A Java Supercompiler and its Application to Verification of Cache-Coherence Protocols

Andrei V. Klimov*

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

Abstract. The Java Supercompiler (JScp) is a specializer of Java programs based on the Turchin's supercompilation method and extended to support imperative and object-oriented notions absent in functional languages.
The results of successful application of the Java Supercompiler for verification of a number of parameterized models of cache-coherence protocols are reported. Protocols are modeled in Java following the known method by G. Delzanno and experiments by A. Lisitsa and A. Nemytykh on verification of protocol models by means of the Refal Supercompiler SCP4. The part of the supercompilation method relevant to the protocol verification is described. It deals with an imperative subset of Java.
Keywords: specialization, verification, supercompilation, object-oriented languages, Java.

1 Introduction

Program specialization methods—partial evaluation [14], supercompilation [31–34], mixed computation [10], partial computation [12], etc.—have been first developed for functional and simplified imperative languages. Later the time has come for specialization of more complex practical object-oriented languages.

There are already a number of works on (*off-line*) partial evaluation of object-oriented languages [2, 1, 29, 5, 18]. However, to the best of our knowledge, our work is the first one on supercompilation of a practical object-oriented language [13, 15, 17].

The research on supercompilation for the Java language was started in 1996 together with Larry Wittie and Valentin Turchin under support by an ONR US grant (No. 00014-96-1-080).

In 1999–2003 the main part of work on construction of the first version of a Java supercompilation system was done by the author together with Arkady Klimov and Artem Shvorin under support by Supercompilers, LLC, and our

* Supported by Russian Foundation for Basic Research projects No. 08-07-00280-a and No. 09-01-00834-a and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

partners Ben Goertzel and Yuri Mostovoy who helped identify important applications for Java program specialization by supercompilation. As result, the first Java Supercompiler JScp has emerged [13, 15, 17].

During the last two years, inspired by far-reaching results by Alexei Lisitsa and Andrei Nemytykh on verification of protocol models by means of the Refal Supercompiler SCP4 [21, 23–26, 22], we extended the Java Supercompiler with the elements of the supercompilation method that were absent in it at that time, but were needed to reproduce the results in Java [16].

It is impossible to present the whole of the supercompilation method for object-oriented languages in a short conference paper. Specialization of operations on objects in JScp is discussed in another paper [15]. Since objects are not used in the protocol models coded in Java following G. Delzanno [8, 6], in this paper we describe supercompilation of the imperative subset of Java.

A novelty of this part of the supercompilation method implemented in JScp is that *breadth-first* unfolding of the graph of configurations and recursive construction of the residual code of a statement from the residual codes of nested statements is used rather than *depth-first* traversal of configuration as in other known supercompilers. This allowed us to elaborate the subtleties of each control statement one by one and simplify stepwise development of JScp. The method is applicable to many imperative languages with a sophisticated set of control statements.

Another contribution of this paper is reproduction of the results of the experiment on verification of protocols by another supercompiler (JScp instead of SCP4) for a rather different language (the object-oriented Java instead of the functional Refal). This confirms the result is based on the essence of supercompilation rather than on technical implementation details. As a consequence of the experiment the part of supercompilation method relevant to verification of protocols has been uncovered.

This paper is organized as follows. In Section 2 the part of the Java supercompilation method that is relevant to verification of protocols is described. In Section 3 the results of an experiment on verification of protocol models are shown. In Section 4 we conclude.

2 Java Supercompilation

2.1 The Notion of a Configuration

While an interpreter runs a program on a ground data, a supercompiler runs the program on a set of data. The later process is referred to as *driving* in the theory of supercompilation. A representation of a subject program state in a supercompiler is referred to as a *configuration*. We follow the general rule of construction of the notion of a configuration in a supercompiler from that of the program state in an interpreter that reads as follows: add configuration variables to the data domain, and allow the variables to occur anywhere where an ordinary ground value can occur. A configuration represents the set of states that can be obtained by replacing configuration variables with all possible values.

In the Java virtual machine, a program state consists of *global variables* (`static` fields of Java classes), a representation of *threads* and a *heap*.

In the Java Supercompiler, non-`final` global variables are not represented in a configuration, since at supercompilation time they are considered unknown and no information about them is kept. One the other hand, the values of `final static` fields are evaluated only once at the initialization stage and may be (partially) known at supercompilation time. Since they do not change after initialization and behave as constants, one copy of them is kept for all configurations.

The current version of JScp does not specialize multi-threaded programs. Hence a configuration contains only one thread now.

Like a program state in a Java virtual machine, a call stack in JScp consists of *frames* comprised of *local variables* ("environment"), an *evaluation stack* and a somehow represented *program point* ("program counter").

According to the general rule mentioned above, global variables, local variables and evaluation stack can contain *configuration variables* in addition to ground values.[1] Each configuration variable is identified by a unique integer index and carries the type of values it stands for: either one of the Java primitive types, or the reference type along with a class name and some additional information (which we don't discuss in this paper), or the string type.[2]

This is the minimum configuration stuff sufficient to implement a reasonable Java Supercompiler. However we have found this is not enough to verify protocol models coded in Java following the G. Delzanno's method. The models are parameterized by integers, and verification of the class of models considered by G. Delzanno requires propagation of restrictions on integer parameters of form $v \geq n$, where v is an integer parameter (a configuration variable in a supercompiler), n an integer constant, $n \geq 0$. To meet this class of applications, the notion of a configuration has been extended with restrictions of such kind.

Thus, the definition of a configuration in the current JScp is as follows:

- a *configuration* is a triple (*thread*, *restrictions*, *heap*);
- a *thread* is a *call stack*, that is, a sequence of *frames*;
- a *frame* is a triple (*local environment*, *evaluation stack*, *program point*);
- a *local environment* is a mapping of *local variables* to *configuration values*;
- an *evaluation stack* is a sequence of *configuration values*;
- the representation of a *program point* does not matter. It is sufficient to assume it allows us to resume supercompilation from the point;
- a *configuration value* is either a *ground value*, or a *configuration variable*. Each configuration variable carries a Java type;
- *restrictions* are a mapping *Restr* of configuration variables to predicates on their values. If a configuration variable v is not bound by the mapping,

[1] To distinguish configuration variables, which are elements of configurations, from global and local variables, which are elements of programs, we refer to the latter as *program variables* when it is not clear from the context which kind of the variable is meant.

[2] For inessential technical reasons, our representation of strings and string configuration variables differs from the general representation of reference values and variables.

$Restr(v) = \lambda x.\texttt{true}$. In the current version of JScp only restrictions of form $Restr(v) = \lambda x.(x \geq n)$, where n is an integer, on variables of the integral types of the Java language are implemented;
- we leave the notion of a *heap* unspecified here, since this paper does not deal with supercompilation of programs with objects.

JScp performs an equivalent transformation of one or more methods of a Java program. The supercompiled methods replace the original ones. Supercompilation of a method is started from an *initial configuration* that consists of one call stack frame with the formal parameters bound to fresh configuration variables.

A characteristic feature of supercompilation is that configuration variables become local variables of the residual program.

2.2 Operations on Configurations

The following three operations on configurations are used in the supercompiler.

Comparison of configurations for set inclusion represented by a substitution: we consider $C_1 \subseteq C_2$ if there exist a substitution δ that binds configuration values to configuration variables such that $C_1 = \delta C_2$ (application of substitution δ to configuration C_2 gives configuration C_1).

The substitution respects types and restrictions: for each binding $v \mapsto x$, $Vals(x) \subseteq Vals(v)$, where $Vals(v)$ is the subset of values of the type of a configuration variable v, satisfying $Restr(v)$; $Vals(x) = \{x\}$ if x is a ground value.

Generalization of configurations:

- a configuration G is a *generalization of a configuration* C if $C \subseteq G$ (that is, $\exists \delta : C = \delta G$);
- a configuration G is the most specific *generalization of two configurations* C_1 and C_2 if $C_1 \subseteq G$ and $C_2 \subseteq G$ and for all other G' satisfying this property, $G \subseteq G'$.

Homeomorphic embedding used for termination of loop unrolling similar to other supercompilers [30, 33, 34, 27]: $C_1 \trianglelefteq C_2$ if the call stacks of C_1 and C_2 have the same "shape" (their lengths are equal and the program points of each stack frame and hence the sets of local variables are the same) and the homeomorphic relation $x_1 \trianglelefteq x_2$ holds for all pairs of corresponding values x_1 from C_1 and x_2 from C_2, where $\trianglelefteq$ is the least relation satisfying:

- $v_1 \trianglelefteq v_2$ for all configuration variables v_1 and v_2 (including references). If the configuration variables have an integral type, their restrictions must embed as well, $Restr(v_1) \trianglelefteq Restr(v_2)$ (see below);
- $x_1 \trianglelefteq x_2$ for all values x_1 and x_2 of the String class unless this is switched off by the user (when he considers beneficial to generate separate configurations for different string values);
- $x_1 \trianglelefteq x_2$ for all ground values x_1 and x_2 of the same floating type;

- $n_1 \trianglelefteq n_2$ for all ground values n_1 and n_2 of the same integral type such that $0 \le k \le n_1 \le n_2$ or $0 \ge -k \ge n_1 \ge n_2$, where k is a user-defined parameter that influences the number of basic configurations and the shape of the residual graph. For verification of the protocols [16] values $k = 0$ and $k = 1$ were used (due to observation by A. Nemytykh);
- embedding of restrictions: $r_1 \trianglelefteq r_2$ if $r_1 = \lambda x.\texttt{true}$, or $0 \le n_1 \le n_2$, or $0 \ge n_1 \ge n_2$, where $r_1 = \lambda x.(x \ge n_1)$ and $r_2 = \lambda x.(x \ge n_2)$.

Thus defined homeomorphic embedding relation is a well-quasi order: for any infinite sequence of configurations C_i there exist i and j such that $i \le j$, $C_i \trianglelefteq C_j$. The proof is by reduction to the Higman-Kruskal theorem [20]: encode configurations as terms, where all variables are mapped to one symbol and integer constants are represented in the unary system, e.g., as a sequence of 1's.

2.3 Driving of Method Invocations

In the current version of JScp method invocations are either inlined, or residualized. No specialized methods are generated as in other supercompilers [27] and partial evaluators [14].

Inlining in a supercompiler is done like invocation in interpreters and inlining in ordinary compilers: A new stack frame is added to the call stack with the method parameters bound to the values of arguments, which may be configuration variables. On exit from the method the value of the expression in the `return` statement, if any, is passed to the calling stack frame as the value of the method invocation expression.

Whether to inline or not is controlled by certain JScp options (e.g., inline until call stack length is equal to a given number). In our experiments on verification all method invocations were inlined.

2.4 Driving of Expressions and Assignments

The process of partial program execution in a supercompiler is referred to as *driving*. Driving of an expression in a current configuration yields the value of the expression, a residual code, and a new configuration.

For the purpose of this paper, a Java expression consists of constants, local and global (`static`) program variables and `static` method invocations connected by unary and binary operators. Driving of an expression recursively reduces to driving of its constituents like ordinary interpretation.

Driving of a constant yields just the constant value. The result of driving of a local variable is a configuration value bound to the variable in the current configuration, and empty residual code. A `final` global variable g is processed similarly except that if the value bound to it at initialization stage is a configuration variable v, a local variable declaration statement of form t v `=` g is residualized, where t is the type of g and v. (This may produce extra repeated variable declarations, which are deleted by post-processing.) A non-`final` global variable is driven like a `final` one with a fresh configuration variable bound to it.

Each unary or binary operation is either evaluated, if there is sufficient information to produce a ground resulting value, or otherwise residualized with a fresh configuration variable v as its value in form of a local variable declaration of form $t\ v$ `=` e, where e is the expression representing residualized operation with the values of arguments substituted into it.

Restrictions are taken into account in the following way. Integer operations $v + i$ and $v - i$, where i an integer constant, v a configuration variable with restriction $\lambda x.(x \geq n)$, are residualized in form $t\ v'$ `=` $v + i$ and $t\ v'$ `=` $v - i$, and a new configuration variable v' with a restriction of form $\lambda x.(x \geq n + i)$ or $\lambda x.(x \geq n - i)$ is added to the configuration.

Integer comparisons v `==` i, v `!=` i, v `<` i, v `<=` i, v `>` i, v `>=` i and their commutative counterparts, where i is an integer ground value, v a configuration variable with restriction $\lambda x.(x \geq n)$, evaluate to `true` or `false`, when this is clear from comparison $n > i$ or $n \geq i$.

2.5 Driving of Conditional Statements

Consider a source program fragment `if (`c`)` a `else` b`;` d, where c is a conditional expression, statements a and b are branches, and a sequence of statements d a continuation executed on exit from the `if` statement.

If driving of c gives `true` or `false`, the respective branch a or b is used for further driving instead of the `if` statement.

Otherwise, the `if` statement is residualized and each of the branches is supercompiled with the initial configuration corresponding to the end of driving of the conditional c. Restrictions on the configuration variables involved in the conditional c are refined, if possible, when c has form x_1 `==` x_2, x_1 `!=` x_2, x_1 `<` x_2, x_1 `<=` x_2, x_1 `>` x_2 or x_1 `>=` x_2, where x_i is a ground integer value or a configuration variable. Additionally to the residual code of the `if` statement of form `if (`c'`)` a' `else` b', two configurations C_a and C_b corresponding to the ends of the branches a' and b' are returned.

On exit from the `if` statement there are two alternatives:

- either supercompile the statements d two times with initial configurations C_a and C_b producing residual codes d'_a and d'_b respectively, and return the residual code for the whole fragment of form `if (`c'`)` $\{a'$`;` $d'_a\}$ `else` $\{b'$`;` $d'_b\}$;
- or generalize the configurations C_a and C_b to a configuration G, where $C_a = \delta_a G$ and $C_b = \delta_b G$, δ_a and δ_b are substitutions; supercompile d with initial configuration G producing d', and return the following residual code for the whole fragment: `if (`c'`)` $\{a'$`;` $\alpha_a\}$ `else` $\{b'$`;` $\alpha_b\}$`;` d', where α_a and α_b assignments encoding substitutions δ_a and δ_b in Java.

The choice between the alternatives is made by the JScp user. For the task of protocol verification we used the more aggressive first option.

The `switch` statement is supercompiled similarly.

2.6 Configuration Analysis of Loop Statements

Proper configuration analysis is performed only for loops in the current JScp. All kinds of loops in Java are reducible to a loop of form `L: while (true)` b, where b is a loop body statement.

Four kinds of exits are possible from the source and residual code of a loop body: `throw`, `return`, `break` and `continue`. The first three kinds are terminal nodes from the viewpoint of supercompilation of the loop statement. A `throw` statement is just residualized and no more actions are taken on that branch.[3] A `return` statement is reduced to a `break` with a label of an appropriate enclosing statement. Processing of `break`s and `continue`s to a level higher than the loop statement is postponed until the corresponding level is reached. Statements `break L` along with the configurations corresponding to them are exits from the residual code of the loop statement. Residual statements `continue L` with their configurations are subject to further configuration analysis.

Let a loop statement `L: while (true)` b be supercompiled with an initial configuration C_0. First, the loop body b is supercompiled with C_0 producing residual code b_0 and the list of `continue L` statements with configurations C_i, $i \in [1..n_0]$. For those C_i that $C_i \subseteq C_0$, $C_i = \delta_i C_0$, the `continue` statements are residualized in form α_i `; continue L`, where α_i are assignments encoding the substitution δ_i.

The remaining configurations C_i, $C_i \not\subseteq C_0$, comprise a current set $Cont$ of to-be-supercompiled `continue` statements. They are points of further loop unrolling: the loop body b is supercompiled with each $C \in Cont$ and the residual code is analyzed in the same way as for C_0.

This process is repeated and a residual code in form of a tree consisting of residual loop bodies supercompiled with various initial configurations is built. Each new configuration C_i on a leaf of an unfinished tree is checked for looping-back to all of the initial configurations of the residual loop bodies on the path from C_0 to this leaf.[4]

This process terminates when the set Cont is empty. However this does not happen in general case.

Generalization and termination To guarantee termination of supercompilation of loops, we have to generalize some configurations. A criterion that tells whether to stop or to continue and selects a configuration to be generalized is called a *whistle* in the supercompilation jargon. The now most popular whistle [30, 33, 34, 27] is based on the homeomorphic embedding of the representations of configurations and the Higman-Kruskal theorem [20]. The embedding relation $\trianglelefteq$ used in JScp was described in Section 2.2.

[3] In the current JScp, `try` statements are residualized without change except some trivial cases. In future we plan to implement collecting information about possible exceptions in residual code and proper specialization of `try` statements based on this information. `try` statements are not used in the protocol models in our experiments.

[4] This is a simplification to a case where residual code is easier to represent back in Java. A more compact residual code may be obtained if the restriction that configurations for comparison are taken from the path from C_0 only, is lifted.

Before supercompilation of the loop body with a next configuration C_i, the configuration is compared for homeomorphic embedding with all of the previous initial configurations of the residual loop bodies on the path to it from C_0. If such C_j that $C_j \trianglelefteq C_i$ is found, C_j is generalized with C_i obtaining a configuration G, $C_j \subseteq G$. Then the residual subtree below C_j is erased, a sequence of assignments corresponding to the substitution δ that reduces C_j to G, $C_j = \delta G$, is inserted into the point of C_j, and supercompilation is repeated from the configuration G.

This process terminates due to that there can be only a finite number of generalizations for each configuration and due to the Higman-Kruskal theorem [20] using our homeomorphic embedding of configurations.

3 Application to Verification of Cache-Coherence Protocols

A. Lisitsa and A. Nemytykh [21–26] have found a nice class of applications solvable by supercompilation. They used the Refal Supercompiler SCP4 developed by A. Nemytkh and V. Turchin [27] and encoded in the functional language Refal the protocol models from site [6] developed by G. Delzanno [8]. They verified the protocol models by applying the compiler SCP4. The code of the models and the results of the supercompilation may be found in [24].

Here we demonstrate this method of verification with the use of Java and the Java supercompiler JScp. The protocol models in Java and the results of supercompilation are collected in [16]. The Java code of the models is rather close to the code in the domain-specific language HyTech used in [6]. We translated the code manually but the work was almost mechanical.

For the lack of space we don't describe the G. Delzanno's approach to the parameterized modeling of cache-coherence protocols. See his papers, e.g. [8], for details. Only the structure of the Java code of models is essential for demonstration of the JScp work.

A model is an extended finite state machine (EFSM) parameterized by a finite number of integer variables with positive values. State change is caused by a finite number of external actions. An action is applicable if a certain condition is met. State change is encoded in Java as assignments to the state variables with simple arithmetic expressions at the right-hand sides dependent on the previous values of the state variables. When the sequence of actions ends, the state machine turns to the final state which must satisfy a certain "safety" condition. A protocol model is correct if for any sequence of actions the final state meets the "safety" condition.

Figure 1 contains the pattern used to encode the protocol models. A model is programmed as the body of method `runModel`.

To attempt to prove the correctness of a model we supercompile the method `runModel` and observe the residual code, which is equivalent to the source code (provided the supercompiler is correct). If all `return` statements has form `return true`, we conclude the model can never reach an "unsafe" state, a state where the post-condition returns `false`.

```
class model-class-name extends ProtocolModel {
  boolean runModel(int[] actions, int[] pars) throws ModelException {
    int state-var-1 = initial-value-1-or-pars[0]; ...
    require(precondition);
    for (int i = 0; i < actions.length; i++) {
      switch (actions[i]) {
        case 1:
          require(condition-for-action-1);
          computation-of-next-state
          break;
        case 2:
        ...
        default:
          require(false);
      }
    }
    if (condition-for-unsafe-state-1) return false; ...
    return true;
  }
  void require(boolean b) throws ModelException {
    if (!b) throw new ModelException();
  }
}
```

Fig. 1. Model encoding pattern

As an example, the shortest model from the collection in [16]—the model of MOESI cache-coherence protocol—is shown in Figures 2 and 3. It took less than 1 second on Pentium 4 1.8 GHz to supercompile the example. Other protocol models in [16] are supercompiled in a few seconds as well.

Since the residual code produced by JScp is too large, only its residual graph (the flow-chart of the residual program) is shown in Fig. 3. The essential information contains in the `return` statements that are drawn as double octagons with labels `T` or `F` meaning `return true` and `return false` respectively. Since there are no `F`-nodes in this graph, we conclude the MOESI protocol model is correct.

The graph in Fig. 3 is larger than the residual graph of this model in Refal produced by the Refal supercompiler SCP4 as it is shown in [25]. The reason is SCP4 always produces minimal graphs, while the configuration analysis in the current version of JScp is not perfect and often returns non-minimal graphs. In SCP4 a current configuration is compared with all configurations produced so far, while JScp now uses for comparison only the configurations lying on the path from the initial configuration of current loop statement. This drawback is planned to be fixed in future versions of JScp. It does not influence the class

```
public boolean runModel(int[] actions, int[] pars)
  throws ModelException
{
  // set and check initial state (precondition)
  int invalid   = pars[0], invalid_   = invalid;
  int exclusive = 0,       exclusive_ = exclusive;
  int shared    = 0,       shared_    = shared;
  int modified  = 0,       modified_  = modified;
  int owned     = 0,       owned_     = owned;
  require (invalid >= 1);

  // execute actions
  for (int i = 0; i < actions.length; i++) {
    switch (actions[i]) {
      ... ==========================>
      default:
        require (false);
    }
    invalid   = invalid_;
    exclusive = exclusive_;
    shared    = shared_;
    modified  = modified_;
    owned     = owned_;
  }

  // check safety conditions (postcondition)
  if (exclusive + shared + owned >= 1 &&
      modified >= 1)
      return false;
  if (exclusive >= 1 && shared + owned >= 1)
      return false;
  if (modified >= 2) return false;
  if (exclusive >= 2) return false;
  return true;
}
```

```
case 1; // rm
  require (invalid >= 1);
  invalid_   = invalid - 1;
  exclusive_ = 0;
  modified_  = 0;
  shared_    = shared +
               exclusive + 1;
  owned_     = owned + modified;
  break;
case 2: // wh2
  require (exclusive >= 1);
  exclusive_ = exclusive - 1;
  modified_  = modified + 1;
  break;
case 3: // wh3
  require (shared + owned >= 1);
  shared_    = 0;
  exclusive_ = 1;
  modified_  = 0;
  owned_     = 0;
  invalid_   = invalid + modified +
               exclusive + shared +
               owned - 1;
  break;
case 4: // wm
  require (invalid >= 1);
  shared_    = 0;
  exclusive_ = 1;
  modified_  = 0;
  owned_     = 0;
  invalid_   = invalid + modified +
               exclusive + shared +
               owned - 1;
  break;
```

Fig. 2. MOESI cache-coherence protocol model source code in Java

of verification tasks solved by the supercompiler, but only the supercompilation time and the residual graph size.

It is instructive to note a residual code and its graph is actually the graph of an induction proof of the correctness of a protocol. The residual Java code could be mechanically translated into mathematical notation.

A supercompiler is also capable of finding counterexamples in some cases where a model is incorrect. The site [16] contains such an example: an incorrect version of Xerox PARC Dragon cache-coherence protocol published in [7].

Of course, a supercompiler is not capable of verifying all protocol models and all programs in general. When it fails, the residual graph contains `return false` statements. In this case, the user should perform several supercompilation experiments varying JScp options and modifying the source code.

Legend

- Diamond:
  `if` statement.
- Hexagon:
  head of loop statement.
- Double octagon:
  `return` statement.
  Label `T` means `return true`.
  There are no `return false`
  statements here.
- Arcs:
  basic blocks (optionally
  ending with `break` or
  `continue`).
  Arcs with `throw` statements
  are not shown.

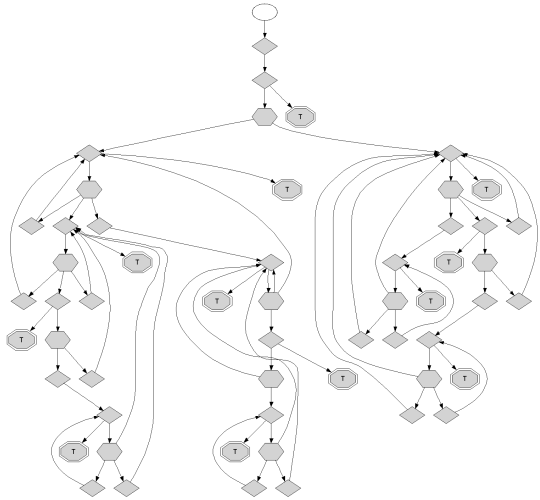

Fig. 3. MOESI cache-coherence protocol model residual graph. The absence of nodes `return false` (with label `F`) shows JScp has proven the protocol model is correct.

4 Conclusion and Related Work

As compared to specific verification methods and modeling languages used by G. Delzanno and others, we have shown that a universal program specialization method for a universal programming language Java implemented in the Java Supercompiler JScp is capable of verification of a large class of protocol models. The part of the supercompilation method sufficient for solving this problem was described in this paper. It is simpler than model checking methods of program verification based on temporary logics [9], although model checking can verify more statements about programs than the considered part of supercompilation.

The Java supercompilation method is an extension of supercompilation for functional languages [30, 33, 34, 27]. The extension concerns specialization of operations on objects (described in another paper [15]) and a new method of configuration analysis of control statements reviewed in this paper.

Supercompilation of the imperative subset of the Java language is worth comparing with works on mixed computation [11, 4] and partial evaluation of imperative [14] and object-oriented languages [29, 5, 19]. The main distinctive feature of supercompilation is the explicit notion of a configuration with configuration variables and operations on configurations. This allows for more sophisticated analysis and transformation of programs, including induction hypothesis generation by methods of generalization of configurations as well as termination based of the homeomorphic embedding relation. These methods are essential for program verification. Mixed computation and partial evaluation lacks the notions of a configuration variable and a configuration and, from our viewpoint, this is their main limitation.

Another often mentioned difference of supercompilation from partial evaluation is absence of a preliminary (binding time) analysis. Supercompilation as an "on-line" technique is capable of performing deeper specialization than "off-line" partial evaluation. This is also crucial for application of program specialization methods to program verification.

5 Acknowledgements

The development of the Java supercompiler would not be possible without collaboration with many people. The project was started together with Larry Witte and Valentin Turchin to whom the author is greatly indebted. Special thanks are due to the developers of various parts of the JScp system Arkady Klimov and Artem Shvorin. We are very grateful to our partners Ben Goertzel and Yuri Mostovoy: without their help and support such a complex project as JScp could not be done.

It was a pleasure to collaborate with Andrei Nemytykh on application of supercompilation to program verification.

The ideas and results of Java supercompilation and program verification were discussed at the seminar on Refal and metacomputation at Keldysh Institute. Special thanks to its most active participants: Sergei Abramov, Arkady Klimov, Yuri Klimov, Ilya Klyuchnikov, Andrei Nemytykh, Leonid Provorov, Igor Shchenkov, Sergei Romanenko, Anton Orlov, and others.

References

1. Reynald Affeldt, Hidehiko Masuhara, Eijiro Sumii, and Akinori Yonezawa. Supporting objects in run-time bytecode specialization. In *ASIA-PEPM*, pages 50–60, 2002.
2. Peter Bertelsen. *Binding-time analysis for a JVM core language*, 1999. Unpublished note; available from http://www.petermb.dk/bta-core-jvm.ps.gz.
3. 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*. Springer, 2003.
4. Mikhail A. Bulyonkov. From partial evaluation to mixed computation. *Theor. Comput. Sci.*, 90(1):47–60, 1991.
5. Andrei M. Chepovsky, Andrei V. Klimov, Arkady V. Klimov, Yuri A. Klimov, Andrei S. Mishchenko, Sergei A. Romanenko, and Sergei Yu. Skorobogatov. Partial evaluation for common intermediate language. In Broy and Zamulin [3], pages 171–177.
6. Giorgio Delzanno. *Automatic Verification of Cache Coherence Protocols via Infinite-state Constraint-based Model Checking.* Web site http://www.disi.unige.it/person/DelzannoG/protocol.html.
7. Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2000.

8. Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
9. Orna Grumberg Edmund M. Clarke Jr. and and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
10. Andrei P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
11. Andrei P. Ershov and V. E. Itkin. Correctness of mixed computation in Algol-like programs. In Jozef Gruska, editor, *MFCS*, volume 53 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 1977.
12. Yoshihiko Futamura. Partial computation of programs. In Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa, editors, *RIMS Symposium on Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 1983.
13. Ben Goertzel, Andrei V. Klimov, and Arkady V. Klimov. *Supercompiling Java Programs, white paper*, 2002. http://www.supercompilers.com/white_paper.shtml.
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
15. Andrei V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In Nemytykh [28], pages 43–53. http://meta2008.pereslavl.ru/accepted-papers/paper-info-4.html.
16. Andrei V. Klimov. *JVer Project: Verification of Java programs by Java Supercompiler*, 2008. Web site http://pat.keldysh.ru/jver/.
17. Andrei V. Klimov, Arkady V. Klimov, and Artem B. Shvorin. *The Java Supercompiler Project*. Web site http://www.supercompilers.ru.
18. Yuri A. Klimov. An approach to polyvariant binding time analysis for a stack-based language. In Nemytykh [28], pages 78–84. http://meta2008.pereslavl.ru/accepted-papers/paper-info-6.html.
19. Yuri A. Klimov. Program specialization for object-oriented languages by partial evaluation: approaches and problems. Preprint 28, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2008. (In Russian).
20. Joseph B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.
21. Alexei P. Lisitsa and Andrei P. Nemytykh. Towards verification via supercompilation. In *COMPSAC (2)*, pages 9–10. IEEE Computer Society, 2005.
22. Alexei P. Lisitsa and Andrei P. Nemytykh. *Experiments on verification via supercompilation*, 2007. Web site http://refal.botik.ru/protocols/.
23. Alexei P. Lisitsa and Andrei P. Nemytykh. A note on specialization of interpreters. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *CSR*, volume 4649 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2007.
24. 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.
25. Alexei P. Lisitsa and Andrei P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.
26. Alexei P. Lisitsa and Andrei P. Nemytykh. Verification as specialization of interpreters with respect to data. In Nemytykh [28], pages 94–112. http://meta2008.pereslavl.ru/accepted-papers/paper-info-8.html.
27. Andrei P. Nemytykh. The supercompiler SCP4: General structure. In Broy and Zamulin [3], pages 162–170.

28. Andrei P. Nemytykh, editor. *The First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2-5, 2008*. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008.
29. Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.
30. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *International Logic Programming Symposium, December 4-7, 1995, Portland, Oregon*, pages 465–479. MIT Press, 1995.
31. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
32. Valentin F. Turchin. The algorithm of generalization in the supercompiler. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
33. Valentin F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 481–509. Springer, 1996.
34. Valentin F. Turchin. Supercompilation: techniques and results. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996, Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 227–248. Springer, 1996.