Verification as Specialization of Interpreters with Respect to Data

Alexei P. Lisitsa[1] and Andrei P. Nemytykh[2]*

[1] Department of Computer Science, The University of Liverpool
`A.Lisitsa@csc.liv.ac.uk`
[2] Program Systems Institute of Russian Academy of Sciences
`nemytykh@math.botik.ru`

Abstract. In the paper we explain the technique of *verification via supercompliation* taking as an example verification of the parameterised Load Balancing Monitor system. We demonstrate detailed executable specification of the Load Balancing Monitor protocol in a functional programming language REFAL and discuss the result of its supercompilation by the supercompiler SCP4.
This case study is interesting both from the point of view of verification and program specialization. From the point of view of verification, a new type of non-determinism is involved in the protocol, which has not been covered yet in previous applications of the technique. With regard to program specialization, we argued earlier that our approach to program verification may be seen as specialization of interpreters with respect to data [25]. We showed that by supercompilation of an interpreter of a simplest purely imperative programming language. The language corresponding to the Load Balancing Monitor protocol that we consider here has some features both of imperative and functional languages.

Keywords: Program specialization, supercompilation, program verification, broadcast protocols.

1 Introduction

Valentin Turchin in his classical paper on supercompilation [41] has suggested the following scheme of using this program transformation technique for proving properties of the (functional) programs:

> ... if we want to check that the output of a function $F(x)$ always has the property $P(x)$, we can try to transform the function $P(F(x))$ into an identical T.

* The second author is supported by Russian Foundation for Basic Research (grants 07-07-92100-GFEN_a, 08-07-00280-a), Program for Basic Research of Presidium of Russian Academy of Sciences (as a part of "Development of the basis of scientific distributed informational-computing environment on the base of GRID technologies") and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

The scheme albeit being very natural has not attracted much attention and has not been used until recently for proving properties of programs. In [19,21,22,20] we revitalized the idea and proposed the particular scheme of

parameterized testing + supercompilation

based on Turchin's proposal, suitable for verification of parameterized protocols. Various protocols have been verified using the scheme [19,21,22,23,24,25]. In this paper we explain the technique by presenting the verification of yet another protocol that is a Load Balancing Monitor [2].

This case study is interesting both from the point of view of verification and program specialization.

From the point of view of verification, a new type of non-determinism is involved in the protocol, which has not been covered yet in previous applications of the technique.

With regard to program specialization we argued in [25] that our approach to program verification may be seen as specialization of interpreters with respect to data; we gave an example of the task (successfully resolved by the supercompiler SCP4), where the language to be interpreted was a purely imperative programming language L. Here under the language we mean a simplest programming language L corresponding to a parameterized cache coherence protocol. A prorgam in the language L is a *finite* sequence of instructions corresponding to the actions of the protocol. Any instruction when executed updates the global state of the computing system controlled by the protocol.

The LBM protocol provides an example of a simplest language with functional features: the programs not only transform the global memory, but also the data passed through the arguments. Taking into account our choice of a program model of the protocol, the "brute force" algorithm involved in supercompilation and traversing all programs together with their arguments makes possible the analysis of non-deterministic choices of a next action (and/or a values of an argument).

Despite simplicity of the languages generated by the protocols (and, what is more, due to their algorithmic incompleteness), automatic specialization of their interpreters opens very interesting and important problems leading to more fundamental understanding the nature of program specialization. Indeed, algorithmic completeness of any language to be interpreted makes any interesting specialization task (*per se*) of its interpreter algorithmic undecidable, while algorithmic incompleteness of a language provides reasonable hope that statements on properties of the tools specializing such interpreters[3] may be formulated and proved. For example, the classical specialization task considered in the following section (specialization of an interpreter `int` with respect to a given program) aims to eliminate the whole interpretive overhead. But such a task (*per se*) is undecidable, when the language to be interpreted is algorithmically complete; just because in such a case the `int` has to be written itself in an algorithmically complete programming language. An incomplete programming language L to

[3] The properties concerning the specialization task to be solved.

be interpreted provides a hope that an interpreter `int` simulating the programs written in L may be implemented itself in another incomplete programming language. The last in its turn gives a hope that the problem to eliminate the whole overhead may be decidable (not only `per se` but by means of a concrete specializer). Especially that is very interesting when such an incomplete language L (to be interpreted) originates from practice. That is our case.

In practical implementation of the technique we use the functional programming language REFAL [43] and the most advanced REFAL supercompiler SCP4 [31,34]

2 Verification as Specialization of Interpreters

Given two programming languages `L, M` and the semantics of `L` described by an interpreter `int(p,d)` written in `M`, where the first argument stands for the source `L`-programs and the second ranges over the data of the `L` language. There is a famous task for automated specialization of the interpreter with respect to the first argument `int(p`$_0$`,d)`, i.e. the program `p`$_0$ is known while the data `d` is unknown. Specialization has to generate a residual program `q` such that `q(d) = int(p`$_0$`,d)`, where the equality holds whenever the pair `(p`$_0$`,d)` belongs to the domain of the interpreter. Certainly the `q` is written in `M`: consequently `q` can be seen as a result of compilation of `p`$_0$ from `L` to `M`. The goal is to construct an *optimal* `q`. The formulated problem is both undecidable (of course) and interesting. A lot of work was devoted to approximation of the problem (see [5,6,8,9,29,35] for examples).

In the paper [25] we showed that specialization of interpreters with respect to data may be reasonable and leads to interesting applications in verification. We considered the following specialization problem `int(p,d,d`$_0$`)`, where the known part of the data is separated from the unknown part. Firstly, for the sake of simplicity, let us think of the languages `L,M` as relational languages, i.e. the languages defining only (partial) predicates rather than arbitrary recursive functions. Another assumption is that the interpreter `int` terminates for all possible values of its arguments, but for some values it may terminate with abnormal stop. The abnormal stop indicates that the input values of the arguments are outside of the domain. Let us have a robust specializer generating a residual program `q` defining an extension of the partial predicate defined by the problem `int(p,d,d`$_0$`)`. Assume that `q` is a partial constant function `TRUE` or `FALSE` and this property is expressed explicitly in syntax of `q`. For example, `q` does not contain any syntactic construction with the semantics `return FALSE` (in the case of the `TRUE` partial constant). Thus, we assume that specializer was weak enough not to be able to optimize the predicate `int` as

```
q(p,d) { return TRUE; }
```

but was strong enough to eliminate all syntactic constructors of the form `return FALSE;`. In such a case, the result of specialization can be considered as a proof of the (partial) constant property. The termination property of `int` mentioned above guarantees that the domain of the original partial predicate is not empty. Notice that we assume that the specializer is allowed to extend the domain of the original partial predicate. This provides additional important possibilities for specialization (see [30]) and distinguishes supercompilation, the technology of specialization we use (see Section 5), from other well known specialization technologies (e.g. partial evaluation [8]).

Consider now a more complicated interpreter. Let `int` be a composition $\varphi \circ$`fint` of a functional language interpreter `fint` (i.e. not only predicative) and a predicate-postcondition φ testing the result of `fint`-interpretation. Now the `TRUE`-constant property of the residual program `q` means all source programs `p` satisfy the post-condition φ (in the given context of specialization). In such a case we conclude that the specializer solved a verification problem. The composition $\varphi \circ$ `fint` can be encoded in various ways.

The following sections are devoted to a non-trivial application of the idea.

3 Parameterized Testing and Verification

In this section we describe our general technique for the verification of parameterized systems. The technique is based on the translation of the statements about *safety* properties of a system to be verified into the statements about properties of the program that *simulates and tests* the system. The reader is called to trace parallels with the previous Section 2.

The scheme works as follows. Let S be a parameterized system (a protocol) and we would like to establish some safety property Q of S. We write a program $\texttt{fint}_S$ simulating execution of S for n steps, where n is an input parameter. If the system is non-deterministic, an additional parameter $\bar{p}$ is provided, whose value is assumed to be a sequence of choices at the branching points of execution, e.g. it may be a string of characters labeling the choices. Thus, we assume that given the values of input parameters n and $\bar{p}$, the program $\texttt{fint}_S$ returns the state of the system S after the execution of n steps of the system, following the choices provided by the value of $\bar{p}$. Let $T_Q(\_)$ be a testing program, which given a state s of S returns the result of testing the property Q on s (`TRUE` or `FALSE`). Consider a composition $T_Q(\texttt{fint}_S(n, \bar{p}))$. This program first simulates the execution of the system and then tests the property required. Now the statement

"the safety property Q holds in any possible state reachable by the execution of the system S"

is equivalent to the statement

"the program $T(\texttt{fint}_Q(n, \bar{x}))$ never returns the value `FALSE`, no matter what values are given to the input parameters".

Here we assume additionally that both programs $\texttt{fint}_Q$ and T terminate for all possible inputs, but for some values they may terminate with abnormal stop.

In practical implementation of the scheme we use functional programming language REFAL-5 to implement a program $T_Q(\mathtt{fint}_S(n,\bar{p}))$ and the supercompiler SCP4 to transform a program to a form, from which one can easily establish the required property.

In present paper we extend this basic technique to tackle protocols with new type of non-determinism. To this end choices at the branching points of execution of protocols are labeled not by characters but rather by terms. Further, there is not need for two separate parameters n and $\bar{p}$ – the length of (the value of) $\bar{p}$ will play the role of n.

4 REFAL Programming Language

The REFAL programming language [43] (Recursive Functions Algorithmic Language) is a first-order strict functional language. Unlike LISP the language is based on the model of computation known as Markov's algorithms [28]. Here we restrict ourselves with a fragment of REFAL and everywhere we will mean the fragment.

```
program    ::= $ENTRY definition+
definition ::= function-name { sentence;+ }
sentence   ::= left-side = expression
left-side  ::= pattern
expression ::= empty | term expression | function-call expression
function-call ::= <function-name arg>
arg ::= expression
pattern  ::= empty | term pattern
term     ::= SYMBOL | variable | (expression)
variable ::= e.variable-name | s.variable-name | t.variable-name
empty    ::=                      /* nihil */
```

REFAL data are defined by the grammar:

$$\mathtt{d ::= d_1\ d_2\ |\ (d_1)\ |\ SYMBOL\ |\ empty}$$

Roughly speaking, a program in REFAL is a term rewriting system. The semantics of the language is based on pattern matching. As usual, the rewriting rules are ordered to match from the top to the bottom. The terms are generated using two constructors. The first is concatenation. It is binary, *associative* and is used in infix notation, which allows us to drop its parentheses. The blank is used to denote concatenation. The second constructor is unary. It is syntactically denoted by its parentheses only (that is without a name). The unary constructor is used for constructing tree structures. Formally, every function is unary. The empty sequence is a special basic ground term. This term is denoted with nothing and called "empty expression". It is the neutral element (both left and right) of concatenation. All other basic ground terms are named as "symbols". That is unlike the LISP data set including only binary trees (i.e. not arbitrary trees and not sequences of trees).

There exist three types of variables – `e.name`, `s.name` and `t.name`. An `e`-variable can take any expression as its value, an `s`-variable can take any symbol as its value and `t`-variable can take any term as its value (a term is either a symbol or an expression in structure brackets). For every sentence its set of variables from the left side includes its set of variables from the right side; there are no other restrictions on the variables. Associativity of the concatenation may cause abstract pattern matching to be ambiguous on some patterns[4]. In the context of this paper, it is not important how the ambiguousness is actually resolved. It is sufficient to assume that the pattern matching is done deterministically.

Let a current active function call be given. A step of the REFAL machine is the following sequence of actions: pattern matching, replacement of the right side variables with their values – with the result of the pattern matching, replacement of the active function call (in the function stack) with the updated right side and labeling of a new function call on the top of the changed stack as active.

Example: The following program replaces every occurrence of the identifier `LISP` with the identifier `REFAL` in an arbitrary REFAL datum.

```
$ENTRY Go { e.inp = <Repl (LISP REFAL) e.inp>; }
Repl {
(s.x e.v) = ;
(s.x e.v) s.x e.inp = e.v <Repl (s.x e.v) e.inp>;
(s.x e.v) s.y e.inp = s.y <Repl (s.x e.v) e.inp>;
(s.x e.v) (e.y) e.inp = (<Repl (s.x e.v) e.y>)
                         <Repl (s.x e.v) e.inp>;
}
```

On the right side of the first sentence of `Repl` we see the empty expression. The left sides of the last three sentences and the right side of the second sentence of `Repl` show associativity of the concatenation.

Consider a trace of a REFAL computation for the program given above. Let the computation start with the call `<Go (A LISP)>`. The REFAL datum `(A LISP)` represents a binary tree with the leaves `A`, `LISP`. The computation proceeds with the following steps:

```
2: <Repl (Lisp REFAL)  (A LISP)>
3: (<Repl (Lisp REFAL) A LISP>) <Repl (LISP REFAL)>
4: (A <Repl (Lisp REFAL)  LISP>) <Repl (LISP REFAL)>
```

[4] For example, the following equation `e.1 e.2 = A B` has three solutions: 1) `e.1 = [], e.2 = A B`; 2) `e.1 = A, e.2 = B`; 3) `e.1 = A B, e.2 = []`; Here `[]` stands for the empty expression. In such cases the real REFAL pattern matching takes the solution with minimal length of the datum taken by the first e-variable (from the left to the right) and so on by induction (see [43] for the details). In our case the first solution `e.1 = [], e.2 = A B` will be chosen.

```
5: (A REFAL <Repl (LISP REFAL)>) <Repl (LISP REFAL)>
6: (A REFAL) <Repl (LISP REFAL)>
7: (A REFAL)
```

Another example is the function `append`, which can be defined in REFAL in one line:

```
append { (e.xs) (e.ys) = e.xs e.ys; }
```

The LISP style `append`-function is defined as follows:

```
LispAppend {
 ()         (e.ys) = e.ys;
 (t.x e.xs) (e.ys) = t.x <LispAppend (e.xs) (e.ys)>;
}
```

A detailed description of the language is available in an electronic format [43] (see also [30]).

5 Supercompiler SCP4

In this section we present a short introduction to supercompilation process, as it is implemented in the supercompiler SCP4. More details can be found in [31,34,32,33].

Consider a program written in some programming language together with a parameterized input entry of the program. Such a pair defines a partial input-output mapping `f:` $D \mapsto D$, where D is the data set of the language. By definition, a supercompiler is a transformer of such pairs.

The supercompiler SCP4 iterates an *extension* of the interpretation of REFAL steps (see Section 4), called *driving* [41], on parameterized sets of the input entries. Driving constructs a directed tree of all possible computations for the given parameterized input entry and a given REFAL step. The edges of this tree are labeled with predicates over values of the parameters. The predicates specify concrete computation branches and describe the narrowing of the parameters (unknown data) along the chosen branches[5].

Iteration of the driving unfolds a potentially infinite tree of all possible computations. The computations can depend on the values of the parameters that can be unknown during transformation. The supercompiler reduces in the process the redundancy that could be present in the original program. It folds the tree into a finite graph of states and transformations between possible configurations of the computing system. To make a folding possible a *generalization* procedure is used. Sometimes it may lead to the loss of some information on the structure of arguments of configurations.

[5] In this sense the driving works similarly to a PROLOG interpreter. Both tools accept parameters (free variables) as their input data and narrow the parameters.

If it is not possible to reduce a current configuration (to be developed in the meta-tree) to a previous configuration (on the path from the tree root to the latter) then generalization looks for a previous configuration, which is *similar* to the current. A *homeomorphic embedding pre-order* specifies the *similarity* relation on the configurations [16,38,31]. Only similar configurations are generalized. We say the term `Current` is not less complex compared to the `Previous` iff the `Previous` can be homeomorphically embedded to the `Current`. If the set of the basic terms is reasonable enough (see [11,15,16] for the details), then any infinite term sequence $\mathtt{t}_n$ has a pair $\mathtt{t}_i$, $\mathtt{t}_k$ such that $k > i$ and $\mathtt{t}_k$ is not less complex compared to $\mathtt{t}_i$. The property is crucial to ensure termination of SCP4, if *all* configurations appearing in the meta-tree are analyzed by generalization (in a *weak* strategy of supercompilation).

The aim of specialization is to perform as many actions of the input program at supercompile-time as possible. The parameterized configurations corresponding to the meta-tree nodes originating single branches can be one-step-developed uniformly on values of the parameters.

Thus, we emphasize that the output of the supercompiler is defined in terms of the parameters (*semantic objects*). The resulting definition is constructed solely based on the meta-interpretation of the source program rather than by a step-by-step transformation of the program. The crucial property of the supercompilation procedure, that we rely upon in our verification methodology, is

Property 1. The output pair (the residual program and its input entry) defines an extension of the partial mapping defined by the corresponding input pair.

6 Load Balancing Monitor Protocol

As a case study we consider in this section verification of a multiprocess system with a load balancing monitor. The Figure 1 (from [2]) shows an abstraction of such a system, two different finite automata, one is for the *monitor* another for a process. In general, we are interested in parameterized systems, consisting of an arbitrary number m of processes (here m is a parameter) and a single monitor. In the initial configuration of the system the processes are in state `req`, and the monitor is in state the `idle`. When the monitor broadcasts the message $\mathtt{swap}_{out}$ (and moves to busy) all processes in the CPU are suspended. Two different priorities, `high` and `low` are assigned non-deterministically to the suspended processes. When the CPU is released by the monitor (through the broadcast release), it is assigned to processes with high priority. Processes with low-priority go back to the `request` state.

We use a specification of such a parameterized system given in [2] in terms of Extended Finite State Machines (EFSM)[1]:

```
(0) req ≥ 0, use ≥ 0, idle ≥ 0, busy ≥ 0, high ≥ 0, low ≥ 0 → .
(1) req ≥ 1, idle ≥ 1 → req' = req - 1, use' = use + 1.
(2) use ≥ 1, idle ≥ 1 → req' = req + 1, use' = use - 1.
(3) idle ≥ 1 → idle' = idle - 1, busy' = busy + 1,
               high' + low' = high + low + use,
               high' ≥ high, use' = 0.
(4) busy ≥ 1 → busy' = busy - 1, idle' = idle + 1,
               high' = 0, low' = 0,
               use' = use + high, req' = req + low.
```

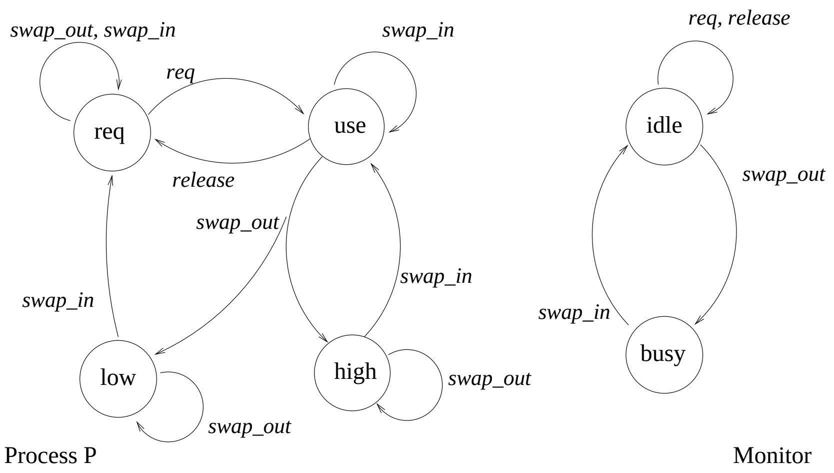

Fig. 1. Load balancing monitor

Here `req,use,idle,busy,high,low` are non-negative integer variables of the EFSM model, which represent *counting abstraction* of the original parameterized automata model: the names denote various states of the automaton and the values of the variables keep track of the number of automata in corresponding states. The rules `(0)-(4)` define the dynamics of the EFSM model. Starting with some initial evaluation of the variables, the system may apply non-deterministically any of the rules. In the case the guard of a rule (its left-hand part) is satisfied in a current state (evaluation of all variables, i.e. the integer vector), the update expressed by the right-hand side of the rule is executed. Primed variable names are used in updates to denote updated values. Updates may be deterministic, like `use' = use + 1`, or non-deterministic, like

`high' + low' = high + low + use`.

In the latter case execution of an update assigns the values to the variables non-deterministically, provided they satisfy all constraints of the right-hand side of the rule. For example in the rule `(3)` an additional constraint is `high' ≥ high`.

Correctness of the Load Balancing Monitor Protocol specified by the above EFSM is formulated as follows: the system if started in the initial configuration where all processes are in state `req` and the monitor is in a state `idle` never reaches a configuration where there coexists processes in states `busy` and `use`.

7 Load Balancing Monitor Specification in REFAL

In this section we apply *the parameterized testing + supercompilation* approach described above to the verification of the Load Balancing Monitor. To do so we need to write down a REFAL program, which first simulates the execution of the protocol (in fact, it may be considered to be an executable specification) and then tests the correctness condition. The following fragment of the program defines the function `int` which is the entry point of the program:

```
$ENTRY int {
 e.p (e.d) =
        <fint (e.p) (idle I)(busy )(req e.d)(use )(high )(low )>;
}
```

It has two input parameters `e.p` and `e.d` which according to the REFAL conventions may take arbitrary REFAL expressions as the values. The function `int` is defined though only for the inputs of special form. For `e.p` the only values in the range of `int` are of the form $(t_{i_1})\ldots(t_{i_k})$, where t_{i_j} are expressions labeling different rules of the above EFSM model. With the exception of the rule `(3)` these are just names, whilst for the rule `(3)` this is a name with an additional parameter (see definition of the `RandomAction` function below). The value of the variable `e.d` is assumed to be a string of characters each representing a process in the model. So the length of a string (value of) `e.d` is a number of processes in the modeled system. In general, we use the following representation of the global state of the system by REFAL data:

```
  (e.p) (idle e.1)(busy e.2)(req e.3)(use e.4)(high e.5)(low e.6)
```

where `e.p` represents the sequence of remaining rules to be executed, and the length of the value of each string `e.`i represents the value of the corresponding variable (e.g. the length of the value of `e.4` is a value of `use` at any given moment).

Returning to the definition of the function `int`, it takes two input parameters and calls the function `fint` (reminder: angular brackets denote a function call in REFAL). The syntactical form of arguments for this call of `fint` reflects constraints on the initial configuration of EFSM – the (single) monitor is in `idle` state, some processes are in the state `req` and *no processes* are in any other state.

The definition of the function `fint` contains two sentences: one for quitting the loop and passing to the correctness testing (first sentence)[6] and another for

[6] Such a kind of the encoding of the *composition* is crucial for successful automatic verification of the protocol.

making a recursive call of `fint` with the decremented first argument value and the current state updated depending on the term `t.t`. Update is done by the call to the `RandomAction` function.

```
fint {
  () (idle e.1)(busy e.2)(req e.3)(use e.4)(high e.5)(low e.6) =
         <Test (idle e.1)(busy e.2)(req e.3)
                (use e.4)(high e.5)(low e.6)>;
  (t.t e.p) (idle e.1)(busy e.2)(req e.3)
                      (use e.4)(high e.5)(low e.6) =
         <fint (e.p) <RandomAction t.t (idle e.1)(busy e.2)
                         (req e.3)(use e.4)(high e.5)(low e.6)>>;
}
```

The sentences in the definition of the `RandomAction` function[7] correspond to the rules of the EFSM model. Since the rule `(0)` does not do anything (the system is waiting with no changes in its global state) and we are going to verify a safety property, we can safely omit this rule from the REFAL specification[8]

```
RandomAction {
* r1
(r1) (idle I e.1)(busy e.2)(req I e.3)(use e.4)(high e.5)(low e.6)
                     = (idle I e.1)(busy e.2)(req e.3)(use I e.4)
                                              (high e.5)(low e.6);
* r2
(r2) (idle I e.1)(busy e.2)(req e.3)(use I e.4)(high e.5)(low e.6)
                     = (idle I e.1)(busy e.2)(req I e.3)(use e.4)
                                              (high e.5)(low e.6);
* r3
(r3  e.r3) (idle I e.1)(busy e.2)(req e.3)(use e.4)
                                 (high e.5)(low e.6) =
           (idle e.1)(busy I e.2)
          <RandomDistribution (r3  (low_use e.4 e.6) e.r3)
                              (req e.3)(use )(high e.5)(low e.6)>;
* r4
(r4) (idle e.1)(busy I e.2)(req e.3)(use e.4)(high e.5)(low e.6)
              = (idle I e.1)(busy e.2)(req e.3 e.6)(use e.4 e.5)
                                                   (high )(low );
}
```

As an example consider the case when `RandomAction` is called with arguments matching the left-hand side of the first rule. Then the call will return the (representation of) global state expected after application of the rule `(1)`.

[7] Here the asterisk sign stands for one line comment.

[8] Leaving this rule in place and providing suitable REFAL translation would not do any difference in the verification.

Further, it is straightforward to check that the left-hand sides (resp. right-hand sides) of sentences commented as `*r1, *r2, *r4` implement guards (resp. deterministic updates) of the corresponding EFSM rules `(1), (2), (4)`. The rule (3) is more involved because of its non-deterministic update. Its implementation by the sentence `*r3` of `RandomAction` definition uses an additional call to function `RandomDistribution`, the definition of which is as follows:

```
RandomDistribution {
(r3  (low_use I e.lu) (high_low I e.hl))
                          (req e.3)(use e.4)(high e.5)(low e.6)
       = <RandomDistribution (r3  (low_use e.lu)(high_low e.hl))
                         (req e.3)(use e.4)(high I e.5)(low e.6)>;

(r3  (low_use e.lu) (high_low ))
                          (req e.3)(use e.4)(high e.5)(low e.6)
       = (req e.3)(use e.4)(high e.5)(low e.lu e.6);

(r3  (low_use ) (high_low e.hl))
                          (req e.3)(use e.4)(high e.5)(low e.6)
       = (req e.3)(use e.4)(high e.5)(low e.6);
}
```

The function `RandomDisribution` implements the non-deterministic update `high'+low' = high+low+use` with the constraint `high'` $\geq$ `high`. The latter condition may be reformulated as `high' = high + delta` for some non-negative integer value `delta`. Then we have `low' = low + use - delta`. The last two conditions can be considered as almost deterministic updates, where the only non-determinism remaining is concerning the value of `delta`. Consider the definition of `RandomDistribution`. Following the same convention for representing integer variables by REFAL data it introduces two auxiliary integer variables[9] `low_use` and `high_low`. When `RandomDistribution` is called within the `*r3` sentence of `RandomAction` the variable `low_use` gets the value `use+low` (in REFAL terms on the left-hand side of the `*r3` we have `(use e.4)` and `(low e.6)` and on the right-hand side `(low_use e.4 e.6)`). As to the `high_low` variable its value represents the above `delta`. Where does it come from? Inspection of definitions of all functions defined so far shows that the value of `high_low` is passed via parameters `e.r3` of `RandomAction`, `t.t` of `fint` and `e.p` of `int`.

Consider now computation of `RandomDistribution` in terms of integer variables. It starts[10] with `low_use = use + low`. If `low_use` ≥ 1 and `high_low` ≥ 1, both `low_use` and `high_low` are decremented by 1, `high` is incremented by 1 and `RandomDistribution` is recursively called (the first sentence of definition).

If `high_low` hits 0 then the call returns with `low' = use + low - high_low` and `high' = high + high_low` as required (the second sentence).

[9] Here we mean integer variables in a sense of the EFSM model, not REFAL variables. Integer variables `low_use` and `high_low` are presented by REFAL terms (`low_use` ...) and (`high_low` ...).

[10] When called from the `*r3`.

If, however, `low_use` hits 0 first (the third sentence) that indicates the value of `high_low` is incorrect (`high_low` $>$ `low+use`), but the call returns the still correct update: `low' = 0` and `high' = high+low+use`.

Correctness condition. Finally, the definition of the function `Test` embodies the correctness conditions of the protocol (compare with the definition of safety properties in Section 6).

```
Test {
(idle e.1)(busy I e.2)(req e.3)(use I e.4)
                      (high e.5)(low e.6) = FALSE;

(idle e.1)(busy e.2)(req e.3)(use e.4)
                    (high e.5)(low e.6) = TRUE;
}
```

The function `Test` returns `FALSE` if called on a configuration with `busy` ≥ 1 and `idle` ≥ 1, in all other cases it returns `TRUE`.

Taking all above definitions together we get a REFAL program

`int(e.p, e.d)`

which simulates execution of the Load Balancing Monitor system with one monitor and k (= the length of the value of `e.d`) processes for n steps (= the length of sequence $(t_{i_1}) \dots (t_{i_n})$, the value of `e.p`) and then tests the correctness condition.

8 Verification via Supercompilation

Now we apply the supercompiler SCP4 to the program `int(e.p, e.d)` (with the parameterized entry point). That is to say, we specialize the interpreter `fint` with respect to partial known data, while the program `e.p` is unknown:

```
<fint (e.p) (idle I)(busy )(req e.d)(use )(high )(low )>
```

Here we treat the `fint`-interpreter as an interpreter of the programming language L defined by the protocol rules: each program is a *finite* sequence of the actions evaluating the protocol. Notice that not all actions are applicable to all protocol states; an attempt of execution of a non-applicable action leads to an abnormal stop of the program. The simplest language L has a functional aspect: calls for the action `r3` have arguments `e.r3`. I.e. this action may be considered as a function transforming both the global (the global protocol state) and local (the value of the `e.r3`) memory.

The residual program is given in the Appendix A. At first glance the resulting program does not look much simpler than the original one and definitely it is much less comprehensible. However, it is now simple to check the entire resulting program unlike the original one does not contain the operator `return FALSE`.

That means whatever values input parameters are given, the program will never return FALSE. Since the resulting program is equivalent to the original one on the domain of the original program and the original program is never looping forever, we conclude that original program will also never return FALSE. That implies the correctness of the encoded parameterized protocol.

8.1 On Associativity of the Concatenation

An interesting question arises here: how does the associative property of the REFAL concatenation matter in the successful verification of the LBM protocol? And a more general question is: how does the associative property influence the transformation power of the supercompiler SCP4? In fact the questions are questions on language dependence of results of verification.

The answer is as follows. On the one hand associativity of the concatenation simplifies the structure of the programs encoding the models of the protocols. There exist no loops just adding stepwise the terms forming the expressions (i.e. the loops modify in no way the terms). As a consequence the analysis of such a *potentional loop* is shifted to purely syntactic structures of the corresponding configurations representing such concatenation and may be done much more precisely compared to the concatenation loops, which (of course) are in no way marked out by specific syntax and hence, in general, are algorithmically unrecognizable as loops encoding concatenation. On the other hand the algorithm generalizing the configurations during supercompilation becomes ambiguous: the non-trivial relation imposed on the object terms has to be taken into account.

We may imitate the LISP style concatenation by means of the LispAppend function defined in Section 4. The LISP style program encoding the LBM protocol is as follows:

```
RandomAction {
 .........
* r3
(r3  e.r3) (idle I e.1)(busy e.2)(req e.3)(use e.4)
                                 (high e.5)(low e.6) =
          (idle e.1)(busy I e.2)
         <RandomDistribution
                  (r3  (low_use <LispAppend (e.4) (e.6)>) e.r3)
                             (req e.3)(use )(high e.5)(low e.6)>;
* r4
(r4) (idle e.1)(busy I e.2)(req e.3)(use e.4)(high e.5)(low e.6)
         = (idle I e.1)(busy e.2)(req <LispAppend (e.3) (e.6)>)
                    (use <LispAppend (e.4) (e.5)>)(high )(low );
}
```

```
RandomDistribution {
 .........
(r3  (low_use e.lu) (high_low ))
                              (req e.3)(use e.4)(high e.5)(low e.6)
    = (req e.3)(use e.4)(high e.5)(low <LispAppend (e.lu) (e.6)>);
 .........
}
```

Where we omitted sentences and functions coinciding with the REFAL style encoding.

The result of specialization by the supecompiler SCP4 once again does not contain the operators `return FALSE;`, which in REFAL terms have to be represented just as the symbolic constant `FALSE`. Thus the SCP4 succeeds in verification of the LISP style encoding the LBM protocol.

9 Discussion and Further Directions

The correctness of the method is heavily based on the crucial Property 1 of supercompilers. It has been shown, in particular in [38,40,37] that (variants of) supercompilation is a correct transformation, in a sense it always returns (if any) the program equivalent to the input program (on the domain of the latter). So the answer for the above question is positive if the SCP4 indeed implements correctly the supercompilation process as it is described in the above papers. This however is not a trivial question, especially because of the specific semantic assumptions of REFAL, like built-in associativity of concatenation as a term forming construct.

We incorporated a call for the `Test` into the body of `fint` (in its first sentence) to organize the composition $T_Q(\texttt{fint}_S(n, \bar{p}))$ (Section 7). Such a kind of encoding of the *composition* is crucial for successful automatic verification of the LBM protocol. Another important point allowing us to successfully verify the protocol is the following property of the testing function `Test`: the number of its REFAL steps is uniformly bounded on the size of the input data of the function. In fact in our case there is just a single step. That would be very interesting to bring a protocol with a *safety* property not satisfying such a uniform condition to successful verification via supercompilation. The simple protocols given in the survey [14] in terms of counter machines seem to be good candidates to try.

With the point of view of strengthening the supercompilation algorithm based on associative concatenation, it is very important to implement Makanin's algorithm solving string equations [26]. In our opinion, implementation of Khmelevskiĭ's algorithm [12] (working only with such equations with three variables) and using this algorithm for handling of *restrictions* (see [41,44,32,33]) could lead to solution (by supercompilation technology) of new very interesting tasks.

We would like to say again that despite simplicity of the languages generated by the protocols (and, what is more, due to their algorithmic incompleteness), automatic specialization of their interpreters opens very interesting and important problems leading to more fundamental understanding of the nature of program specialization (Section 1). An extension of the class of the protocols and their properties to be successfully verified is a very attractive task. Here we mean both automatic verification *per se* and specialization of the interpreters of the algorithmic incomplete programming languages generated by the protocols. We have to point to a class of elementary algorithms [13,27], which (as far as we know) was still never studied in the context of specialization of its interpreters.

An interesting direction for future work would be to modify supercompiler so that during supercompilation process within the parameterized testing scenario it would produce an inductive proof of safety properties. For any particular successfull verification then one can check the produced inductive proof by a simple proof checker.

Another important direction is to establish completeness results for classes of verification problems and particular strategies of the supercompiler and to compare the method with other verification methods based on program transformations [7,17,18,36].

Acknowledgements. The authors thank anonymous referees for several insightful comments that led to a substantial improvement of the paper.

References

1. Cheng, K.-T., Krishnakumar A.S.: Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. *ACM Transactions on Design Automation of Electronic Systems* 1(1):57–79, 1996.
2. Delzanno, G.: Load Balancing Monitor, at the web page *Automatic Verification of Parameterized Synchronous Systems.* `http://www.disi.unige.it/person/DelzannoG/parameterized.html`
3. Delzanno, G.: Verification of Consistency Protocols via Infinite-state Symbolic Model Checking, A Case Study. In *Proc. of FORTE/PSTV*, pp 171–188, 2000.
4. Delzanno, G.: Contsraint-based Verification of Paremeterized Cache Coherence Protocols. *Formal Methods in System Design* 23(3):257-301, 2003.
5. Futamura, Y.: Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. In: Systems. Computers. Controls. **2(5)** (1971) 45–50
6. Futamura, Y., Nogi, K., Takano, A.: Essence of generalized partial computation. Theoretical Computer Science. **90** (1991) 61–79. Amsterdam. North-Holland Publishing Co.
7. Glück, R., Leuschel, M.: Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In Proc. of the PSI'99, LNCS, Vol. **1755** (1999) 93–100, Springer-Verlag
8. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. (1993) Prentice Hall International
9. Jones, N.D.: What not to do when writing an interpreter for specialization. In Proc. of the PEPM'96, LNCS, Vol. **1110** (1996) 216–237, Springer-Verlag
10. Handy, J.: The Cache Memory Book. Academic Press, 1993.
11. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. **2(7)** (1952) 326–336

12. Khmelevskiĭ, Yu.I.: Equations in Free Semigroups. (in Russian) In I.G. Petrovskiĭ (Ed.), *Trudy Math. Inst. Steklov*, Vol. **107** (1971). English translation in: *Proc. of Steklov Inst. Math., 107*, Amer. Math. Soc., 1976.
13. Kossovski, N.K.: Foundation of the theory of elementary algorithms. (*Book in Russian*), Leningrad University Press, Leningrad, 1987.
14. Ibarra, O.H., Dang, Zh., Yang, L.: On counter machines, Diophantine equations, and reachability problems. In *Proc. of the Workshop on Reachability Problems*, TUCS General Publication, No. 45, Part 2, June 2007, pp 8–24.
15. Kruskal, J.B.: Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. Trans. Amer. Math. Society, **95** (1960) 210–225
16. Leuschel, M.: On the Power of Homeomorphic Embedding for Online Termination. In *Proc. of the SAS'98*, LNCS, **1503** (1998), Springer-Verlag.
17. Leuschel, M., Lehmann, H.: Solving coverability problems of Petri nets by partial deduction. In *Proc. of the 2nd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'2000)*, Montreal, Canada, pp 268-279, 2000.
18. Leuschel, M., Massart, T.: Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi (Ed.), *Logic-Based Program Synthesis and Transformation. Proc. of LOPSTR'99*, LNCS, **1817** (2000) 63–82, Springer-Verlag.
19. Lisitsa, A.P., Nemytykh, A.P.: Verification via Supercompilation. `http://www.csc.liv.ac.uk/~alexei/VeriSuper/`
20. Lisitsa, A.P., Nemytykh, A.P.: Experiments on verification via supercompilation. `http://refal.botik.ru/protocols/`, 2007.
21. Lisitsa, A.P., Nemytykh, A.P.: Towards Verification via Supercompilation. In *Proc. of COMPSAC 05, the 29th Annual International Computer Software and Applications Conference, Workshop Papers and Fast Abstracts*, pages 9-10, IEEE, 2005.
22. Lisitsa, A.P., Nemytykh, A.P.: Verification of parameterized systems using supercompilation. A case study, in Proc. of the Third Workshop on Applied Semantics (APPSEM05), M. Hofmann, H.W. Loidl (Eds.) , Fraunchiemsee, Germany. Ludwig Maximillians Universitat Munchen. (2005), Accessible via: `ftp://www.botik.ru/pub/local/scp/refal5/appsem\_verification2005.ps`
23. Lisitsa, A.P., Nemytykh, A.P.: Verification as a Parameterized Testing (Experiments with the SCP4 Supercompiler). Programmirovanie. No.**1** (2007) (In Russian). English translation in J. Programming and Computer Software, Vol. **33**, No.1 (2007) 14–23.
24. Lisitsa, A.P., Nemytykh, A.P.: Reachability Analysis in Verification via Supercompilation. In *Proc. of the Workshop on Reachability Problems*, TUCS General Publication, No. 45, Part 2, June 2007, pp 53–67.
25. Lisitsa, A.P., Nemytykh, A.P., A Note on Specialization of Interpreters. In *Proc. of the 2nd International Computer Science Symposium in CSR 2007*, LNCS, **4649** (2007) 237–248, Springer-Verlag.
26. Makanin, A.S.: The problem of solvability of equations in a free semigroup. (in Russian) *Matematicheskii Sbornik, 103(2)*, 147–236, 1977. English translation in: *Math. USSR-Sb., 32*, 129–198, 1977.
27. Marchenkov, S.S.: Elementary recursive functions. (*Book in Russian*), Moscow, MCCME, 112 pages, 2003.
28. Markov, A.A.: The Theory of Algorithms (in Russian), *Trudy V.A. Steklov Math. Inst.*, Vol. **42** (1954) 3–374.
29. Mogensen, T.: Evolution of Partial Evaluators: Removing Inherited Limits. In Proc. of the PEPM'96, LNCS, Vol. **1110** (1996) 303–321, Springer-Verlag

30. Nemytykh, A.P.: A Note on Elimination of Simplest Recursions.. In Proc. of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, (2002) 138–146, ACM Press
31. Nemytykh, A.P.: The Supercompiler SCP4: General Structure (extended abstract). In Proc. of the Perspectives of System Informatics, LNCS, **2890** (2003) 162–170, Springer-Verlag
32. Nemytykh, A.P.: The Supercompiler SCP4: General Structure *Program systems: theory and applications*, vol. 1, pp. 448-485. (in English) Moscow, Fizmatlit. 2004. (`ftp://ftp.botik.ru/pub/local/scp/refal5/GenStruct.ps.gz`)
33. Nemytykh, A.P.: The Supercompiler SCP4: General Structure. (*Book in Russian*), URSS, Moscow, 152 pages, 2007.
34. Nemytykh, A.P., Turchin, V.F.: The Supercompiler SCP4: sources, on-line demonstration, `http://www.botik.ru/pub/local/scp/refal5/`, (2000).
35. Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. In J. of Logic Programming. **19,20** (1994) 261–320
36. Roychoudhury, A., Ramakrishnan, I.V.: Inductively Verifying Invariant Properties of Parameterized Systems. In. J. *Automated Software Engineering.* **11** (2004) 101–139
37. Sands, D.: Proving the correctness of recursion-based automatic program transformation. In *Theory and Practice of Software Development*, LNCS, **915** (1995) 681–695, Springer-Verlag
38. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, pp 486–479. MIT Press, 1995.
39. Sørensen, M.H., Glück, R.: Introduction to Supercompilation. *Partial Evaluation - Practice and Theory*, DIKU 1998 International Summer School. June 1998. http://repository.readscheme.org/ftp/papers/pe98-school/D-364.pdf
40. Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. In *Journal of Functional Programming*, **6(6)** (1996) 811–838
41. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems. **8** (1986) 292–325, ACM Press
42. Turchin, V.F.: The algorithm of generalization in the supercompiler. In *Proceedings of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation*, pages 531–549. Amsterdam: North-Holland Publishing Co., 1988.
43. Turchin, V.F.: Refal-5, Programming Guide and Reference Manual. Holyoke, Massachusetts. (1989) New England Publishing Co. (electronic version: `http://www.botik.ru/pub/local/scp/refal5/` ,2000)
44. Turchin, V.F.: Supercompilation: Techniques and results. In *Proc. of PSI'96*, LNCS, **1181** (1996) 227–248, Springer-Verlag

Appendix: Residual Program

```
* InputFormat: <int e.41>
$ENTRY int {
 e.41 (e.101) = <F5 (e.41 ) e.101>;
}
```

```
* InputFormat: <F115 (e.146) (e.147) (e.148) (e.149) e.150>
F115 {
 (e.146) (I e.147) (I e.148) (e.149) e.150
                      = <F115 (e.146 ) (e.147) (e.148) (e.149) I e.150>;
 () (e.147 ) () (e.149 ) e.150 = TRUE;
 ((r4 ) e.146) (e.147) () (e.149) e.150
                      = <F24 (e.146) (e.149 e.147) e.150>;
 () () (e.148) (e.149) e.150 = TRUE;
 ((r4 ) e.146) () (e.148) (e.149) e.150 = <F24 (e.146 ) (e.149 ) e.150>;
}

* InputFormat: <F35 (e.109) (e.110) e.111>
F35 {
 () (e.110) e.111 = TRUE ;
 ((r1 ) e.109) (e.110) e.111 = <F24 (e.109) (e.110) e.111>;
 ((r2 ) e.109) (e.110) I e.111 = <F35 (e.109) (I e.110) e.111>;
 ((r3 (high_low I e.121)) e.109) (e.110) I e.111
                            = <F115 (e.109) (e.111) (e.121) (e.110)>;
 ((r3 (high_low ))) (e.110) e.111 = TRUE ;
 ((r3 (high_low )) (r4 ) e.109) (e.110) e.111
                            = <F5 (e.109) I e.110 e.111>;
 ((r3 (high_low e.121 ))) (e.110) = TRUE;
 ((r3 (high_low e.121 )) (r4 ) e.109) (e.110) = <F5 (e.109) I e.110>;
}

* InputFormat: <F24 (e.109) (e.110) e.111>
F24 {
 () (e.110) e.111 = TRUE;
 ((r1 ) e.109) (I e.110) e.111 = <F24 (e.109) (e.110) I e.111>;
 ((r2 ) e.109) (e.110) e.111 = <F35 (e.109) (e.110) e.111>;
 ((r3 (high_low I e.144)) e.109) (e.110) e.111
                      = <F115 (e.109) (e.111) (e.144) (e.110)>;
 ((r3 (high_low ))) (e.110) e.111 = TRUE;
 ((r3 (high_low )) (r4 ) e.109) (e.110) e.111
                      = <F5 (e.109) e.110 I e.111>;
}

* InputFormat: <F5 (e.41) e.101>
F5 {
 () e.101  = TRUE;
 ((r1 ) e.41) I e.101 = <F24 (e.41) (e.101)>;
 ((r3 (high_low ))) e.101 = TRUE;
 ((r3 (high_low )) (r4 ) e.41) e.101 = <F5 (e.41) e.101>;
 ((r3 (high_low e.163))) e.101 = TRUE;
 ((r3 (high_low e.163)) (r4 ) e.41) e.101 = <F5 (e.41) e.101>;
}
```