New York, NY, USA, 1982. ACM.

# A An implementation

Here we show the source code of the multi-result supercompiler for counter systems implemented by means of the MRSC toolkit. More detailed explanations about the code may be found in [11].

## A.1 Operations over configurations

First of all, the supercompiler has to perform the following operations over configurations: testing whether a configuration $c_1$ is an instance of a configuration $c_2$, and enumerating all possible generalizations of a configuration $c$. An implementation of these operations is shown in Fig. 15.

```
package mrsc.counters

object Conf {
  def instanceOf(c1: Conf, c2: Conf): Boolean =
    (c1, c2).zipped.forall((e1, e2) => e1 == e2 || e2 == Omega)

  def gens(c: Conf) =
    product(c map genExpr) - c

  def oneStepGens(c: Conf): List[Conf] =
    for (i <- List.range(0, c.size) if c(i) != Omega)
      yield c.updated(i, Omega)

  def product[T](zs: List[List[T]]): List[List[T]] = zs match {
    case Nil       => List(List())
    case x :: xs => for (y <- x; ys <- product(xs)) yield y :: ys
  }

  private def genExpr(c: Expr): List[Expr] = c match {
    case Omega            => List(Omega)
    case Num(i) if i >= 0 => List(Omega, Num(i))
    case v                => List(v)
  }
}
```

Fig. 15: Operations over configurations: testing for instances and building generalizations

The function `instanceOf` tests whether a configuration `c1` is an instance of a configuration `c2`.

The function `genExpr` generates all possible generalization of an expression (which is a component of a configuration). Note that the original expression is

```
trait GraphRewriteRules[C, D] {
  type N = SNode[C, D]
  type G = SGraph[C, D]
  type S = GraphRewriteStep[C, D]
  def steps(g: G): List[S]
}

case class GraphGenerator[C, D]
  (rules: GraphRewriteRules[C, D], conf: C)
  extends Iterator[SGraph[C, D]] { ... }
```

Fig. 16: MRSC "middleware" for supercompiler construction

included into the set of generalization. The set of generalization of the symbol $\omega$ contains only the symbol $\omega$, while the set of generalizations of a number $N$ consists of two elements: $N$ and $\omega$.

The function `gens` generates the set of all possible generalizations of a configuration $c$. Note that $c$ is not included into this set.

The function `oneStepGens` generates the set of all generalizations of a configurations $c$ that can be produced by generalizing a single component of $c$. This function will be used in the optimized version of the supercompiler shown in Fig. 20.

### A.2 Graph builder

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

Notation:

$g$ – a current graph of configurations
$\beta$ – a current node in a graph of configurations
$c$ – a configuration in a current node $\beta$

Fig. 17: Multi-result supercompilation specified by rewrite rules

Technically, a supercompiler written using MRSC is based upon two components shown in Fig. 16: `GraphRewriteRules` and `GraphGenerator` [14].

The trait `GraphRewriteRules` declares the method `steps`, which is used in the main loop of supercompilation for obtaining all graphs that can be derived from a given incomplete graph $g$ by applying the rewrite rules Fold, Drive and Rebuild [14] shown in Fig. 17. Namely, `steps(`$g$`)` returns a list of "graph rewrite steps" [14]. Then the graph generator applies each of these "steps" to the graph $g$ to produce the collection of the descendants of $g$.

A concrete supercompiler is required to provide an implementation for the method `steps`. The class `GraphGenerator`, by contrast, is a ready-to-use component: it is a constituent part of any supercompiler built on top of MRSC.

```
package mrsc.counters

class MRCountersRules(protocol: Protocol, l: Int)
  extends GraphRewriteRules[Conf, Unit] {

  override def steps(g: G): List[S] =
    fold(g) match {
      case None    => rebuild(g) ++ drive(g)
      case Some(s) => List(s)
    }

  def fold(g: G): Option[S] = {
    val c = g.current.conf
    for (n <- g.completeNodes.find(n => instanceOf(c, n.conf)))
      yield FoldStep(n.sPath)
  }

  def drive(g: G): List[S] =
    if (dangerous(g)) List()
    else List(AddChildNodesStep(next(g.current.conf)))

  def rebuild(g: G): List[S] =
    for (c <- gens(g.current.conf))
      yield RebuildStep(c): S

  def dangerous(g: G): Boolean =
    g.current.conf exists
      { case Num(i) => i >= l; case Omega => false }

  def next(c: Conf): List[(Conf, Unit)] =
    for (Some(c) <- protocol.rules.map(_.lift(c)))
      yield (c, ())
}
```

Fig. 18: Graph rewrite rules: an implementation for counter systems

In the case of supercompilation for counter systems the method `steps` can be straightforwardly implemented as shown in Fig. 18.

The methods `fold`, `drive` and `rebuild` correspond to the rewrite rules Fold, Drive and Rebuild [14]. Since the rewrite rules are independent from each other, the body of the method (steps) could have been defined in the following trivial way:

```
fold(g) ++ rebuild(g) ++ drive(g)
```

However, we have preferred to slightly optimize the implementation by taking into account that the rule Fold is mutually exclusive with the rules Drive and Rebuild. Another subtle point is that, in general, the rule Fold is non-deterministic, because the current configuration may be foldable to several configurations in the graph. Thus, the rule Fold may be applicable in zero, one or more ways. However, in the case of counter systems, all variants of folding are equally good. For this reason, in the implementation in Fig. 18, the method `fold` returns no more than one variant of folding, the type of the results being `Option[S]`, rather than `List[S]`. And the rules Drive and Rebuild are only applied if `fold` returns zero results.

The implementations of the methods `fold` and `rebuild` are straightforward.

The method `dangerous` implements the whistle suggested by Klimov [10,9]: a configuration is considered as "dangerous" if it contains a number $N$, such that $N \geq l$, where $l$ is a constant given to the supercompiler as one of its input parameters.

The implementation of the method `drive` uses an auxiliary method `next`, which tries to apply all transition rules to a configuration $c$. If a rule is applicable, it returns a configuration $c'$, in which case the pair $(c', ())$ is included in the list returned by `next`. In general, this pair has the form $c', d$, where $c'$ is the new configuration and $d$ the label for the edge entering the node containing the configuration $c'$. But, in the case of counter systems, edges need not be labeled, for which reason we put the placeholder () in the second component of the pair.

### A.3 Optimizations

Fig. 19 shows the supercompiler for counter systems that has been produced from the supercompiler in Fig. 18 by implementing the aforementioned optimizations. Technically, the improved supercompiler is implemented as the class `FastMRCountersRules`, which is a subclass of `MRCountersRules`.

The main loop of the optimized supercompiler is shown in Fig. 20. Complete graphs are produced by the iterator `graphs` by demand. Since the goal is to find a graph of minimum size, the variable `minGraph` contains the smallest of the graphs that have been encountered.

Now let us consider the internals of the class `FastMRCountersRules`.

The variable `maxSize` holds the maximum size of graphs that are worth considering: if the supercompiler encounters a graph whose size exceeds `maxSize`, this graph is discarded (see the definition of the method `steps`).

```
package mrsc.counters

class FastMRCountersRules(protocol: Protocol, l: Int)
  extends MRCountersRules(protocol, l) {

  var maxSize: Int = Int.MaxValue

  override def drive(g: G): List[S] =
    for (AddChildNodesStep(ns) <- super.drive(g)
         if ns.forall(c =>!protocol.unsafe(c._1)))
      yield AddChildNodesStep(ns)

  override def rebuild(g: G): List[S] =
    for (c <- oneStepGens(g.current.conf) if !protocol.unsafe(c))
      yield RebuildStep(c): S

  override def steps(g: G): List[S] =
    if (protocol.unsafe(g.current.conf) || size(g) > maxSize)
      List()
    else
      super.steps(g)

  private def size(g: G) =
    g.completeNodes.size + g.incompleteLeaves.size
}
```

Fig. 19: Graph rewrite rules: an optimized implementation for counter systems

```
    val rules = new FastMRCountersRules(protocol, l)
    val graphs = GraphGenerator(rules, protocol.start)

    var minGraph: SGraph[Conf, Unit] = null
    for (graph <- graphs) {
      val size = graphSize(graph)
      if (size < rules.maxSize) {
        minGraph = graph
        rules.maxSize = size
      }
    }
```

Fig. 20: Optimized implementation of the main loop of multi-result supercompilation

The method `rebuild` is redefined: now, instead of considering all possible generalization (produced by the method `gens`), it only considers one-step generalizations (produced by the method `oneStepGens`).

All other modifications are related to detecting unsafe configurations: the goal is to detect unsafe configurations as soon as possible. This is achieved by applying the predicate `unsafe` to configurations at several places.