Verifying Programs via Intermediate Interpretation

Alexei P. Lisitsa
Department of Computer Science,
The University of Liverpool
a.lisitsa@liverpool.ac.uk

Andrei P. Nemytykh
Program Systems Institute,
Russian Academy of Sciences
nemytykh@math.botik.ru

We explore an approach to verification of programs via program transformation applied to an interpreter of a programming language. A specialization technique known as Turchin's supercompilation is used to specialize some interpreters with respect to the program models. We show that several safety properties of functional programs modeling a class of cache coherence protocols can be proved by a supercompiler and compare the results with our earlier work on direct verification via supercompilation not using intermediate interpretation.

Our approach was in part inspired by an earlier work by De E. Angelis et al. (2014-2015) where verification via program transformation and intermediate interpretation was studied in the context of specialization of constraint logic programs.

Keywords: program specialization, supercompilation, program analysis, program transformation, safety verification, cache coherence protocols

1 Introduction

We show that a well-known program specialization task called the first Futamura projection [11, 37, 17] can be used for indirect verifying some safety properties of functional program modeling a class of non-deterministic parameterized computing systems specified in a language that differs from the object programming language treated by a program specializer deciding the task. Let a specializer transforming programs written in a language $\mathscr{L}$ and an interpreter $\texttt{Int}_{\mathscr{M}}$ of a language $\mathscr{M}$, which is also implemented in $\mathscr{L}$, be given. Given a program $\texttt{p}_0$ written in $\mathscr{M}$, the mentioned task is to specialize the interpreter $\texttt{Int}_{\mathscr{M}}\texttt{(p}_0\texttt{,d)}$ with respect to its first argument, while the data d of the program $\texttt{p}_0$ is unknown.

Our interest in this task has been inspired by the following works [2, 4] especially the presentation given by the authors of these articles at the Workshop VPT-2015. The authors work in terms of *constraint logic programming* (CLP), where the constraint language is the linear arithmetic inequalities imposed on integer values of variables. They use partial deduction [22] and CLP program specialization [9, 3] methods for specializing an interpreter of a C-like language with respect to given programs, aiming at verification of the C-like imperative specifications with respect to the postconditions defined in CLP and defining the same functions (relations) as done by the corresponding C-like programs. Additionally to the CLP program specialization developed by De E. Angelis et al. and called VeriMAP [3] they also use external satisfiability modulo theories (SMT) solvers. We would also refer to an earlier work by J. P. Gallagher et al. [12] reported on a language-independent method for analyzing the imperative programs via intermediate interpretation by a logic programming language. The success of such a method depends eventually on the analysis tools available for the logic programming languages. Note that the transformation examples given in the papers [12, 9, 3] presenting the both mentioned approaches deal with neither function nor constructor application stack in the interpreted programs.

In this paper we focus our attention on self-sufficient methods for specialization of functional programs, aiming at proving some *safety* properties of the programs. Such a property problem is a (un)reachability problem meaning that when a program is running, an unsafe program configuration cannot be

Submitted to:
VPT 2017

reached from any initial configuration. We consider a program specialization method called Turchin's supercompilation [38, 37, 36, 18] and study potential capabilities of the method for verifying the safety properties of the functional programs modeling a class of non-deterministic parameterized cache coherence protocols [6]. We use an approach to functional modeling of non-deterministic computing systems, first presented by these authors in [24, 25, 27]. The simple idea behind the approach is as follows. Given a program modeling a deterministic computing system, whose behavior depends on and is controlled by an input parameter value, let us call for an oracle producing the input value. Then the meta-system including both the program and the external oracle becomes non-deterministic one. And vice versa, given a non-deterministic system, one may be concerned about the behavior of the system only along one possible path of the system evaluation. In such a case, the path of interest may be given as an additional input argument of the system, forcing the system to follow along the path. Dealing with unknown value of the additional parameter one can study any possible evolution of the non-deterministic system, for example, aiming at verifying some properties of the system.

Viability of such an approach to verification has been demonstrated in previous works using supercompilation as a program transformation and analyzing technique [24, 25, 27, 26, 19], where it was applied to safety verification of program models of parameterized protocols and Petri Nets models. Furthermore, the functional program modeling and supercompilation have been used to specify and verify cryptographic protocols, and in the case of insecure protocols a supercompiler was utilized in an interactive search for the attacks on the protocols [1, 32]. In these cases the supercompiler has been used for specializing the corresponding program models, aiming at moving the safety properties of interest from the semantics level of the models to simplest syntactic properties of the residual programs produced by the supercompiler. Later this approach was extended by G. W. Hamilton for verifying wider class of temporal properties of reactive systems [13, 14].

Given a specializer transforming the program written in a language $\mathscr{L}$ and used for program model verification, in order to mitigate the limitation of the specification language $\mathscr{L}$, in this paper we study potential abilities of the corresponding specialization method for verifying the models specified in another language $\mathscr{M}$, removing the intermediate interpretation overheads. We analyze the supercompilation algorithms allowing us crucially to remove the interpretation layer and to verify indirectly the safety properties. The corresponding experiments succeed in verifying some safety properties of the series of parameterized cache coherence protocols specified, for example, in the imperative WHILE language by N. D. Jones [16]. Nevertheless, in order to demonstrate that our method is able to deal with non-imperative interpreted programs, we consider the case when a modelling language $\mathscr{M}$ is a non-imperative subset of the basic language $\mathscr{L}$. On the other hand, that allows us to simplify the presentation. In order to prove the properties of interest, some our program models used in the experiments required one additional supercompilation step (i.e., the corresponding residual programs should be supercompiled once again[1]).

The considered class of cache coherence protocols effectively forms a benchmark on which various methods for parameterized verification have been tried [6, 25, 27]. In [25, 27] we have applied direct verification via supercompilation approach without intermediate interpretation. The corresponding models may be very large and the automatic proofs of their safety properties may have very complicated structures. See, for example, the structure of the corresponding proof [26] produced by the supercompiler SCP4 [28, 29, 31] for the functional program model of the parameterized Two Consumers - Two Producers (2P/2C) protocol for multithreaded Java programs [42]. Taking that into account, the experiment

[1] Note that the method presented in the papers [2, 4] mentioned above sometimes requires a number of iterations of the specialization step and the number is unknown.

presented in this paper can also be considered as a partial verification of the intermediate interpreters $\texttt{Int}_{\mathscr{M}}\texttt{(p,d)}$ used in the experiments. That is to say, a verification of the interpreters with respect to the subset of the input values of the argument p, being the program models of the cache coherence protocols.

A supercompiler is a program specilalizer based on the supercompilation technique. The program examples given in this paper were specialized by the supercompiler SCP4 [28, 29, 31]. We present our interpreter examples in a variant of a pseudocode for a functional program while real supercompilation experiments with the programs were done in the strict functional programming language Refal [40][2], [41] being both the object and implementation language of the supercompiler SCP4. One of advantages of using supercompilation, instead of other forms of partial evaluation or CLP specialization, is the use of Turchin's relation 4.2 (see also [39, 29, 34]) defined on function-call stacks, where the function calls are labeled by the times when they are generated by the unfold-fold loop. This relation is responsible for accurate generalizing the stack structures of the unfolded program configurations. It is *based on global properties* of the path in the corresponding unfolded tree rather than on the structures of two given configurations in the path. Turchin's relation both stops the loop unfolding the tree and *provides an evidence of how a given call-stack structure has to be generalized*. Proposition 1 proven in this paper shows that a composition of the Turchin and Higman-Kruskal relations may prevent generalization of two given interpreter configurations encountered inside one big-step of the interpreter. Such a prevention from generalization is crucial for optimal specialization of any interpreter with respect to a given program.

This paper assumes that the reader has basic knowledge of concepts of functional programming, pattern matching, term rewriting systems, and program specialization.

The contributions of this paper are: (1) Developing a method aiming at uniform reasoning on properties of configurations' sequences that are encountered in specializing an interpreter of a Turing complete language. (2) In particular, we have proved the following statement. Consider specialization of the interpreter with respect to any interpreted program from an infinite program set that is large enough to specify a series of parameterized cache coherence protocols, controlled by a composition of the Turchin 4.2 and Higman-Kruskal 4.1 relations. Given a big-step of the interpreter to be processed by the unfold-fold loop, we assume that no both generalization and folding actions were still done by this loop up to the moment considered. Then any two non-transitive 4.3 big-step internal configurations C_1, C_2 are prevented from both generalization and folding actions. (3) We show that supercompilation controlled by the composition of the relations above is able to verify some safety properties of the series of parameterized cache coherence protocols via intermediate interpretation of their program models. Note that these program specifications include both the function call and constructor application stacks, where the size of the first one is uniformly bounded on the input parameter while the second one is not. Unlike VeriMAP [3] our indirect verification method involves no post-specialization unfold-fold.

The paper is organized as follows. In Section 2 we describe the syntax and semantics of a pseudocode for a subset of the strict functional language Refal which will be used throughout this paper. We also give the operational semantics of the subset, defining its "self-interpreter". In Section 3 we outline our approach for specifying non-deterministic systems by an example used through this paper. In Section 4 we shortly introduce an unfold-fold program transformation method known as Turchin's supercompilation that is used in our experiments presented in this paper. We describe the strategy controlling the unfold-fold loop. The corresponding relation is a composition of Turchin's relation and a variant of the Higman-Kruskal relation. This composition plays a central role in verifying the safety properties of the

[2]The reader is welcome to execute several sample Refal programs and even any program written by the user directly from the electronic version of the Turchin book.

cache coherence protocols' models via intermediate interpretation. In Section 5 we prove in a uniform way a number of properties of a huge amount of the complicated configurations generated by specialization of the self-interpreter with respect to the given program modeling a cache coherence protocol. The argumentations given in the section are applicable for the whole series of the protocols mentioned in Section 6. Developing the method of such argumentations is the main aim of the paper and *both Proposition 1 and the proof of this proposition are the main results of the paper*. The statement given in Proposition 1 can be applied to a wide class of interpreters of Turing complete programming languages. This statement is a theoretical basis to the following. Why the approach suggested in the paper does succeed in verifying the safety properties of the series of the cache coherence protocols via intermediate interpretation. Finally, in Section 6 we report on some other experimental results obtained by using the approach, discuss the results presented in the paper, and compare our experiments with other ones done by existing methods.

2 An Interpreter for a Fragment of the SCP4 Object Language

A first interpreter we consider is an interpreter of a subset $\mathscr{L}$ of the SCP4 object language, which we aim to put in between the supercompiler SCP4 and programs modeling the cache coherence protocols to be verified. We will refer to this interpreter as a "self-interpreter".

In this section, we describe the syntax and semantics of a pseudocode for the subset of the strict functional language Refal which will be used throughout this paper. We also give the operational semantics (that is, the interpreter) of the subset.

2.1 Language

$prog$ ::=	$def_1 \ \ldots \ def_n$	Program
def ::=	$f(\ ps_1\ ) \Rightarrow exp_1;\ \ \ldots;\ \ f(\ ps_n\ ) \Rightarrow exp_n;$	Function Definition
exp ::=	v	Variable
|	$term : exp$	*Cons* Application
|	$f(\ exp_1, \ldots, exp_n\ )$	Function Application
|	exp_1 **++** exp_2	*Append* Application
|	`[]`	*Nil*
$term$ ::=	$s.name$	Symbol-Type Variable
|	(exp)	Constructor Application
|	σ	Symbol
ps ::=	$p_1, \ldots, p_n$	Patterns
p ::=	v	
|	$s.name : p$	
|	$(p_1) : p_2$	
|	$\sigma : p$	
|	`[]`	
v ::=	$s.name \mid e.name$	Variable

Programs in $\mathscr{L}$ are *strict* term rewriting systems based on pattern matching.

The rules in the programs are ordered from the top to the bottom to be matched. To be closer to Refal we use two kinds of variables: *s.*variables range over *symbols* (i.e., characters and identifiers, for example, `'a'` and *True*), while *e.*variables range over the whole set of the S-expressions.[3] Given a rule $l \Rightarrow r$, any variable of r should appear in l. Each function f has a fixed arity, i.e., the arities of all left-hand sides of the rules of f and any expression $f(\ exp_1,\ ...,\ exp_n\ )$ must equal the arity of f. The parenthesis constructor $(\bullet)$ is used without a name. *Cons* constructor is used in infix notation and may be omitted. The patterns in a function definition are not exhaustive. If no left-hand side of the function rules matches the values assigned to a call of the function, then executing the call is interrupted and its value is undefined. In the sequel, the expression set is denoted by $\mathbb{E}$; $\mathbb{D}$ and $\mathbb{S}$ stand for the data set, i.e., the patterns containing no variable, and the symbols set, respectively. The name set of the functions of an arity n is denoted by $\mathbb{F}_n$ while $\mathbb{F}$ stands for $\bigcup_{n=0}^{\infty}\mathbb{F}_n$. $\mathscr{V}_e$ and $\mathscr{V}_s$ stand for the *e*- and *s*-variable sets, respectively, and $\mathscr{V}$ denotes $\mathscr{V}_e \cup \mathscr{V}_s$. Let an expression *exp* be given. $\mathscr{V}_e(exp)$, $\mathscr{V}_s(exp)$, $\mathscr{V}(exp)$ denote the corresponding variable sets of *exp*. $\mu_v(exp)$ denotes the multiplicity of $v \in \mathscr{V}$ in *exp*, i. e., the number of all the occurrences of v in *exp*. *exp* is called passive if no function application occurs in *exp* otherwise it is called an active expression. $\mathscr{T}$ stands for the term set, σ stands for a symbol. Given an expression *exp* and a variable substitution θ, $exp\theta$ stands for $\theta(exp)$.

2.2 Encoding

In our experiments considered in this paper the protocol program models have to be input values of the interpreter argument with respect to which the interpreter is specialized. Thus the program model should be encoded in the data set of the implementation language of the interpreter. The program models used in this paper are written in a fragment of the language described in Section 2.1, where `++` constructor is not allowed and only unary functions may appear.

Now we have to define the corresponding encoding function denoted by the underline, where the function $\mathfrak{A}$ groups the program rules belonging to the same function as it is shown in the second definition line.

$\underline{prog} = (\ \underline{\mathfrak{A}(\ prog\ )}\ )$;		Program
$\underline{f\{rules\}\ defs} = (f\ \ \underline{rules}\ ) : \underline{defs}$;		Function Definitions
$\underline{rule;\ rules} = (\ \underline{rule}\ ) : \underline{rules}$;		Rules
$\underline{f(\ pattern\ )\ \Rightarrow\ exp} = (\ \underline{pattern}\ )\ :$ `'='` $:\ (\ \underline{exp}\ )$;		Rule
$\underline{term : exp} = \underline{term}\ :\ \underline{exp}$;		Here $term ::= (exp) \mid s.name \mid \sigma$
$\underline{(exp)} = ($`'*'` $\underline{exp}\ )$;	$\underline{f(\ exp\ )} = (Call\ f\ \ \underline{exp}\ )$;	Applications
$\underline{e.name} = (Var$ `'e'` $name)$;	$\underline{s.name} = (Var$ `'s'` $name)$;	Variables
$\underline{[]} = []$;	$\underline{\sigma} = \sigma$;	*Nil* and Symbol

Note that any pattern is an expression.

Supercompiler SCP4 in its processing dealing with programs as input data uses this encoding function and utilizes its properties. The image of $\mathbb{D}$ under the encoding is a proper subset of $\mathbb{D}$, i. e., $\underline{\mathbb{D}} \neq \mathbb{D}$. For example, $(Var$ `'s'` $name) \notin \underline{\mathbb{D}}$.

[3] This fragment of Refal is introduced for the sake of simplicity. The reader may think that the syntactic category *exp* of list expressions and the parenthesis constructor are Lisp equivalents. Actually Refal does not include *Cons* constructor but, instead of *Cons*, *Append* is used as an associative constructor. Thus the Refal data set is wider as compared with the Lisp data set: the first one is the set of finite sequences of *arbitrary* trees, while the second one is the set of binary trees. See [40] for details.

```
Int( (Call s.f e.d), e.P ) ⇒ Eval( EvalCall( s.f, e.d, e.P ), e.P );

Eval( (e.env) : (Call s.f e.q) : e.exp, e.P )
        ⇒ Eval( EvalCall( s.f, Eval( (e.env) : e.q, e.P ), e.P ), e.P ) ++ Eval( (e.env) : e.exp, e.P );
Eval( (e.env) : (Var e.var) : e.exp, e.P ) ⇒ Subst( e.env, (Var e.var) ) ++ Eval( (e.env) : e.exp, e.P );
Eval( (e.env) : ('*' e.q) : e.exp, e.P ) ⇒ ('*' Eval( (e.env) : e.q, e.P )) : Eval( (e.env) : e.exp, e.P );
Eval( (e.env) : s.x : e.exp, e.P ) ⇒ s.x : Eval( (e.env) : e.exp, e.P );
Eval( (e.env) : [], e.P ) ⇒ [];
    EvalCall( s.f, e.d, (Prog s.n) ) ⇒ Matching( F, [], LookFor( s.f, Prog( s.n ) ), e.d );

Matching( F, e.old, ((e.p) : '=' : (e.exp)) : e.def, e.d )
        ⇒ Matching( Match( e.p, e.d, ([]) ), e.exp, e.def, e.d );
Matching( (e.env), e.exp, e.def, e.d ) ⇒ (e.env) : e.exp;
    Match( (Var 'e' s.n), e.d, (e.env) ) ⇒ PutVar( (Var 'e' s.n) : e.d, (e.env) );
    Match( (Var 's' s.n) : e.p, s.x : e.d, (e.env) )
            ⇒ Match( e.p, e.d, PutVar( (Var 's' s.n) : s.x, (e.env) ) );
    Match( ('*' e.q) : e.p, ('*' e.x) : e.d, (e.env) )
            ⇒ Match( e.p, e.d, Match( e.q, e.x, (e.env) ) );
    Match( s.x : e.p, s.x : e.d, (e.env) ) ⇒ Match( e.p, e.d, (e.env) );
    Match( [], [], (e.env) ) ⇒ (e.env);
    Match( e.p, e.d, e.fail ) ⇒ F;

PutVar( e.assign, (e.env) ) ⇒ CheckRepVar( PutV( (e.assign), e.env, [] ) );
    PutV( ((Var s.t s.n) : e.val), ((Var s.t s.n) : e.pval) : e.env, e.penv )
            ⇒ (Eq( e.val, e.pval )) : ((Var s.t s.n) : e.pval) : e.env;
    PutV( (e.assign), (e.passign) : e.env, e.penv )
            ⇒ PutV( (e.assign), e.env, (e.passign) : e.penv );
    PutV( (e.assign), [], e.penv ) ⇒ (T) : (e.assign) : e.penv;

    CheckRepVar( (T) : e.env ) ⇒ (e.env);
    CheckRepVar( (F) : e.env ) ⇒ F;

Eq( s.x : e.xs, s.x : e.ys ) ⇒ Eq( e.xs, e.ys );
Eq( ('*' e.x) : e.xs, ('*' e.y) : e.ys ) ⇒ ContEq( Eq( e.x, e.y ), e.xs, e.ys );
Eq( [], [] ) ⇒ T;
Eq( e.xs, e.ys ) ⇒ F;
    ContEq( F, e.xs, e.ys ) ⇒ F;
    ContEq( T, e.xs, e.ys ) ⇒ Eq( e.xs, e.ys );

LookFor( s.f, (s.f : e.def) : e.P ) ⇒ e.def;
LookFor( s.f, (s.g : e.def) : e.P ) ⇒ LookFor( s.f, e.P );

Subst( ((Var s.t s.n) : e.val) : e.env, (Var s.t s.n) ) ⇒ e.val;
Subst( (e.assign) : e.env, e.var ) ⇒ Subst( e.env, e.var );
```

Figure 1: Self-Interpreter

2.3 The Interpreter

The self-interpreter used in the experiments is given in Figure 1.

The entry point is of the following form *Int*((*Call s.f e.d*), *e.P*). Here the first argument is the application constructor of the main function to be executed. The second argument provides the name of a program to be interpreted. The encoded source of the program will be returned by a function call of *Prog* whenever it is asked by *EvalCall*. For example, interpretation of program model Synapse N+1 given in Section 3.1 starts with the following application *Int*((*Call Main* $\underline{d_0}$), (*Prog Synapse*)), where d_0 is an input data given to program Synapse. Due to the large size of the encoded programs we omit the definition of function *Prog*.

EvalCall((*Call s.f e.d*), *e.P*) asks for the definition of function *s.f*, calling *LookFor*, and initiates matching the data given by *e.d* against the patterns of the definition rules. In order to start this pattern matching, it imitates a fail happening in matching the data against a previous nonexistent pattern.

Function *Matching* runs over the definition rules, testing the result of matching the input data (*e.d*) against the current pattern considered. In the case if the result is *F* the function calls function *Match*, asking for matching the input data against the next rule pattern. The environment *e.env* is initialized by `[]` (see the third argument of the *Match* call). If the pattern matching succeeds then function *Matching* returns (*e.env*) : *e.exp*, where expression *e.exp* is the right-hand side of the current rule and the environment includes the variable assignments computed by the pattern matching.

Function *Match* is trying to match the input data given in its second argument, step by step, against the pattern given in its first argument. It computes the environment containing the variable substitution defined by the matching (see the first and second function rules). If a variable is encountered then function *PutVar* calls *PutV* looking for an assignment to the same variable and, if such an assignment exists, the function tests a possible coincidence of the new and old values assigned to the variable. The third rule of function *Match* deals with the tree structure, calling this function twice.

Function *Eval* passes through an expression given in its second argument. The second *Eval* rule deals with a variable and calls function *Subst* looking the environment for the variable value and replacing the variable with its value.

We intend to specialize interpreter *Int* with respect to its second argument. The corresponding *Refal* source code of the self-interpreter may be found in
`http://refal.botik.ru/protocols/Self-Int-Refal.zip`.

3 Specifying Cache Coherence Protocols

We show our method [25, 27] for specifying non-deterministic systems by an example used through this paper. The Synapse N+1 protocol definition given below is borrowed from [5]. The parameterized version of the protocol is considered and *counting abstraction* is used in the specification. The protocol has to react to five external non-deterministic events by updating its states being three integer counters. The initial value of counter *invalid* is parameterized (so it could be any positive integer), while the other two counters are initialized by zero. The primed state names stand for the updated state values. The empty updates mean that nothing happened.

(rh) $dirty + valid \geq 1 \ \rightarrow \ .$

(rm) $invalid \geq 1 \ \rightarrow dirty' = 0, valid' = valid + 1, invalid' = invalid + dirty - 1 \ .$

(wh1) $dirty \geq 1 \ \rightarrow \ .$

(wh2) $valid \geq 1 \ \rightarrow valid' = 0, dirty' = 1, invalid' = invalid + dirty + valid - 1 \ .$

(wm) $invalid \geq 1 \ \rightarrow valid' = 0, dirty' = 1, invalid' = invalid + dirty + valid - 1 \ .$

Specification of Safety Properties Any state reached by the protocol should not satisfy any of the two following properties: (1) $invalid \geq 0, dirty \geq 1, valid \geq 1$; (2) $invalid \geq 0, dirty \geq 2, valid \geq 0$.

$Main(\ (e.time) : (e.is)\ ) \Rightarrow Loop(\ (e.time)\ :\ (Invalid\ I\ e.is) : (Dirty\ ) : (Valid\ )\ );$

$Loop(\ (\ [\ ]\ ) : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )$
$\quad \Rightarrow Test(\ (Invalid\ e.is)\ :\ (Dirty\ e.ds)\ :\ (Valid\ e.vs)\ );$
$Loop(\ (s.t\ :\ e.time)\ :\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )$
$\quad \Rightarrow Loop(\ (e.time)\ :\ Event(\ s.t\ :\ (Invalid\ e.is)\ :\ (Dirty\ e.ds)\ :\ (Valid\ e.vs)\ )\ );$

$Event(\ rm\ :\ (Invalid\ I\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )$
$\quad \Rightarrow (Invalid\ Append(\ (e.ds)\ :\ (e.is)\ ))\ :\ (Dirty\ )\ :(Valid\ I\ e.vs);$
$Event(\ wh2\ :\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ I\ e.vs)\ )$
$\quad \Rightarrow (Invalid\ Append(\ (e.vs)\ :\ (Append(\ (e.ds)\ :\ (e.is)\ ))\ ))\ :\ (Dirty\ I)\ :\ (Valid\ );$
$Event(\ wm\ :\ (Invalid\ I\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )$
$\quad \Rightarrow (Invalid\ Append(\ (e.vs)\ :\ (Append(\ (e.ds)\ :\ (e.is)\ ))\ ))\ :\ (Dirty\ I)\ :\ (Valid\ );$
$\quad Append(\ (\ [\ ]\ ) : (e.ys)\ ) \Rightarrow e.ys;$
$\quad Append(\ (s.x\ :\ e.xs) : (e.ys)\ ) \Rightarrow s.x\ :\ Append(\ (e.xs)\ :\ (e.ys)\ );$

$Test(\ (Invalid\ e.is) : (Dirty\ I\ e.ds) : (Valid\ I\ e.vs)\ ) \Rightarrow False;$
$Test(\ (Invalid\ e.is) : (Dirty\ I\ I\ e.ds) : (Valid\ e.vs)\ ) \Rightarrow False;$
$Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ ) \Rightarrow True;$

Figure 2: Model of the Synapse N+1 cache coherence protocol

3.1 Program Model of the Synapse N+1 Cache Coherence Protocol

Let us then specify a program model of our sample protocol. The Synapse N+1 program is given in Figure 2. The idea behind the program specifications modeling the reactive systems is given in Introduction 1 above. The *finite* stream of events is modeled by a *time* value. The time ticks are labeled by the events. The counters' values are specified in the unary notation. The unary addition is directly defined by function *Append*, i.e., without referencing to the corresponding macros. Function *Loop* exhausts the event stream, step by step, and calls for *Test* verifying the safety property required from the protocol. Thus function *Main* is a predicate.

Note that given input values the partial predicate terminates since the event stream is finite. The termination is normal, if the final protocol state asked by the input stream is reachable one, otherwise it is abnormal.

4 On Supercompilation

In this paper we are interested in one particular approach in program transformation and specialization, known as supercompilation[4]. Supercompilation is a powerful semantics-based program transformation

[4]From *super*vised *compilation*.

technique [36, 38] having a long history well back to the 1960-70s, when it was proposed by V. Turchin. The main idea behind a supercompiler is to observe the behavior of a functional program p running on a *partially* defined input with the aim to define a program, which would be equivalent to the original one (on the domain of the latter), but having improved properties. Given a program and its parameterized entry point, supercompilation is performed by an *unfold-fold cycle* unfolding this entry point. It reduces the redundancy that could be present in the original program. It folds the tree into a finite graph of states and transitions between possible parameterized configurations of the computing system. And, finally, it analyses global properties of the graph and specializes this graph with respect to these properties (without additional unfolding steps).[5] The resulting program definition is constructed solely based on the meta-interpretation of the source program rather than by a (step-by-step) transformation of the program. The result of supercompilation may be a specialized version of the original program, taking into account the properties of partially known arguments, or just a re-formulated, and sometimes more efficient, equivalent program (on the domain of the original).

Turchin's ideas have been studied by a number of authors for a long time and have, to some extent, been brought to the algorithmic and implementation stage [31]. From the very beginning the development of supercompilation has been conducted mainly in the context of the programming language Refal [28, 29, 30, 40]. A number of model supercompilers for subsets of functional languages based on Lisp data were implemented with the aim of formalizing some aspects of the supercompilation algorithms [18, 20, 36]. The most advanced supercompiler for Refal is SCP4 [28, 29, 31].

The verification system VeriMAP [3] by De E. Angelis et al. [2, 4] uses nontrivial properties of integers recognized by both CLP built-in predicates and external SMT solvers. We, just like the VeriMAP authors, use a nontrivial property of the configurations. The property is the associativity of the built-in append function ++ supported by the supercompiler SCP4 itself[6], rather than by an external solver.

4.1 The Well-Quasi-Ordering on $\mathbb{E}$

The following relation is a variant of the Higman-Kruskal relation and is a well-quasi-ordering [15, 21] (see also [23]).

Definition 1 *The homeomorphic embedding relation $\preceq$ is the smallest transitive relation on $\mathbb{E}$ satisfying the following properties, where $f \in \mathbb{F}_n$, $\alpha, \beta, \tau, s, t, t_1, \ldots, t_n \in \mathbb{E}$ and $\alpha, \beta, \tau \in \mathscr{T}$.*

(4.1) $\forall x, y \in \mathscr{V}_e.\ x \preceq y, \forall u, v \in \mathscr{V}_s.\ u \preceq v$; *Variables*

(4.2) $[] \preceq t$, $t \preceq t$, $t \preceq f(t_1, \ldots, t, \ldots, t_n)$, $t \preceq (t)$, $t \preceq \alpha : t$; *Monotonicity*
if $s \preceq t$ *and* $\alpha \preceq \beta$, *then* $(s) \preceq (t), \alpha : s \preceq \beta : t$;
if $s \preceq t$, *then* $f(t_1, \ldots, s, \ldots, t_n) \preceq f(t_1, \ldots, t, \ldots, t_n)$.

Note that the definition takes into account function *append*, since its infix notation exp_1 `++` exp_2 stands for `append(` exp_1, exp_2 `)`. We use relation $\preceq$ modulo associativity of `++` and the following equalities $term : exp_1 = term$ `++` exp_1, exp `++` `[]` $= exp$ and `[]` `++` $exp = exp$.

Corollary 1 *If $t_1, \ldots, t_{n-1} \in \mathscr{T}$ and $t_n \in \mathbb{E}$, and given $i_1, \ldots, i_j$ such that $1 \leq i_1 < i_2 < \ldots < i_j \leq n$, then $t_{i_1} : \ldots : t_{i_j} \preceq t_1 : \ldots : t_n$.*

[5] See also Section 7.1 in Appendix.

[6] As well as by the real programming language in terms of which the experiments described in this paper were done.

Corollary 2 *For any* $f \in \mathbb{F}_n,\ \alpha,\beta,\gamma,t_1,\ldots,t_n \in \mathbb{E}$ *(monotonicity of relation* $\not\preceq$ *)*
(4.3) *if* $\alpha \not\preceq \beta$, *then* $f(\,t_1,\,\ldots,\,t_{i-1},\,\alpha,\,t_{i+1},\,\ldots,\,t_n\,) \not\preceq f(\,t_1,\,\ldots,\,t_{i-1},\,\beta,\,t_{i+1},\,\ldots,\,t_n\,)$.

Given an infinite sequence of expressions $t_1,\ldots,t_n,\ldots$, relation $\preceq$ is relevant to approximation of loops increasing the syntactical structures in the sequence; or in other words to looking for the regular similar cases of mathematical induction on the structure of the expressions. That is to say the cases, which allow us to refer one to another by a step of the induction.

An additional restriction separates the basic cases of the induction from the regular ones. The restriction is: $\forall\sigma\,.\,(\,[\,]\,) \not\preceq (\,\sigma\,)\ \&\ \forall v \in \mathscr{V}_s.(\,[\,]\,) \not\preceq (\,v\,)$.

We impose this restriction on the relation $\preceq$ modulo the equalities above and denote the obtained relation as $\preccurlyeq$. It is easy to see that such a restriction does not violate the quasi-ordering property. Note that the restriction may be varied in the obvious way, but for our experiments its simplest case given above is used to control generalization and has turned out to be sufficient. In the sequel, $t_1 \prec t_2$ stands for the following relation $t_1 \preccurlyeq t_2$ and $t_1 \neq t_2$, which is transitive.

Definition 2 *A parameterized configuration is a finite sequence of the form*
`let` $e.h = f_1(\,exp_{11},\ \ldots\ ,\,exp_{1m}\,)$ `in` $\ldots$ `let` $e.h = f_k(\,exp_{k1},\ \ldots\ ,\,exp_{kj}\,)$ `in` exp_{n+1}, *where* exp_{n+1} *is passive, for all* $i > 1$ $\mu_{e.h}(f_i(\,\ldots\,)) = \mu_{e.h}(exp_{n+1}) = 1$, *and* $\mu_{e.h}(f_1(\,\ldots\,)) = 0$*; for all i and all j variable e.h does not occur in any function application being a sub-expression of* exp_{ij}. *In the sequel, we refer to such a function application* $f_i(\,\ldots\,)$ *given explicitly in the configuration as an upper function application.*

The configurations present the function application stacks, in which all constructors' applications not occurring in arguments of the upper function applications are moved to the rightmost expressions. Here the *append* `++` is treated as a complex constructor[7] , rather than a function. The rightmost expression is the bottom of the stack. Since the value of $e.h$ is reassigned in each `let` in the stack, for brevity sake, we use the following presentation of the configurations:
$f_1(\,exp_{11},\ \ldots\ ,\,exp_{1m}\,),\ \ldots\ ,f_k(\,exp_{k1},\ \ldots\ ,\,exp_{kj}\,),\ exp_{n+1}$, where variable $e.h$ is replaced with bullet $\bullet$. I.e., the bullet is just a placeholder. The last expression may be omitted if it equals $\bullet$. An example follows $f(\,a\ :\ e.xs$ `++` $e.ys\,),\ g(\,\bullet$ `++` $e.ys,\ (Var\ b\ c),\ [\,]\,),\ f(\,s.x\ :\ \bullet\,),\ s.x\ :\ \bullet$ `++` $t(\,s.x\ :\ e.zs\,),\ \bullet$.

4.2 The Well-Disordering on Timed Configurations

Let a program to be specialized and a path starting at the root of the tree unfolded by the unfold-fold loop widely used in program specialization be given. The vertices in the path are labeled by the program parameterized configurations. These configurations form a sequence. Given a configuration from such a sequence and a function application from the configuration, we label the application by the time when it is generated by the unfold-fold loop. Such a labeled function application is said to be a timed application. A configuration is said to be timed if all upper function applications in the configuration are timed. Note that, given a timed configuration, all its timed applications have differing time-labels. Given two different configurations C_1, C_2, if the unfold-fold loop copies an upper function application from C_1 and uses this copy in C_2, then C_1, C_2 share this timed application. In the sequel, a sequence of the timed configurations generated by the unfold-fold loop is also called just a path.

In this section we define a binary relation $\lhd$ on the timed configurations in the path. The relation is originated from V. F. Turchin [39] (see also [29, 33, 34]). It is *not* transitive[8], but like the well-quasi-ordering it satisfies the following crucial property used by supercompilation to stop the loop unfolding the tree.

[7] I.e., we use nontrivial properties of configurations containing the *append* `++`. See the remark made in the footnote on p. 5.
[8] See Section 7.3 in Appendix for an example of the non-transitivity.

For any infinite path $C_1, C_2, \ldots, C_n, \ldots$ there exist two timed configurations C_i, C_j such that $i < j$ and $C_i \lhd C_j$ (see [39, 34]). For this reason we call relation $\lhd$ a well-disordering relation. In the sequel, the time-labels are denoted with subscripts.

Definition 3 *Given a sequence of timed configurations $C_1, \ldots C_n, \ldots$; C_i and C_j are elements of the sequence such that $i < j$ and $C_i ::= f^1_{t_1}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_1$, $C_j ::= g^1_{\tau_1}(\ldots), \ldots, g^m_{\tau_m}(\ldots), exp_2$, where $f^s_{t_s}$ and $g^q_{\tau_q}$, $1 \le s \le k$ and $1 \le q \le m$, stand for function names f^s, g^q labeled with times t_s and τ_q, respectively, and exp_1, exp_2 are passive expressions.*

If $k \le m$, $\delta = m - k$ and $\exists l\,.\,(1 < l \le k)$ such that $\forall s\,.\,(0 \le s \le k-l)$ $f^{l+s}_{t_{l+s}} = g^{\delta+l+s}_{\tau_{\delta+l+s}}$ (i.e., $f^{l+s} = g^{\delta+l+s}$ and $t_{l+s} = \tau_{\delta+l+s}$ hold), $f^{l-1}_{t_{l-1}} \ne g^{\delta+l-1}_{\tau_{\delta+l-1}}$, and $\forall s\,.\,(0 < s < l)$ $f^s_{t_s} \simeq g^s_{\tau_s}$ (i.e., $f^s = g^s$), then $C_i \lhd C_j$.

We say that configurations C_i, C_j are in Turchin's relation $C_i \lhd C_j$. This longest coincided rest of the configurations are said to be a context, while the parts equal one to another modulo their time-labels are called prefixes of the corresponding configurations.

The idea behind this definition is as follows. The function applications in the context never took a part in computing the configuration C_j, in this segment of the path, while any function applications in the prefix of C_i took a part in computing the configuration C_j. Since the prefixes of C_i, C_j coincide modulo their time-labels, these prefixes approximate a loop in the program being specialized. The prefix of C_i is the entry point in this loop, while the prefix of C_j initiates the loop iterations. The common context approximates computations after this loop. Note that Turchin's relation does not impose any restriction on the arguments of the function applications in C_i, C_j.

For example, consider the following two configurations $C_1 ::= f_4(\ldots), f_3(\ldots), g_2(\ldots), t_1(\ldots), \bullet$ and $C_2 ::= f_{10}(\ldots), f_7(\ldots), g_5(\ldots), \ldots, t_1(\ldots), \bullet$, then $C_1 \lhd C_2$ holds. Here the context is $t_1(\ldots)$, the prefix of C_1 is $f_4(\ldots), f_3(\ldots), g_2(\ldots)$, and the prefix of C_2 is $f_{10}(\ldots), f_7(\ldots), g_5(\ldots)$, where the subscripts of the application names stand for the time-labels. See also Section 7.3 for a detailed example regarding Turchin's relation.

4.3 The Strategy Controlling the Unfolding Loop

Now we describe the main relation controlling the unfold-fold loop. That is to say, given a path starting at the root of the unfolded tree, and two timed configurations C_1, C_2 in the path such that C_1 was generated before C_2, this relation stops the loop unfolding the tree and calls the tools responsible for folding this path. These tools, firstly, attempt to fold C_2 by a previous configuration and, if that is impossible, then attempt to generalize this configuration pair. The relation is a composition of relations $\lhd$ and $\preccurlyeq$. It is denoted with $\lhd \circ \preccurlyeq$ and is a well-disordering (see [39, 34]).

Thus we are given two timed configurations C_1, C_2 from a path, such that C_1 is generated before C_2, and C_2 is the last configuration in the path. If relation $C_1 \lhd C_2$ does not hold, then the unfold-fold loop unfolds the current configuration C_2 and goes on. In the case relation $C_1 \lhd C_2$ holds, these configurations are of the forms (see Section 4.2 for the notation used below):
$C_1 = f^1_{t_1}(\ldots), \ldots, f^{l-1}_{t_{l-1}}(\ldots), f^l_{t_l}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_1$,
$C_2 = f^1_{\tau_1}(\ldots), \ldots, f^{l-1}_{\tau_{l-1}}(\ldots), g^l_{\tau_l}(\ldots), \ldots, g^m_{\tau_m}(\ldots), f^l_{t_l}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_2$, where the context starts at $f^l_{t_l}(\ldots)$. Let C^p_i stand for the prefix of C_i, and C^c_i stand for the context of C_i followed by exp_i.

Now we compare the prefixes as follows. If there exists i $(1 \le i < l)$ such that $f^i_{t_i}(\ldots) \preccurlyeq f^i_{\tau_i}(\ldots)$ does not hold, then C_2 is unfolded and the unfold-fold loop goes on. Else, the sub-tree rooted in C_1 is removed and the specialization task defined by the C_1 is decomposed into the two specialization tasks corresponding to C^p_1 and C^c_1. Further the usual attempts to fold C^p_2 by C^p_1 and C^c_2 by C^c_1, respectively, do work. If some of these attempts fail, then the corresponding configurations are generalized. Note

that the context may be generalized despite the fact that it does not take a part in computing the current configuration C_2, since a narrowing of the context parameters may have happened.

A parameterized program configuration is said to be a transitive configuration if one-step unfolding of the configuration results in a tree containing only the vertices with at most one outgoing edge. That is to say, any of the tree vertices is not a branching vertex. For example, any function application of the form $f(d_1,\ldots,d_n)$, where any $d_i \in \mathbb{D}$, is transitive.

For the sake of simplicity, in the experiments described in this paper, the following strategy is used. The unfold-fold loop skips all transitive configurations encountered and removes them from the tree being unfolded. In the sequel, we refer to the strategy described in this section, including relation $\lhd \circ \preccurlyeq$, as the $\lhd \circ \preccurlyeq$-strategy.

5 Indirect Verifying the Synapse N+1 Program Model

In this section we present an application of our program verification method based on supercompilation of intermediate interpretations. In general the method may perform a number of program specializations[9], but all the cache coherence protocol program models that we have tried to verify by supercompiler SCP4 require at most two specializations.

Given a program partial predicate modeling both a cache coherence protocol and a safety property required from the protocol, we use supercompilation aiming at moving the property hidden in the program semantics to a simple syntactic property of the residual program generated by supercompilation, i.e., this syntactic property should be easily recognized. In the experiments discussed in this paper we hope the corresponding residual programs will include no operator *return False*;. Since the original direct program model terminates on any input data (see Section 3.1), this property means that the residual predicate never returns *False* and always *True*. Thus we conclude the original program model satisfies the given safety property. In the terms of functional language $\mathscr{L}$ presented in Section 2.1 the corresponding syntactic property is *"No rule's right-hand side contains identifier False"*.[10]

We can now turn to the program modeling the Synapse N+1 protocol given in Section 3.1. In order to show that the Synapse program model is safe, below we specialize the self-interpreter *Int* (see Section 2.3) with respect to the Synapse program model rather than the program model itself. Since program Synapse terminates, the self-interpreter terminates when it interprets any call of the *Main* entry function of Synapse. Since Synapse is a partial predicate, the calls of the forms $Int((Call\ Main\ e.d),\ (Prog\ Synapse))$, where $e.d$ takes any data, define the same partial predicate. Hence, the self-interpreter restricted to such calls is just another program model of protocol Synapse N+1. This indirect program model is much more complicated as compared with the direct model. We intend now to show that supercompiler SCP4 [28, 29, 31] is able to verify this model. Thus our experiments show potential capabilities of the method for verifying the safety properties of the functional programs modeling some complex nondeterministic parameterized systems. In particular, the experiments can also be considered as a partial verification of the intermediate interpreter used. In other words, verifying the interpreter with respect to a set of the interpreted programs that specify the cache coherence protocols. This specialization by supercompilation is performed by following the usual *unfold-fold cycle* controlled by the $\lhd \circ \preccurlyeq$-strategy described in Section 4.3. Note that this program specification includes both the function call and

[9] I. e., the iterated ordinary supercompilation, which does not use any intermediate interpretation, of the residual program produced by one indirect verification.

[10] Actually *False* is never encountered at all in any residual program generated by repeated launching the supercompiler SCP4 verifying the cache coherence protocol models considered in this paper. I.e., the property is simpler than the formulated one.

constructor application stacks, where the size of the first one is uniformly bounded on the input parameter while the second one is not.

We start off by unfolding the initial configuration $Int(\ (Call\ Main\ e.d),\ (Prog\ Synapse)\ )$, where the value of $e.d$ is unknown. The safety property will be proved if supercompilation is able to recognize all rules of the interpreted program model, containing the *False* identifier, as being unreachable from this initial configuration.

In our early work [27] we have given a formal model of the verification procedure above by supercompilation. Let a program model and its safety property be given as described above, i.e., a partial program predicate. Given an initial parameterized configuration of the partial predicate, it has been shown that the unfold-fold cycle may be seen as a series of proof attempts by structure induction over the program configurations encountered during supercompilation aiming at verification of the safety property. Here the initial configuration specifies the statement that we have to prove. There are too many program configurations generated by the unfold-fold cycle starting with the initial configuration given above and the self-interpreter configurations are very large. As a consequence it is not possible to consider all the configurations in details. We study the configurations' properties being relevant to the proof attempts and the method for reasoning on such properties.

5.1 On Meta-Reasoning

Let a program P_0 written in $\mathscr{L}$ and a function application $f(d_0)$, where $d_0 \in \mathbb{D}$ is its input data, be given. Let π_0 stand for expression $(Prog\ \ N_{P_0})$, where N_{P_0} is the program name.

The unfolding loop standing alone produces a computation path $C_0, C_1, \ldots, C_n, \ldots$ starting off $f(d_0)$. If $f(d_0)$ terminates then the path is finite $C_0, C_1, \ldots, C_k$. In such a case, for any $0 \leq i < k$ C_i is a configuration not containing parameters, while C_k is either a passive expression, if partial function f defined on the given input data, or the abnormal termination sign $\perp$ otherwise. If the application does not terminate then all C_i are non-parameterized configurations. The unfolding iterates function $step(\cdot)$ such that $step(C_i) = C_{i+1}$.

Now let us consider the following non-parameterized configuration $K_0 = Eval((\texttt{[]}) : \underline{f(\ d_0\ )}, \pi_0)$ of the self-interpreter. If $f(d_0)$ terminates then the loop unfolding the configuration K_0 results in the encoded passive configuration produced by the loop unfolding $f(d_0)$.

$$K_1 = step(K_0) = Eval(\ EvalCall(\ \underline{f},\ Eval(\ (\texttt{[]}) : \underline{d_0},\ \pi_0\ ),\ \pi_0\ ),\ \pi_0\ )\ \texttt{++}\ Eval(\ (\texttt{[]}) : \texttt{[]},\ \pi_0\ )$$

Expression K_1 is not a configuration. According to the strategy described in Section 2.3 the unfolding has to decompose expression K_1 in a sequence of configurations connected by the let-variables. This decomposition results in

$\{\ Eval(\ (\texttt{[]}) : \underline{d_0},\ \pi_0\ ),\ EvalCall(\ \underline{f},\ \bullet,\ \pi_0\ ),\ Eval(\ \bullet,\ \pi_0\ ),\ \bullet\ \}$,

$\texttt{let}\ e.x\ \texttt{=}\ \bullet\ \texttt{in}\ \{\ Eval(\ (\texttt{[]}) : \texttt{[]},\ \pi_0\ ),\ e.x\ \texttt{++}\ \bullet\ \}$, where $e.x$ is a fresh parameter.

Hence, considering modulo the arguments, the following holds. Given a function-call stack element f, this *step* maps the interpreted stack element to this segment of the interpreting function-call stack represented by the first configuration above, when this stack segment will be computed then its result is declared as a value of parameter $e.x$ and the last configuration will be unfolded. Note that (1) these two configurations split by the `let`-construct will be unfolded completely separately one from the other, i.e., the first configuration becomes the input of the unfolding loop, while the second configuration is postponed for a future unfolding call; (2) built-in function append `++` is not inserted in the stack at all, since it is treated by the supercompiler as a kind of a special constructor, which properties are known by the supercompiler handling this special constructor on the fly. The steps' sequence $step(K_i), \ldots, step(K_{j-1})$

between two consecutive applications $step(K_i)$, $step(K_j)$ of the first *Eval* rewriting rule unfolds the big-step of the interpreter, interpreting the regular step corresponding to the application of a rewriting rule of the f definition interpreted.

Given an expression *exp* to be interpreted by the interpreter, *exp* defines the current state of the P_0 function-call stack. Let C be the configuration representing this stack state (see Section 2.3). Let $f(exp_1)$ be the function application on the top of the stack. Then the current step $step(K_i)$ corresponding to the application of the first *Eval* rewriting rule maps f to the stack segment $Eval_1$, $EvalCall_2$, $Eval_3$ of the interpreter, considering modulo their arguments, and this stack segment becomes the leading segment of the interpreting function-call stack. The remainder of the interpreted stack is encoded in the arguments of $Eval_3$, $Eval_4$.

This remark allows us to follow the development of these two stacks in parallel. Given the following two parameterized configurations $f(exp_1)$ and $Eval((exp_{env}) : \underline{f(\ exp_1\ )}, \pi_0)$ we are going to unfold these configurations in parallel, step by step. The simpler logic of unfolding $f(exp_1)$ will provide hints on the logic unfolding $Eval((exp_{env}) : \underline{f(\ exp_1\ )}, \pi_0)$.

Now we consider the set of the configuration pairs that may be generated by the unfold-fold loop and are in the relation $\lhd \circ \preccurlyeq$.

5.2 Internal Properties of the Interpreter Big-Step

In this section we consider several properties of the configurations generated by the unfold-fold loop inside one big-step of the self-interpreter.

Thus in order to prove indirectly that the program model is safe, we start off by unfolding the following initial configuration $Int(\ (Call\ Main\ e.d),\ (Prog\ Synapse)\ )$, where the value of $e.d$ is unknown. Let π_s stand for $(Prog\ Synapse)$.

Consider any parameterized configuration C_b generated by the unfold-fold loop and initializing a big-step of the interpreter. Firstly, we assume that C_b is not generalized and no configuration was generalized by this loop before C_b. In such a case, C_b is of the following form

$$Eval(\ (env) : \underline{arg},\ \pi_s\ ),\ EvalCall(\ \underline{f},\ \bullet,\ \pi_s\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

where *arg* stands for the formal *syntactic* argument taken from the right-hand side of a rewriting rule where $\underline{f(\ arg\ )}$ originates from, $env ::= (\ var : val\ ) : env \mid \texttt{[]}$, $var ::= \underline{s.n} \mid \underline{e.n}$, and *val* stands for a partial known value of variable *var*. Since application $\underline{f(\ arg\ )}$ is on the top of the stack, argument *arg* includes no function application. As a consequence, the leading *Eval* application has only to look for variables and to call substitution *Subst* if a variable is encountered.

Thus, excluding all the transitive configurations encountered before the substitution, we consider the following configuration

$\{\ Subst(\ env,\ (Var\ \underline{var_{t.n}})\ ),\ \bullet\ \},\quad \texttt{let}\ e.x_1\ \texttt{=}\ \bullet\ \texttt{in}\ \{\ \{\ Eval(\ (env) : \underline{arg_1},\ \pi_s\ ),\ \bullet\ \},$
$\texttt{let}\ e.x_2\ \texttt{=}\ \bullet\ \texttt{in}\ \{\ EvalCall(\ \underline{f},\ e.x_1\ \texttt{++}\ e.x_2,\ \pi_s\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots\}\ \}$

where $e.x_1, e.x_2$ are fresh parameters, arg_1 stands for a part of *arg* above to be processed, and $\underline{var_{t.n}}$ denotes the type and the name of the variable encountered.

We turn now to the first configuration to be unfolded. All configurations unfolded, step by step, from the first configuration are transitive since *Subst* tests only types and names of the environment variables. Function *Subst* is tail-recursive and returns value *val* asked for.

We skip transforming these transitive configurations and continue with the next one.

$\{\ Eval(\ (env) : \underline{arg_1},\ \pi_s\ ),\ \bullet\ \},\ \texttt{let}\ e.x_2\ \texttt{=}\ \bullet\ \texttt{in}\ \{\ EvalCall(\ \underline{f},\ val\ \texttt{++}\ e.x_2,\ \pi_s\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots\}$

By our assumption above, the loop unfolding this first configuration never generates a function application. So the leading configuration proceeds to look for the variables in the same way shown above.

When *arg* is entirely processed and all variables occurring in *arg* are replaced with their partially known values from the environment, then the current configuration looks as follows.

$$EvalCall(\ \underline{f},\ arg_2,\ \pi_s\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

Here expression arg_2 is $\underline{arg}\theta$, were θ is the substitution defined by environment *env*. I. e., arg_2 may include parameters standing for unknown data, while $\underline{arg}$ does not.

Function *EvalCall* is defined by the only rewriting rule. So, any application of this function is one-step transitive. Recalling π_s, we turn to the next configuration.

$$Prog(\ Synapse\ ),\ LookFor(\ \underline{f},\ \bullet\ ),\ Matching(\ F,\ [\,],\ \bullet,\ arg_2\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

Prog(*Synapse*) returns the source code of the interpreted program *Synapse*, while the *LookFor* application returns the definition of the function called by the interpreter, using the known name $\underline{f}$. Skipping the corresponding transitive configurations, we have:

$$Matching(\ F,\ [\,],\ ((\underline{p_1}):\text{'='}:(\underline{exp_1})):\underline{def_{r_1}},\ arg_2\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

Here the third *Matching* argument is the f definition, where p_1, exp_1, def_{r_1} stand for the pattern, the right-hand side of the first rewriting rule of the definition, and the rest of this definition, respectively. This *Matching* application transitively initiates matching the parameterized data arg_2 against pattern $\underline{p_1}$ and calls another *Matching* application in the context of this pattern matching. This second *Matching* application is provided with the f definition rest and arg_2 for the case this pattern matching will fail. The next configuration is as follows.

$$(\checkmark)\quad Match(\ \underline{p_1},\ arg_2,\ (\,[\,]\,)\ ),\ Matching(\ \bullet,\ \underline{exp_1},\ \underline{def_{r_1}},\ arg_2\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

Remark 1 *By now all the configurations generated by the unfolding loop were transitive. The steps processing syntactic structure of the function application considered might meet constructor applications. These constructor applications are accumulated in the second EvalCall argument. The analysis above did not use any particular property of the interpreted program despite the fact that the source code of the interpreted program has been received and processed.*

Now we start to deal with function *Match* playing the main role in our analysis. In order to unfold this configuration, we have now to use some particular properties of the interpreted program.

Since for any pattern p_1 in program Synapse and any $v \in \mathcal{V}$ $\mu_v(p_1) < 2$ holds, Proposition 1 below implies that the unfold-fold loop never stops unfolding the configuration ✓ until the *Match* application on the top of the stack will be completely unfolded to several passive expressions, step by step. These expressions will appear on different possible computation paths starting at the configuration above. Skipping the steps unfolding the tree rooted in this stack-top configuration, we turn to the configurations that appear on the leaves of this tree. Each path starting at the top configuration leads to a configuration of one of the following two forms. These configurations are transitive.

$$Matching(\ (env_1),\ \underline{exp_1},\ \underline{def_{r_1}},\ arg_3\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

$$Matching(\ F,\ \underline{exp_1},\ \underline{def_{r_1}},\ arg_3\ ),\ Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

In the first case, the pattern matching did succeed and function *Matching* replaces the current function application with the right-hand side of the chosen rewriting rule, provided with the constructed environment. The big-step being considered has been finished. In order to launch the next big-step, interpreter *Int* has now to update the top of the interpreting function application stack.

In the second case, the pattern matching fails and function *Matching* once again calls *Match*, aiming to match the parameterized data against the pattern of the next rewriting rule of function f. The next configuration is of the form ✓ above, in which the third *Matching* argument value is decremented with the rewriting rule has been considered. If this value is empty and cannot be decremented, then, according to the language $\mathscr{L}$ semantics 2.1, we have the abnormal deadlock state and the interpreter work is interrupted.

Starting off from this configuration, the unfold-fold loop proceeds in the way shown above.

Proposition 1 *For any pattern p_0 such that for any $v \in \mathscr{V}$ $\mu_v(p_0) < 2$ and any parameterized passive expression d, the unfold-fold loop, starting off from configuration Match($\underline{p_0}$, d, ([])) and controlled by the $\triangleleft \circ \preccurlyeq$-strategy, results in a tree program*[11] *such that any non-transitive vertex in the tree is labeled by a configuration of the form Match($\underline{p_i}$, d_i, (env_i)), ... Given a path in the tree and any two configurations of the forms Match($\underline{p_i}$, d_i, (env_i)), ... and Match($\underline{p_j}$, d_j, (env_j)), ... belonging to the path, such that the second configuration is a descendant of the first one, then $\underline{p_j} \prec \underline{p_i}$ holds.*

Proof If all descendants of configuration *Match*($\underline{p_0}$, d, ([])) are transitive then the unfold-fold loop results in a tree being a root and this tree satisfies the property required. Now consider non-transitive descendants of *Match*($\underline{p_0}$, d, ([])) that may be generated by the unfold-fold loop before the first generalization or folding action happened. The patterns of the *PutV* rewriting rules never test unknown data, hence any application of *PutV* is transitive. Since for any $v \in \mathscr{V}$ relation $\mu_v(p_0) < 2$ holds, then function *Eq* will be never applied and the application of *CheckRepVar* is transitive. As a consequence, all the non-transitive descendants are of the forms *Match*($\underline{p_i}$, d_i, (env_i)),... .

Note that only the paths originated by applications of the 2-nd, 3-rd, 4-th *Match* rewriting rules may contain configurations of such forms.

Consider a configuration *Match*($\underline{p_i}$, d_i, (env_i)),... . Below M_i denotes such a configuration.

Since p_i is a constant, hence one-step unfolding this configuration by the 2-nd rewriting rule leads to configuration *PutVar*($\underline{s.n}$: d_s, (env_i)), *Match*($\underline{p_{i+1}}$, d_{i+1}, •),... such that $\underline{p_{i+1}}$ is a proper piece of $\underline{p_i}$, in which at least one constructor is removed. Hence, $\underline{p_{i+1}} \prec \underline{p_i}$ holds. Since the *PutVar* application is transitive, a number of unfolding steps lead transitively to *Match*($\underline{p_{i+1}}$, d_{i+1}, (env_{i+1})),... such that $\underline{p_{i+1}} \prec \underline{p_i}$.

One-step unfolding the configuration *Match*($\underline{p_i}$, d_i, (env_i)),... by the 3-rd rewriting rule leads to configuration *Match*($\underline{p_{i+1}}$, d_{i+1}, (env_i)), *Match*($\underline{p_{i+2}}$, d_{i+2}, •),... such that $\underline{p_{i+1}}$ and $\underline{p_{i+2}}$ are proper pieces of constant $\underline{p_i}$. Hence, $\underline{p_{i+1}} \prec \underline{p_i}$ and $\underline{p_{i+2}} \prec \underline{p_i}$ hold.

One-step unfolding the configuration *Match*($\underline{p_i}$, d_i, (env_i)),... by the 4-th rewriting rule leads to configuration *Match*($\underline{p_{i+1}}$, d_{i+1}, (env_i)), ... such that $\underline{p_{i+1}}$ is a proper piece of $\underline{p_i}$, in which at least one constructor is removed. Hence, $\underline{p_{i+1}} \prec \underline{p_i}$ holds.

Now consider any two configurations of the forms *Match*($\underline{p_i}$, d_i, (env_i)),... and *Match*($\underline{p_j}$, d_j, (env_j)),... such that the second configuration belongs to a path originating from the first one and is encountered before any generalization. Hence, $\underline{p_i}$ and $\underline{p_j}$ are constants.

Given a configuration C, the length of C, denoted by $ln_c(C)$, is the number of the *upper* function applications in C. (See Definition 2 above.)

If $ln_c(M_j) < ln_c(M_i)$ then $M_i \not\lhd M_j$ and the Turchin relation prevents M_j, M_i from generalization and M_j from any folding action. (See Section 4.3.)

[11] I.e., without any function application, excepting an entry point of this residual program.

If $ln_c(M_j) \geq ln_c(M_i)$ then we consider the first shortest configuration M_k in the path segment being considered. By definition of the stack, $Match(\ \underline{p_k},\ d_k,\ \bullet\ )$ is from M_i and $Match(\ \underline{p_j},\ d_j,\ (env_j)\ )$ is a descendent of $Match(\ \underline{p_k},\ d_k,\ (env_k)\ )$.

Since $Match(\ \underline{p_k},\ d_k,\ \bullet\ )$ took a part in computing M_j, it is the last function application of the stack prefix defined by the following Turchin relation $M_i \triangleleft M_j$, which holds. On the other hand, by the reasoning given above, for any p_{j_l} from the prefix of M_j, defined by *this* Turchin relation, $\underline{p_{j_l}} \prec \underline{p_k}$ holds, and, as a consequence, the following relation $Match(\ \underline{p_{j_l}},\ d_{j_l},\ (env_{j_l})\ ) \prec Match(\ \underline{p_k},\ d_k,\ \bullet\ )$ holds. According to Turchin's relation, this relation prevents M_j, M_i from generalization and M_j from any folding action. (See Section 4.3.)

The proposition has been proven. □

5.3 Dealing with the Interpreter Function-Application Stack

Firstly, assuming again that no generalization happened in the unfold-fold loop up to now, given a right-hand side exp_0 of a rewriting rule returned by function *Matching* as described in Section 5.2, the following configuration has to map, step by step, the segment of the interpreted function-application stack, that is defined by known $\underline{exp_0}$, on the top of the interpreting stack.

$$Eval(\ (env) : \underline{exp_0},\ \pi_s\ ),\ \ldots$$

According to the call-by-value semantics, function *Eval* looks for the function application, whose right bracket is the leftmost closing bracket, in completely known $\underline{exp_0}$. It moves from left to right along $\underline{exp_0}$, substitutes transitively the variables encountered, as shown in Section 5.2, pushes the interpreted function application in the interpreting stack, mapping it into an *EvalCall* application, whenever the interpreted application should be pushed in the interpreted stack. See Section 5.1 for the details. Finally the depth first *EvalCall* application starts the next big-step.

Since for any pattern p of program Synapse and any $v \in \mathcal{V}$ $\mu_v(p) < 2$ holds, hence, all applications of *Eq*, *ContEq* and *PutV* are transitive. This note together with Proposition 1 implies Proposition 2 below.

Let p_i stand for sub-patterns of a pattern of program Synapse, arg_i and *def* stand for partially known parameterized expressions and several, maybe zero, rewriting rules being a rest of a function definition of Synapse, respectively. Let *exp* stand for the right-hand side of a rewriting rule from this definition. $env ::= (\ var : val\ ) : env \mid []$, $var ::= \underline{s.n} \mid \underline{e.n}$, and *val* stands for a partially known value of variable *var*. Let Int_0 denote $Int(\ (Call\ Main\ e.d),\ \pi_s\ )$, where the value of $e.d$ is unknown.

Proposition 2 *Let the unfold-fold loop be controlled by the $\triangleleft \circ \preccurlyeq$-strategy. Let it start off from the initial configuration Int_0. Then the first generalized configuration generated by this loop, if any, will generalize two configurations of the following forms and any configuration folded, before this generalization, by a previous configuration is of the following form, where $n > 0$,*

(¶) $Match(\ \underline{p_1},\ arg_1,\ (env)\ ),\ \ldots,\ Match(\ \underline{p_n},\ arg_n,\ \bullet\ ),\ Matching(\ \bullet,\ \underline{exp},\ \underline{def},\ arg_0\ ),$

$$Eval(\ \bullet,\ \pi_s\ ),\ \ldots$$

Now consider any application of the form $Match(\ \underline{p_1},\ arg_1,\ (env)\ )$ staying on the stack top. Let $Match_1$ denote this application. Only the third rewriting rule of *Match* increases the stack. In this case the next state after the stack ¶ is of the form $Match_3$, $Match_2$ .... Then, by Proposition 1, along any path originating from $Match_2$ the application $Match_2$ is not replaced until application $Match_3$ will be completely unfolded. Hence, for any stack state of the form $f_i,\ \ldots,\ Match_2,\ \ldots$ on such a path, where f_i denotes any function application, the following relation $Match_2,\ \ldots \not\triangleleft\ f_i,\ \ldots,\ Match_2,\ \ldots$ holds. That proves the following corollary using the denotations given above.

Corollary 3 *Given a timed application* $Matching_{t_0}(\bullet, \underline{exp}, \underline{def}, arg_0)$*, any two timed configurations of the form* $\ldots, Matching_{t_0}(\bullet, \underline{exp}, \underline{def}, arg_0), Eval(\bullet, \pi_s), \ldots$ *can neither be generalized nor folded one by the other.*

Since any application $Matching(\ldots, \underline{exp}, \underline{def},)$ decreases, step by step, the list $\underline{def}$ of the rewriting rules, the following corollary holds.

Corollary 4 *Given a big-step of Int, a timed pair* $\underline{exp_{t_1}}, \underline{def_{t_1}}$[12]*, and a timed application* $Matching_{\tau_0}(\bullet, \underline{exp_{t_1}}, \underline{def_{t_1}}, arg_0)$ *inside this big-step, then any two timed configurations of the form* $\ldots, Matching_{\tau_i}(\bullet, \underline{exp_{t_i}}, \underline{def_{t_i}}, arg_0), Eval(\bullet, \pi_s), \ldots$ *can neither be generalized nor folded one by the other.*

Given a function definition F and a rewriting rule r of the definition, let $exp_{F,r}$ stand for the right-hand side of r, while $def_{F,r}$ stand for the rest of this definition rules following the rule r. The following is a simple syntactic property of the Synapse program model given in Section 3.

(5.1) For any $F_1, F_2 \in \{Main, Loop, Event, Append, Test\}$ and for any two *distinct* rewriting rules r_1, r_2 of F_1, F_2, respectively, $(exp_{F_1,r_1}, def_{F_1,r_1}) \neq (exp_{F_2,r_2}, def_{F_2,r_2})$ holds.

That together with Corollaries 3, 4 imply:

Proposition 3 *Given two configurations* C_1 *and* C_2 *of the form* ¶ *to be generalized or folded one by the other. Then (1)* C_1 *and* C_2 *cannot belong to the same big-step of Int; (2) there are two functions* F_1*,* F_2 *of a cache coherence protocol model from the series mentioned in Section 6 (and specified in the way shown in Section 3) and rewriting rules* r_1, r_2 *of* F_1, F_2*, respectively, such that* $(exp_{F_1,r_1}, def_{F_1,r_1}) = (exp_{F_2,r_2}, def_{F_2,r_2})$*, where functions* F_1*,* F_2 *and rules* r_1, r_2 *may coincide, respectively.*

Remark 2 *Proposition 3 depends on Property 5.1. Nevertheless, the restriction imposed by 5.1 is very weak and the most of programs written in* $\mathscr{L}$ *satisfy 5.1. It can easily be overcome by providing the interpreting function Matching with an additional argument that is the interpreted function name. The second statement of this proposition is well and crucial for the expectation of removing the interpretation overheads. Despite the fact that the reasoning above follows the Synapse program model, it can be applied to any protocol model used in our experiments described in this paper.*

We conclude that any configuration encountered by the loop unfolding the given big-step is neither generalized with another configuration generated in unfolding this *big-step nor folded by such a configuration.*

5.4 On Generalizing the Interpreter Configurations

Unfortunately, the paper size limit is exhausted to four pages before this page. We hope to get the reader endurance and the reviewers' forbearance and continue briefly this long story.

The unfold-fold loop processes the paths originating from the initial configuration Int_0, following the corresponding interpreted paths. The latter ones are processed according to the order of the rewriting rules in the Synapse function definitions.

The initial configuration Int_0 is unfolded according to function definition of *Main* of the interpreted program. Application of this function leads transitively to the following function application

[12] Here we use the denotation given above and the timed expressions, which are defined in the same way as the timed applications 4.2.

$$Loop(\ (time) : (Invalid\ I\ is) : (Dirty\ ) : (Valid\ )\ )$$

The interpreter has to match this call against the left-hand side of the first rewriting rule of the *Loop* definition. The first corresponding pattern is: $(\ [\,]\ ) : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)$.

The pattern matching processes the pattern and argument from the left to the right, by means of function *Match*. The known part of the tree structure is mapped into the interpreting stack by the third rule of *Match*. The number of *Match* applications in the stack is increased by this rewriting rule.

In the given context of specialization the values of *time* and *is* are unknown. Hence, the prefix of this stack that is responsible for matching the argument constant structure on the left-hand side of *time* will transitively disappear and the unfolding loop will stop at the configuration of the following form
$Match(\ [\,],\ time,\ (\ [\,]\ )\ ),\ Match(\ \underline{p_2}\ ,\ arg_2,\ \bullet\ ),\ Matching(\ \bullet,\ \underline{exp},\ \underline{def}\ ,\ arg_0\ ),\ Eval(\ \bullet\ ,\ \pi_s\ ),\ \bullet.$
Here the leading *Match* application meets the unknown data *time* and has to match it against `[]` given in the first argument. The environment in the third argument is empty since no variable was still assigned up to now. The second *Match* application is responsible for matching the suspended part of the input data arg_2 against the rest $\underline{p_2}$ of the pattern. I. e., arg_2 equals $(Invalid\ I\ is) : (Dirty\ ) : (Valid\ )$ and p_2 equals $(Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)$.

Now we note that the arguments of all applications of the recursive *Loop* and *Append* functions have exactly the same constant prefix as considered above. That leads to the following proposition.

Proposition 4 *Let the unfold-fold loop be controlled by the $\lhd \circ \preccurlyeq$-strategy. Let it start off from the initial configuration Int_0. Then the first generalized configuration generated by this loop, if any, will generalize two configurations of the following forms and any configuration folded, before this generalization, by a previous configuration is of the following form*
$Match(\ [\,],\ arg_1,\ (\ [\,]\ )\ ),\ Match(\ \underline{p_2}\ ,\ arg_2,\ \bullet\ ),\ Matching(\ \bullet,\ \underline{exp},\ \underline{def}\ ,\ arg_0\ ),\ Eval(\ \bullet\ ,\ \pi_s\ ),\ \ldots$

In the given context of specialization the Turchin relation plays a crucial role in preventing the encountered configurations from generalization (see Proposition 1). It never forces decomposing the generalized configurations as might do, in general (see Section 4.2).

The first generalization has happened on the path unfolding the interpreted *Loop* along the recursive branch following the iteration of the protocol event *rm*. No configuration was folded before this generalization. Only the number of *Valid* items taking a part in the Synapse protocol was generalized. It serves as an accumulator and occurs twice in the generalized configuration C_{g_1}. So this generalization does not violate the safety property. Actually, this parameter might be introduced still in the initial configuration Int_0 and the corresponding initial configuration *generalized in advance* may be seen as a more meaningful formulation of the verification task.

No other generalization of the configurations generated by the recursion defined in function *Loop* has happened.

The first folding action was done in the same context as the first generalization done. The only distinction consists in another action: a generated configuration was folded by C_{g_1}. Thus there is the only variable to be nontrivially substituted by this folding. The variable is the accumulating parameter introduced by this generalization.

The second folding action was performed by the generalized configuration C_{g_1} itself. The folded configuration ends a branch following the iteration of the protocol event *rm*. There are two variables to be nontrivially substituted by this folding. One of the variables is the accumulating parameter mentioned above, while the other one is substituted with constant `[]`.

The third folding action was done by a regular configuration C_1. Configuration C_1 is a descendant of C_{g_1}. The folded configuration ends a branch following the iteration of the protocol event *wm*. The corresponding substitution is completely trivial, i. e., the folded and folding configurations coincide modulo variables names.

The sub-tree rooted in C_1 is completely folded. Configuration C_1 is declared as an entry point of a residual function definition. The fist rewriting rule of this definition is an exit from a recursion returning the value *True*, while the second and third branches refer to C_{g_1} and C_1, respectively.

The second generalization is forced by the interpreted *Append* that is directly defined. It has happened along the recursive branch following the iteration of the protocol event *wh2*. The generalized configurations include several suspended substitutions to be performed in completely known expressions. Thus these substitutions may be done before this generalization if one might reveal formal reasons for such treatment of the suspended substitutions. These suspended substitutions form a stack of the *Eval* applications aiming at executing them. This stack is still presented with delayed syntax composition, i. e., the *Eval* applications are not still pushed in the explicit function application stack of the corresponding configuration. The *Eval* application stack was generalized. A part of its calls were completely generalized by a parameter generalizing a rest of this stack, while the other calls were partially generalized. This second part of the generalization is very bad. The corresponding generalizing substitution contains pieces of the interpreted program model. The pieces are names and types of several variables. Fortunately, this generalization does not violate the safety property to be verified. Neither *Dirty* nor *Valid* counters were generalized by this generalization. Below C_{g_2} stands for the configuration constructed by the second generalization.

The fourth folded configuration is a descendent of configuration C_{g_2}. It was folded by the regular configuration C_1 used in the third folding. The corresponding substitution is quite trivial. There are two parameters in both the folded and folding configurations. The folding substitution renames the parameter corresponding to the initial unknown data *e.time* and replaces the parameter, standing for the unknown value of the *Invalid* counter, with concatenation of completely unknown data early split into three parts.

The fifth folded configuration is a descendent of configuration C_{g_2} and it is folded by C_{g_2} as well. There are thirteen parameters occurring in C_{g_2}. Fife parameters are substituted with expressions containing names and/or types of some variables from the Synapse program model. One parameter is replaced with an expression including an application *Eval* that presents a generalized environment of this program model.

The sub-tree rooted in C_{g_2} is completely folded. Configuration C_{g_2} is declared as an entry point of a residual function definition. The fist rewriting rule of this definition refers to C_1, while the seconds one refers to C_{g_2}, i. e., to itself.

The sixth folding action was done by the regular configuration C_1 used above. The folding substitution increases counter *Invalid* and renames the variable corresponding to stream *e.time*.

The third generalization is forced by the interpreted *Append* that is directly defined. It has happened along the recursive branch following the iteration of the protocol event *wm*. The corresponding generalized configuration C_{g_3} has 14 parameters and the badness similar to C_{g_2}.

The seventh folding action was done in a similar context as the fourth one done. The only distinction consists in increasing the *Invalid* counter by a constant.

The eighth folded configuration is a descendent of configuration C_{g_3} and it is folded by C_{g_3} as well. The corresponding folding substitution has the badness similar to the fifth folding substitution.

The sub-tree rooted in C_{g_3} is completely folded. Configuration C_{g_3} is declared as an entry point of a residual function definition. The fist rewriting rule of this definition refers to C_1, while the seconds one refers to C_{g_3}, i. e., to itself.

That in its turn allows us to fold the sub-tree rooted in the first generalized configuration C_{g_1}. Configuration C_{g_1} is declared as an entry point of a residual function definition. The fist rewriting rule of this definition is an exit from a recursion returning the value *True*. The second rewriting rule calls this entry point C_{g_1}, while the third and fourth branches refer to C_{g_2} and C_{g_3}, respectively.

The last folded configuration belongs to the last branch originating from the initial configuration. It is folded by the regular configuration C_1, which belongs to a parallel path and was used in the third folding.

The entire specialized tree has been completely folded and the safety property has been proven. Note that no generalized configuration was refused by the generalized configurations that were generated later than the former one.

The crucial configurations of the unfolding history leading to verification of this program model are given in Appendix 7.4 and are self-sufficient. The residual program, generated by supercompilation of the residual program resulted in specializing the self-interpreter with respect to the Synapse N+1 program model, is given in Section 7.2.

6 Conclusion

We have shown that a combination of the verification via supercompilation method and the second Futamura projection allows us to perform verification of the program being interpreted. We discussed the crucial steps of the supercompliation process involved in a verification of a parameterized cache coherence protocol used as a case study. In the same way we were able to verify all cache coherence protocols from [6, 7], including MSI, MOSI, MESI, MOESI, Illinois University, Berkley RISC, DEC Firefly, IEEE Futurebus+, Xerox PARC Dragon, specified in the interpreted language $\mathscr{L}$. Furthermore, we were able to verify the same protocols specified in the language WHILE [16]. The complexity of involved processes is huge and further research is required for their better understanding.

Our experimental results show that Turchin's supercompilation is able to verify rather complicated program models of non-deterministic parameterized computing systems. The corresponding models used in our experiments are constructed on the base of the well known series of the cache coherence protocols mentioned above. So *they might be new challenges to be verified by program transformation rather than an approach for verifying the protocols themselves*. This protocol series was early verified by Delzanno [6] and Esparza et al. [8] in abstract terms of equality and inequality constraints. Using unfold-fold program transformation tools this protocol series was early verified by the supercompiler SCP4 [24, 25, 27, 26] in terms of a functional programming language, several of these protocols were verified in terms of logical programming [35, 10]. One may consider the indirect protocol models presented in this paper as a new collection of tests developing the state-of-the-art unfold-fold program transformation.

The intermediate interpreter considered in this paper specifies the operational semantics of a *Turing-complete* language $\mathscr{L}$. We have proved several statements on properties of the configurations generated by the unfold-fold cycle in the process specializing *Int* with respect to the cache coherence protocols specified as shown in Section 3. The main of them is Proposition 1. Some of these properties *do not depend on specific protocols* from the considered series, i. e., they hold for any protocol specified in the way shown in Section 3. That allows us to reason, *in a uniform way*, on huge amount of complicated

configurations. Note that the programs specifying the protocols include both the function call and constructor application stacks, where the size of the first one is uniformly bounded on the input parameter while the second one is not. Unlike system VeriMAP [3] our indirect verification method involves no post-specialization unfold-fold.

As a future work, we would like to address the issue of the description of suitable properties of interpreters to which our uniform reasonings demonstrated in this paper might be applied.

Acknowledgements: We would like to thank Antonina Nepeivoda and several reviewers for helping to improve this work.

References

[1] A. Ahmed, A.P. Lisitsa, and A.P. Nemytykh. Cryptographic protocol verification via supercompilation (A case study). In *VPT 2013*, volume 16 of *EPiC Series*, pages 16–29. EasyChair, 2013.

[2] De E. Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Science of Computer Programming*, 95 / Selected and extended papers from Partial Evaluation and Program Manipulation 2013(Part 2):149–175, 2014. doi:10.1016/j.scico.2014.05.017.

[3] De E. Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verimap: A tool for verifying programs through transformations. In *Proc. of (TACAS 2014)*, volume 8413 of *LNCS*, pages 568–574. Springer, 2014. doi:10.1007/978-3-642-54862-8_47.

[4] De E. Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming*, 15 / 31st International Conference on Logic Programming(4–5):635–650, July 2015. doi:10.1017/S1471068415000289.

[5] G. Delzanno. Automatic verification of cache coherence protocols via infinite-state constraint-based model checking. [online]. `http://www.disi.unige.it/person/DelzannoG/protocol.html`.

[6] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *The Proc. of the 12th Int. Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 53–68, 2000.

[7] G. Delzanno. Verification of consistency protocols via infinite-state symbolic model checking, A case study: the IEEE Futurebus+ protocol. In *The Proc. of FORTE/PSTV*, pages 171–186. Springer US, 2000.

[8] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, pages 352–359. IEEE, 1999.

[9] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *the Proc. of VCL01*, volume DSSE-TR-2001-3 of *Tech. Rep.*, pages 85–96, UK, 2001. University of Southampton.

[10] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. In W. Faber and N. Leone, editors, *Theory and Practice of Logic Programming*, volume 13 / Special Issue 02 (25th GULP annual conference), pages 175–199. Cambridge University Press, March 2013.

[11] Y. Futamura. Partial evaluation of computing process an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[12] J. P. Gallagher, J. C. Peralta, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In B. Demoen and V. Lifschitz, editors, *SAS 1998*, volume 1503 of *LNCS*, pages 246–261. Springer, 1998. doi:10.1007/978-3-540-27775-0_3.

[13] G. W. Hamilton. Verifying temporal properties of reactive systems by transformation. *Electronic Proceedings in Theoretical Computer Science*, 199:33–49, 2015. doi:10.4204/EPTCS.199.3.

[14] G. W. Hamilton. Generating counterexamples for model checking by transformation. *Electronic Proceedings in Theoretical Computer Science*, 216:65–82, 2016. doi:10.4204/EPTCS.216.4.

[15] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 2(7):326–336, 1952.

[16] N. D. Jones. *Computability and Complexity from a Programming Perspective*. The MIT Press, 2000.

[17] N. D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52:307–339, 2004. doi:10.1016/j.scico.2004.03.010.

[18] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher order call-by-value language. *ACM SIGPLAN Notices*, 44(1):277–288, 2009. doi:10.1145/1480881.1480916.

[19] A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In *the Proc. of PSI'11*, volume 7162 of *LNCS*, pages 193–209, 2012. doi:10.1007/978-3-642-29709-0_18.

[20] I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Technical Report 62, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[21] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Society*, 95:210–225, March 1960. doi:10.1090/S0002-9947-1960-0111704-1.

[22] H. Lehmann and M. Leuschel. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce. In *Proceedings of LOPSTR03*, volume 3018 of *LNCS*, pages 1–19, 2004. doi:10.1007/978-3-540-25938-1_1.

[23] M. Leuschel. On the power of homeomorphic embedding for online termination. In *Proc. of the SAS'98*, volume 1503 of *LNCS*, pages 230–245, 1998.

[24] A. P. Lisitsa and A. P. Nemytykh. A note on specialization of interpreters. In *The 2-nd International Symposium on Computer Science in Russia (CSR-2007)*, volume 4649 of *LNCS*, pages 237–248, 2007.

[25] A. P. Lisitsa and A. P. Nemytykh. Verification as parameterized testing (Experiments with the SCP4 supercompiler). *Programmirovanie, (In Russian)*, 1:22–34, 2007. English translation in *J. Programming and Computer Software*, Vol. **33**, No.1, pp: 14–23, 2007.

[26] A. P. Lisitsa and A. P. Nemytykh. Experiments on verification via supercompilation. [online], 2007–2009. `http://refal.botik.ru/protocols/`.

[27] A. P. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *International Journal of Foundations of Computer Science*, 19(4):953–970, August 2008.

[28] A. P. Nemytykh. The supercompiler SCP4: General structure. (extended abstract). In *the Proc. of PSI'03*, volume 2890 of *LNCS*, pages 162–170, 2003. doi:10.1007/978-3-540-39866-0_18.

[29] A. P. Nemytykh. *The Supercompiler SCP4: General Structure*. URSS, Moscow, 2007. (Book in Russian).

[30] A. P. Nemytykh, V. A. Pinchuk, and V. F. Turchin. A self-applicable supercompiler. In *PEPM'96*, volume 1110 of *LNCS*, pages 322–337. Springer-Verlag, 1996.

[31] A. P. Nemytykh and V. F. Turchin. The supercompiler SCP4: Sources, on-line demonstration. [online], 2000. `http://www.botik.ru/pub/local/scp/refal5/`.

[32] Antonina Nepeivoda. Ping-pong protocols as prefix grammars and Turchin relation. In *VPT 2013*, volume 16 of *EPiC Series*, pages 74–87. EasyChair, 2013.

[33] Antonina Nepeivoda. Turchin's relation and loop approximation in program analysis. *Proceedings on the Functional Language Refal (in Russian)*, 1:170–192, 2014. ISBN:978-5-9905410-1-6, available at http://refal.botik.ru/library/refal2014_issue-I.pdf.

[34] Antonina Nepeivoda. Turchin's relation for call-by-name computations: A formal approach. *Electronic Proceedings in Theoretical Computer Science*, 216:137–159, 2016. doi:10.4204/EPTCS.216.8.

[35] A. Roychoudhury and I. V. Ramakrishnan. Inductively verifying invariant properties of parameterized systems. *Automated Software Engineering*, 11(2):101–139, 2004.

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

[37] V. F. Turchin. The language Refal – the theory of compilation and metasystem analysis. Technical Report 20, Courant Institute of Mathematical Sciences, New York University, February 1980.

[38] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986. doi:10.1145/5956.5957.

[39] V. F. Turchin. The algorithm of generalization in the supercompiler. In *Proc. of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland Publishing Co., 1988.

[40] V. F. Turchin. *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts, 1989. Electronic version: `http://www.botik.ru/pub/local/scp/refal5/`, 2000.

[41] V. F. Turchin, D. V. Turchin, A. P. Konyshev, and A. P. Nemytykh. Refal-5: Sources, executable modules. [online], 2000. `http://www.botik.ru/pub/local/scp/refal5/`.

[42] L. van Begin. The BABYLON project: A tool for specification and verification of parameterized systems to benchmark infinite-state model checkers. [online]. `http://www.ulb.ac.be/di/ssd/lvbegin/CST/`.

7 Appendix

7.1 General Structure of the Supercompiler SCP4

Input: (i) A list *Tasks* of parameterized entry configurations, (ii) a source program, and (iii) strategies for performing the transformation.
Output: A set of *residual* function definitions.

```
residual := [] ;                                   /* Initialization.          */
do {
    current-task := head( Tasks ) ;
    unfolding ;                                    /* This actions may add     */
    folding & generalization ;                     /* configurations to Tasks. */
    if ( the current-task has been completely folded )
     {  global-analysis ;
        specialization with respect to the global properties ;
        propagation of the global information over Tasks ;
        if ( the current-task is not empty )
         { residual := residual ∪ {current-task}; }
        Tasks := tail( Tasks ) ;
     } ;
} while ( there is a configuration in Tasks )
dead-code-analysis ;
```

Figure 3: General Structure of the Supercompiler SCP4

7.2 Residual Program

The residual program, generated by supercompilation of the residual program resulted in specializing the self-interpreter 1 with respect to the Synapse N+1 program model 2, is given in Figure 4, where, for sake of readability, the variables' names and the entry function name were renamed.

The following three residual functions *F12*, *F31*, *F46* are specialized versions of the recursion defined in function *Loop*, while the two residual functions *F64* and *F100* are reflection of the recursion defined by function *Append* directly specified in the Synapse N+1 program.

Any right-hand side of the rewriting rules generated by the first supercompilation, which are not shown here, includes no identifier *False*. Hence, the Synapse N+1 program model already was successfully verified by the first application of the supercompiler SCP4, but the corresponding residual models includes syntactic pieces of the second rewiring rule of function *Append*. The second application of this supercompiler removes these syntactic pieces.

See Section 7.4 for some details of the first supercompilation application.

$IntRes(\ ('*'):('*':e.is)\ ) \Rightarrow True;$
$IntRes(\ ('*':rm:e.time):('*':e.is)\ ) \Rightarrow F12(\ (e.time):e.is\ );$
$IntRes(\ ('*':wm):('*':e.is)\ ) \Rightarrow True;$
$IntRes(\ ('*':wm:rm:e.time):('*':I:e.is)\ ) \Rightarrow F12(\ (e.time):I:e.is\ );$
$IntRes(\ ('*':wm:wm:e.time):('*':I:e.is)\ ) \Rightarrow F46(\ (e.time):e.is\ );$

$F12(\ ([]):e.is2\ ) \Rightarrow True;$
$F12(\ (rm:e.time):I:e.is2\ ) \Rightarrow F31(\ ([]):(e.time):e.is2\ );$
$F12(\ (wh2):e.is2\ ) \Rightarrow True;$
$F12(\ (wh2:rm:e.time):I:e.is2\ ) \Rightarrow F12(\ (e.time):I:e.is2\ );$
$F12(\ (wh2:wm:e.time):I:e.is2\ ) \Rightarrow F46(\ (e.time):e.is2\ );$
$F12(\ (wm:e.time):I:e.is2\ ) \Rightarrow F46(\ (e.time):e.is2\ );$

$F31(\ (e.is3):([]):e.is2\ ) \Rightarrow True;$
$F31(\ (e.is3):(rm:e.time):I:e.is2\ ) \Rightarrow F31(\ (I:e.is3):(e.time):e.is2\ );$
$F31(\ (e.is3):(wh2:e.time):e.is2\ ) \Rightarrow F64(\ (e.is3):(e.is2):(e.time)\ );$
$F31(\ (e.is3):(wm:e.time):I:e.is2\ ) \Rightarrow F100(\ (e.is3):(e.is2):(e.time)\ );$

$F46(\ ([]):e.is2\ ) \Rightarrow True;$
$F46(\ (rm:e.time):e.is2\ ) \Rightarrow F12(\ (e.time):I:e.is2\ );$
$F46(\ (wm:e.time):e.is2\ ) \Rightarrow F46(\ (e.time):e.is2\ );$

$F64(\ ([]):(e.ys2):(e.time):e.ys3\ ) \Rightarrow F46(\ (e.time):e.ys3\ \texttt{++}\ e.ys2\ );$
$F64(\ (s.x1:e.xs2):(e.ys2):(e.time):e.ys3\ ) \Rightarrow F64(\ (e.xs2):(e.ys2):(e.time):e.ys3\ \texttt{++}\ s.x1\ );$

$F100(\ ([]):(e.ys2):(e.time):e.ys3\ ) \Rightarrow F46(\ (e.time):I:e.ys3\ \texttt{++}\ e.ys2\ );$
$F100(\ (s.x1:e.xs2):(e.ys2):(e.time):e.ys3\ ) \Rightarrow F100(\ (e.xs2):(e.ys2):(e.time):e.ys3\ \texttt{++}\ s.x1\ );$

Figure 4: Residual Model of the Synapse N+1 Cache Coherence Protocol

7.3 Non-Transitivity of Turchin's Relation

The following example shows that Turchin's relation 4.2 is not transitive. The idea of the example is borrowed from [33].

$[R_0]$ $Main(\ e.xs\ ) \Rightarrow C(\ Fb(\ Fab(\ e.xs\ )\ )\ );$

$[R_1]$ $Fab(\ \texttt{[]}\ ) \Rightarrow \texttt{'c'};$
$[R_2]$ $Fab(\ \texttt{'a'}:e.xs\ ) \Rightarrow \texttt{'b'}:Fab(\ e.xs\ );$
$[R_3]$ $Fab(\ e.xs\ ) \Rightarrow Fc(\ Fab(\ e.xs\ )\ );$

$[R_4]$ $Fc(\ \texttt{[]}\ ) \Rightarrow \texttt{[]};$
$[R_5]$ $Fc(\ \texttt{'c'}:e.xs\ ) \Rightarrow \texttt{'d'}:\texttt{'d'};$
$[R_6]$ $Fc(\ s.y : e.xs\ ) \Rightarrow s.y : Fc(\ e.xs\ );$

$\ldots$ $C(\ \ldots\ ) \Rightarrow \ldots;$
$\ldots$ $\ldots\ldots\ldots\ldots\ldots$

$[R_7]$ $F(\ \texttt{[]}\ ) \Rightarrow \texttt{[]};$
$[R_8]$ $F(\ e.xs\ ) \Rightarrow Fc(\ Fab(\ e.xs\ )\ );$

$[R_9]$ $Fb(\ \texttt{[]}\ ) \Rightarrow \texttt{[]};$
$[R_{10}]$ $Fb(\ \texttt{'b'}:e.xs\ ) \Rightarrow \texttt{'b'}:e.xs;$
$[R_{11}]$ $Fb(\ s.y : e.xs\ ) \Rightarrow E(\ s.y : Fb(\ F(\ e.xs\ )\ )\ );$

$\ldots$ $E(\ \ldots\ ) \Rightarrow \ldots;$
$\ldots$ $\ldots\ldots\ldots\ldots\ldots$

Consider the path starting at initial configuration *Main* and following the sequence of the rewriting rules $[R_0],[R_3],[R_1],[R_5],[R_{11}],[R_8]$:

[1]: $Main_0(\ e.xs\ ) \xrightarrow{R_0}$ [2]: $Fab_3(\ e.xs\ ), Fb_2(\ \bullet\ ), C_1(\ \bullet\ ) \xrightarrow{R_3}$ [3]: $Fab_5(\ e.xs\ ), Fc_4(\ \bullet\ ), Fb_2(\ \bullet\ ), C_1(\ \bullet\ )$
$\xrightarrow{R_1}$ [4]: $Fc_4(\ \texttt{'c'}\ ), Fb_2(\ \bullet\ ), C_1(\ \bullet\ ) \xrightarrow{R_5}$ [5]: $Fb_2(\ \texttt{'d'}:\texttt{'d'}\ ), C_1(\ \bullet\ )$
$\xrightarrow{R_{11}}$ [6]: $F_8(\ \texttt{'d'}\ ), Fb_7(\ \bullet\ ), E_6(\ \texttt{'d'}:\bullet\ ), C_1(\ \bullet\ )$
$\xrightarrow{R_8}$ [7]: $Fab_{10}(\ \texttt{'d'}\ ), Fc_9(\ \bullet\ ), Fb_7(\ \bullet\ ), E_6(\ \texttt{'d'}:\bullet\ ), C_1(\ \bullet\ )$

Here the pairs [2], [3] and [3], [7] of the timed configurations are in Turchin's relation:

- [2] $\lhd$ [3] holds, where the common context is $Fb_2(\ \bullet\ ), C_1(\ \bullet\ )$ and the prefixes are $Fab_3(\ e.xs\ )$ and $Fab_5(\ e.xs\ )$, respectively;
- [3] $\lhd$ [7] hold, where the common context is $C_1(\ \bullet\ )$ and the prefixes are $Fab_5(\ e.xs\ ), Fc_4(\ \bullet\ ), Fb_2(\ \bullet\ )$ and $Fab_{10}(\ \texttt{'d'}\ ), Fc_9(\ \bullet\ ), Fb_7(\ \bullet\ )$, respectively.

While [2] $\lhd$ [7] does not hold.

7.4 Generalized and Folded Configurations

7.4.1 Generalization

1. *Gener-1*: *Loop*. The first generalization has happened on the path unfolding the interpreted *Loop* along the recursive branch following the iteration of the protocol event *rm*. No configuration was folded before this generalization. The boxed pieces of the configurations below stand for the generalized parts. Only the number of *Valid* items taking a part in the Synapse protocol was generalized. It serves as an accumulator and occurs twice in the generalized configuration. So this generalization does not violate the safety property.

   The previous configuration:
   $Match(\ \texttt{[]},\ e.101,\ (\texttt{[]})\ ),$
   $Match(\ \underline{(Invalid\ e.is):(Dirty\ e.ds):(Valid\ e.vs)},\ \underline{(Invalid\ e.102):(Dirty\ ):(Valid\ I\ \boxed{\texttt{[]}}\ )},\ \bullet\ ),$
   $Matching(\ \bullet,\ \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )}\ ,$
   $(\ \underline{Loop(\ (s.t : e.time) : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )}$
   $\underline{\Rightarrow Loop(\ (e.time) : Event(\ s.t : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )\ );}$

$$
\begin{array}{l}
\quad), \underline{(e.101) : (Invalid\ \ e.102) : (Dirty) : (Valid\ \ I\ \boxed{[]})} \\
), \\
Eval(\ \bullet, (Prog\ Synapse)\ )
\end{array}
$$

The current configuration:

$$
\begin{array}{l}
Match(\ [], e.101, ([])\ ), \\
Match(\ \underline{(Invalid\ e.is):(Dirty\ e.ds):(Valid\ e.vs)}, \underline{(Invalid\ e.102):(Dirty\ ):(Valid\ I\ \boxed{I})}, \bullet\ ), \\
Matching(\ \bullet, \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )}\ , \\
\quad (\ \underline{Loop(\ (s.t : e.time) : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )} \\
\qquad\qquad \underline{\Rightarrow Loop(\ (e.time) : Event(\ s.t : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )\ );} \\
\quad), \underline{(e.101) : (Invalid\ \ e.102) : (Dirty) : (Valid\ \ I\ \boxed{I})} \\
), \\
Eval(\ \bullet, (Prog\ Synapse)\ )
\end{array}
$$

The generalized configuration:

$$
\begin{array}{l}
Match(\ [], e.101, ([])\ ), \\
Match(\ \underline{(Invalid\ e.is):(Dirty\ e.ds):(Valid\ e.vs)}, \underline{(Invalid\ e.102):(Dirty\ ):(Valid\ I\ \boxed{e.138}\ )}, \bullet\ ), \\
Matching(\ \bullet, \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )}\ , \\
\quad (\ \underline{Loop(\ (s.t : e.time) : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )} \\
\qquad\qquad \underline{\Rightarrow Loop(\ (e.time) : Event(\ s.t : (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )\ );} \\
\quad), \underline{(e.101) : (Invalid\ \ e.102) : (Dirty) : (Valid\ \ I\ \ \boxed{e.138}\ )} \\
), \\
Eval(\ \bullet, (Prog\ Synapse)\ )
\end{array}
$$

2. *Gener-2*: *Append*

The second generalization is forced by the interpreted *Append* that is directly defined. It has happened along the recursive branch following the iteration of the protocol event *wh2*. The generalized configurations include several suspended substitutions to be performed in completely known expressions. Thus these substitutions may be done before this generalization if one might reveal formal reasons for such treatment of the suspended substitutions. These suspended substitutions form a stack of the *Eval* applications aiming at executing them. This stack is still presented with delayed syntax composition, i. e., the *Eval* applications are not still pushed in the explicit function application stack of the corresponding configuration. The *Eval* application stack was generalized. A part of its calls were completely generalized by a parameter generalizing a rest of this stack, while the other calls were partially generalized. This second part of the generalization is very bad. The corresponding generalizing substitution contains pieces of the interpreted program model. The pieces are names and types of several variables. Fortunately, this generalization does not violate the safety property to be verified. Neither *Dirty* nor *Valid* counters were generalized by this generalization. Below C_{g_2} stands for the configuration constructed by the second generalization.

The cause leading to the bad generalization is a wrong choice of the sequence of the steps generalizing the two configurations. Since constructor `++` is associative some sub-expressions of these configurations may be processed from both the left and right sides of the sub-expressions. The supercompiler SCP4 makes a bad choice in the situation described above. There are simple syntactic tricks allowing us overcome this problems. Nevertheless, the indirect specification of the Synapse N+1 protocol can be verified without using the tricks.

The previous configuration:

Match([], *e.138*, ([])),
Match((*e.ys*), (*e.137*), •),
Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.138*) : (*e.137*)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (*e.136*) : (*Invalid* [] ++ • ++
Eval(((*e.vs* : *e.138*) : (*e*.*is* : *e.137*) : (*e.ds* : [])) : [], (*Prog Synapse*))
++ []
) :
Eval(((*e.vs* : *e.138*) : (*e.is* : *e.137*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *e.138*) : (*e.is* : *e.137*) : (*s.t* : *wh2*) : (*e.time* : *e.136*) : (*e.ds* : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

The current configuration:

Match([], *e.138*, ([])),
Match((*e.ys*), (*e.137*), •),
Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.138*) : (*e.137*)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (*e.136*) : (*Invalid* *s.195* • ++
Eval(((*e.ys* : *e.137*) : (*s* . *x* : *s.195*) : (*e.xs* : *e.138*)) : [], (*Prog Synapse*)) ++
Eval(((*e.vs* : *s.195* : *e.138*) : (*e.is* : *e.137*) : (*e.ds* : [])) : [], (*Prog Synapse*))
) :
Eval(((*e.vs* : *s.195* : *e.138*) : (*e.is* : *e.137*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *s.195* : *e.138*) : (*e.is* : *e.137*) : (*s.t* : *wh2*) : (*e.time* : *e.136*) : (*e.ds* : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

The generalized configuration:

Match([], *e.202*, ([])),
Match((*e.ys*), (*e.137*), •),
Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.202*) : (*e.137*)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (*e.136*) : (*Invalid* *e.207* ++ • ++
Eval(((*e.s.208* : *e.209*) : (*s.210* . *s.211* : *e.212*) : (*e.s.213* : *e.214*)) : [],
(*Prog Synapse*))
++ *e.215*
) :
Eval(((*e.vs* : *e.216*) : (*e.is* : *e.137*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *e.216*) : (*e.is* : *e.137*) : (*s.t* : *wh2*) : (*e.time* : *e.136*) : (*e.ds* : [])) : [],

(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

3. *Gener-3*: *Append*

The third generalization is forced by the interpreted *Append* that is directly defined. It has happened along the recursive branch following the iteration of the protocol event *wm*. The corresponding generalized configuration C_{g_3} has 14 parameters and the badness similar to C_{g_2}.

The previous configuration:

Match([], e.138, ([])),
Match((e.ys), (e.137), •),
Matching(•, e.ys, … The 2-nd *Append* rule …, (e.138) : (e.137)),
Eval(•, (*Prog Synapse*))
EvalCall(*Loop*, (e.136) : (*Invalid I* [] ++ • ++
Eval(((e.ys : e.137) : (s.x : I) : (e.xs : e.138)) : [], (*Prog Synapse*)) ++
Eval(((e. vs : I : e.138) : (e . is : e.137) : (e. ds : [])) : [], (*Prog Synapse*))
++ []
) :
Eval(((e.vs : I : e.138) : (e.is : e.137) : (e.ds : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((e.vs : I : e.138) : (e.is : I : e.137) : (s.t : wm) : (e.time : e.136) : (e.ds : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

The current configuration:

Match([], e.138, ([])),
Match((e.ys), (e.137), •),
Matching(•, e.ys, … The 2-nd *Append* rule …, (e.138) : (e.137)),
Eval(•, (*Prog Synapse*))
EvalCall(*Loop*, (e.136) : (*Invalid I* s.240 • ++
Eval(((e.ys : e.137) : (s.x : s.240) : (e.xs : e.138)) : [], (*Prog Synapse*)) ++
Eval(((e. ys : e.137) : (s . x : I) : (e. xs : s.240 : e.138)) : [], (*Prog Synapse*)) ++
Eval(((e.vs : I : s.240 : e.138) : (e.is : e.137) : (e.ds : [])) : [], (*Prog Synapse*))
) :
Eval(((e.vs : I : s.240 : e.138) : (e.is : e.137) : (e.ds : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((e.vs : I : s.240 : e.138) : (e.is : I : e.137) : (s.t : wm) : (e.time : e.136) : (e.ds : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

The generalized configuration:
Match([], [e.250], ([])),
Match((e.ys), (e.137), •),
Matching(•, e.ys, … The 2-nd *Append* rule …, ([e.138]) : (e.137)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (e.136) : (*Invalid* *I* [e.255] ++ • ++
Eval(((e.ys : e.137) : (s.x : [s.257]) : (e.xs : [e.250])) : [], (*Prog Synapse*)) ++
Eval(((e. s.259 : [e.260]) : (s.261 s.262 : [e.137]) : (e. s.264 : [e.265])) : [],
(*Prog Synapse*))
++ [e.266]
) :
Eval(((e.vs : *I* : [e.267]) : (e.is : e.137) : (e.ds : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((e.vs : *I* : [e.267]) : (e.is : *I* : e.137) : (s.t : wm) : (e.time : e.136) : (e.ds : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

7.4.2 Folding

In this subsection the boxed pieces of the current configurations stand for the parts to be substituted in the corresponding previous configurations, while the double boxed variables of the previous configurations are variables of the substitutions folding the current configurations. The trivial part of these substitutions may be omitted.

1. *Folding*-1: The first folding action was done in the same context as the first generalization done. The only distinction consists in another action: a generated configuration was folded by the generalized configuration constructed by the first generalization. Thus there is the only variable to be nontrivially substituted by this folding. The variable is the accumulating parameter introduced by this generalization.

   The current configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* *e.137*):(*Dirty*):(*Valid* *I* [*I e.138*]), •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule …,
   (*e.136*) : (*Invalid* *e.137*) : (*Dirty*):(*Valid* *I* [*I e.138*])),
   Eval(•, (*Prog Synapse*))

   To the previous configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* *e.137*):(*Dirty*):(*Valid* *I* [[*e.138*]]), •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule …,
   (*e.136*) : (*Invalid* *e.137*) : (*Dirty*):(*Valid* *I* [[*e.138*]])),
   Eval(•, (*Prog Synapse*))

   The substitution:

$e.136 := e.136;\quad e.137 := e.137;\quad e.138 := I : e.138;$

2. *Folding*-2: The second folding action was performed by the first generalized configuration itself. The folded configuration ends a branch following the iteration of the protocol event *rm*.

   The current configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* $\boxed{I\ e.137}$):(*Dirty*):(*Valid* I $\boxed{[]}$), •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,
   (*e.136*) : (*Invalid* $\boxed{I\ e.137}$) : (*Dirty*):(*Valid* I $\boxed{[]}$)),
   Eval(•, (*Prog Synapse*))

   To the previous configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* $\boxed{\boxed{e.137}}$):(*Dirty*):(*Valid* I $\boxed{\boxed{e.138}}$), •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,
   (*e.136*) : (*Invalid* $\boxed{\boxed{e.137}}$) : (*Dirty*):(*Valid* I $\boxed{\boxed{e.138}}$)),
   Eval(•, (*Prog Synapse*))

   The substitution:
   $e.136 := e.136;\quad e.137 := I : e.137;\quad e.138 := [];$

3. *Folding*-3: The third folding action was done by a regular configuration C_1 below. Configuration C_1 is a descendant of the generalized configuration C_{g_1} constructed by the first generalization. The folded configuration ends a branch following the iteration of the protocol event *wm*. The corresponding substitution is completely trivial, i. e., the folded and folding configurations coincide modulo variables names.

   The sub-tree rooted in C_1 is completely folded. Configuration C_1 is declared as an entry point of a residual function definition. The fist rewriting rule of this definition is an exit from a recursion returning the value *True*, while the second and third branches refer to C_{g_1} and C_1, respectively.

   The current configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* $\boxed{I\ e.137}$):(*Dirty* I):(*Valid*) , •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,
   (*e.136*) : (*Invalid* $\boxed{I\ e.137}$) : (*Dirty* I):(*Valid*)),
   Eval(•, (*Prog Synapse*))

   To the previous configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* $\boxed{\boxed{e.137}}$):(*Dirty* I):(*Valid*), •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,
   (*e.136*) : (*Invalid* $\boxed{\boxed{e.137}}$) : (*Dirty* I):(*Valid*)),
   Eval(•, (*Prog Synapse*))

The substitution:
$e.136 := e.136$; $e.137 := I : e.137$;
Function $F1047(e.136, e.137)$ has been created.

4. *Folding*-4: The fourth folded configuration is a descendent of configuration C_{g_2}. It was folded by the regular configuration C_1 used in the third folding. The corresponding substitution is quite trivial. There are two parameters in both the folded and folding configurations. The folding substitution renames the parameter corresponding to the initial unknown data *e.time* and replaces the parameter standing for the unknown value of the *Invalid* counter with concatenation of completely unknown data early split into three parts.

   The current configuration:
   Match([], *e.206*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*) ,
   (*Invalid* $\boxed{e.207 \text{ ++ } e.203 \text{ ++ } e.215}$):(*Dirty I*):(*Valid*) , •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,
   (*e.206*) : (*Invalid* $\boxed{e.207 \text{ ++ } e.203 \text{ ++ } e.215}$) : (*Dirty I*):(*Valid*)),
   Eval(•, (*Prog Synapse*))

   To the previous configuration:
   Match([], *e.136*, ([])),
   Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* $\boxed{\boxed{e.137}}$):(*Dirty I*):(*Valid*), •),
   Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,
   (*e.136*) : (*Invalid* $\boxed{\boxed{e.137}}$) : (*Dirty I*):(*Valid*)),
   Eval(•, (*Prog Synapse*))

   The substitution:
   $e.136 := e.136$; $e.137 := e.207 \text{ ++ } e.203 \text{ ++ } e.215$;

5. *Folding*-5: The fifth folded configuration is a descendent of configuration C_{g_2} and it is folded by C_{g_2} as well. There are thirteen parameters occurring in C_{g_2}. Fife parameters are substituted with expressions containing names and/or types of some variables from the Synapse program model. One parameter is replaced with an expression including an application *Eval* that presents a generalized environment of this program model.

   The sub-tree rooted in C_{g_2} is completely folded. Configuration C_{g_2} is declared as an entry point of a residual function definition. The fist rewriting rule of this definition refers to C_1, while the seconds one refers to C_{g_2}, i. e., to itself.

   The current configuration:
   Match([], *e.202*, ([])),
   Match((*e.ys*), (*e.203*) , •),
   Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.202*) : (*e.203*)),
   Eval(•, (*Prog Synapse*)),

EvalCall(*Loop*, (*e.206*) : (*Invalid* [*e.207* ++ *s.223*] ++ • ++
Eval(((*e.*[*ys*]) : [*e.203*] : ([*s*].[*x*] : [*s.223*]) : (*e.*[*xs*] : [*e.202*])) : [], (*Prog Synapse*)) ++
[*Eval*(((*e.s.208* : *e.209*) : (*s.210*.*s.211* : *e.212*) : (*e.s.213* : *e.214*)) : [], (*Prog Synapse*)) ++ *e.215*]
) :
Eval(((*e.vs* : *e.216*) : (*e.is* : *e.203*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *e.216*) : (*e.is* : *e.203*) : (*s.t* : *wh2*) : (*e.time* : *e.206*) : (*e.ds* : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

To the previous configuration:
Match([], *e.202*, ([])),
Match((*e.ys*), (*e.203*) , •),
Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.202*) : (*e.203*)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (*e.206*) : (*Invalid* [*e.207*] ++ • ++
Eval(((*e.*[*s.208*] : [*e.209*]) : ([*s.210*].[*s.211*] : [*e.212*]) : (*e.*[*s.213*] : [*e.214*])) : [],
(*Prog Synapse*))
++ [*e.215*]
) :
Eval(((*e.vs* : *e.216*) : (*e.is* : *e.203*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *e.216*) : (*e.is* : *e.203*) : (*s.t* : *wh2*) : (*e.time* : *e.206*) : (*e.ds* : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

The substitution:
e.202 := *e.202*; *e.207* := *e.207 s.223*; *s.208* := *ys*; *e.209* := *e.203*;
s.210 := *'s'*; *s.211* := *x*; *e.212* := *s.223*; *s.213* := *xs*; *e.214* := *e.202*;
e.215 := *Eval*(((*e.s.208* : *e.209*) : (*s.210*.*s.211* : *e.212*) : (*e.s.213* : *e.214*)) : [],
(*Prog Synapse*)) ++ *e.215*;
e.216 := *e.216*; *e.203* := *e.203*; *e.206* := *e.206*;
Function
F1646(*e.202*, *e.203*, *e.206*, *e.207*, *s.208*, *e.209*, *s.210*, *s.211*, *e.212*, *s.213*, *e.214*, *e.215*, *e.216*)
has been created.

6. *Folding*-6: The sixth folding action was done by the regular configuration C_1 used above. The folding substitution increases counter *Invalid* and renames the variable corresponding to stream *e.time*.

The current configuration:
Match([], *e.136*, ([])),
Match((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*), (*Invalid* [*I e.137*]):(*Dirty I*):(*Valid*) , •),
Matching(•, *Test*((*Invalid e.is*) : (*Dirty e.ds*) : (*Valid e.vs*)), … The 2-nd *Loop* rule … ,

$\underline{(\ e.136)} : (Invalid\ \boxed{I\ e.137}\ ) : (Dirty\ I\ ) : (Valid\ )\ \ ),$

$Eval(\ \bullet,\ (Prog\ Synapse)\ )$

To the previous configuration:
$Match(\ [],\ e.136,\ ([])\ ),$
$Match(\ \underline{(Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)},\ \underline{(Invalid\ \boxed{\boxed{e.137}}\ ) : (Dirty\ I\ ) : (Valid\ )},\ \bullet\ ),$
$Matching(\ \bullet,\ \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )},\ \underline{\ldots\ \text{The 2-nd } Loop \text{ rule } \ldots},$
$\underline{(\ e.136)} : (Invalid\ \boxed{\boxed{e.137}}\ ) : (Dirty\ I\ ) : (Valid\ )\ \ ),$
$Eval(\ \bullet,\ (Prog\ Synapse)\ )$

The substitution:
$e.136\ :=\ e.136;\quad e.137\ :=\ I : e.137;$

7. *Folding*-7: The seventh folding action was done in a similar context as the fourth one done. The only distinction consists in increasing the *Invalid* counter by a constant.

   The current configuration:
   $Match(\ [],\ \boxed{e.254}\ ,\ ([])\ ),$
   $Match(\ \underline{(Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)}\ ,$
   $\underline{(Invalid}\ \boxed{I\ e.255\ ++\ e.251\ ++\ e.266}\ \underline{) : (Dirty\ I\ ) : (Valid\ )}\ ,\ \bullet\ ),$
   $Matching(\ \bullet,\ \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )},\ \underline{\ldots\ \text{The 2-nd } Loop \text{ rule } \ldots},$
   $\underline{(\ e.136)} : (Invalid\ \boxed{I\ e.255\ ++\ e.251\ ++\ e.266}\ ) : (Dirty\ I\ ) : (Valid\ )\ \ ),$
   $Eval(\ \bullet,\ (Prog\ Synapse)\ )$

   To the previous configuration:
   $Match(\ [],\ \boxed{\boxed{e.136}},\ ([])\ ),$
   $Match(\ \underline{(Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)},\ \underline{(Invalid\ \boxed{\boxed{e.137}}\ ) : (Dirty\ I\ ) : (Valid\ )},\ \bullet\ ),$
   $Matching(\ \bullet,\ \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )},\ \underline{\ldots\ \text{The 2-nd } Loop \text{ rule } \ldots},$
   $\underline{(\ e.136)} : (Invalid\ \boxed{\boxed{e.137}}\ ) : (Dirty\ I\ ) : (Valid\ )\ \ ),$
   $Eval(\ \bullet,\ (Prog\ Synapse)\ )$

   The substitution:
   $e.254\ :=\ e.136;\quad e.137\ :=\ I : e.255\ ++\ e.251\ ++\ e.266;$

8. *Folding*-8: The eighth folded configuration is a descendent of configuration C_{g_3} and it is folded by C_{g_3} as well. The corresponding folding substitution has the badness similar to the fifth folding substitution.

   The sub-tree rooted in C_{g_3} is completely folded. Configuration C_{g_3} is declared as an entry point of a residual function definition. The fist rewriting rule of this definition refers to C_1, while the seconds one refers to C_{g_3}, i. e., to itself.

   That in its turn allows us to fold the sub-tree rooted in the first generalized configuration C_{g_1}.

Configuration C_{g_1} is declared as an entry point of a residual function definition. The fist rewriting rule of this definition is an exit from a recursion returning the value *True*. The second rewriting rule calls this entry point C_{g_1}, while the third and fourth branches refer to C_{g_2} and C_{g_3}, respectively.

The current configuration:

Match([], *e.250*, ([])),
Match((*e.ys*), (*e.251*) , •),
Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.250*) : (*e.251*)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (*e.254*) : (*Invalid I* : [*e.255 ++ s.274*] ++ • ++
Eval(((*e.ys* : [*e.251*] :)(*s.x* : [*s.274*]) : (*e.xs*) : *e.250*) : [], (*Prog Synapse*)) ++
Eval(((*e.*[*ys*]) : [*e.251*] : (*s* . *x* : [*s.257*]) : (*e.*[*xs*] : [*s.274 e.250*])) : [], (*Prog Synapse*)) ++
[*Eval*(((*e.s.259* : *e.260*) : (*s.261 .s.262* : *e.263*) : (*e.s.264* : *e.265*)) : [], (*Prog Synapse*)) ++ *266*]
) :
Eval(((*e.vs* : *I* : *e.267*) : (*e.is* : *e.251*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *e.267*) : (*e.is* : *I* : *e.251*) : (*s.t* : *wm*) : (*e.time* : *e.254*) : (*e.ds* : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

To the previous configuration:

Match([], *e.250*, ([])),
Match((*e.ys*), (*e.251*) , •),
Matching(•, *e.ys*, … The 2-nd *Append* rule … , (*e.250*) : (*e.251*)),
Eval(•, (*Prog Synapse*)),
EvalCall(*Loop*, (*e.254*) : (*Invalid I* : [*e.255*] ++ • ++
Eval(((*e.ys*) : [*e.251*] : (*s.x* : [*s.257*]) : (*e.xs* : *e.250*)) : [], (*Prog Synapse*)) ++
Eval(((*e.*[*s.259*] : [*e.260*]) : ([*s.261*] . [*s.262*] : [*e.263*]) : (*e.*[*s.264*] : [*e.265*])) : [],
(*Prog Synapse*))
++ [*e.266*]
) :
Eval(((*e.vs* : *I* : *e.267*) : (*e.is* : *e.251*) : (*e.ds* : [])) : (*Dirty I*) : (*Valid*), (*Prog Synapse*)) ++
Eval(((*e.vs* : *I* : *e.267*) : (*e.is* : *I* : *e.251*) : (*s.t* : *wm*) : (*e.time* : *e.254*) : (*e.ds* : [])) : [],
(*Prog Synapse*)), (*Prog Synapse*)
),
Eval(•, (*Prog Synapse*))

The substitution:

e.255 := *e.255* ++ *s.274*; *s.257* := *s.274*; *e.250* := *e.250*; *s.259* := *ys*;
e.260 := *e.251*; *s.261* := *'s'*; *s.262* := *x*; *e.263* := *s.257*;
s.264 := *xs*; *e.265* := *s.274 e.250*;
e.266 := *Eval*(((*e.s.259* : *e.260*) : (*s.261 .s.262* : *e.263*) : (*e.s.264* : *e.265*)) : [],
(*Prog Synapse*)) ++ *e.266*;
e.267 := *e.267*; *e.251* := *e.251*; *e.254* := *e.254*;

Function
$F1959(e.249, e.250, e.253, e.254, s.256, s.258, e.259, s.260, s.261, e.262,$
$s.263, e.264, e.265, e.266)$ has been created.
Function $F637(e.136, e.137, e.138)$ has been created.

9. *Folding*-9: The last folded configuration belongs to the last branch originating from the initial configuration. It is folded by the regular configuration C_1, which belongs to a parallel path and was used in the third folding.

The current configuration:
$Match(\ [],\ \boxed{e.101}\ ,\ ([])\ ),$
$Match(\ \underline{(Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)},\ \underline{(Invalid\ \boxed{e.102}\ ) : (Dirty\ \ I\ \ ) : (Valid\ )}\ ,\ \bullet\ ),$
$Matching(\ \bullet,\ \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )},\ \underline{\ldots\ \text{The 2-nd } Loop \text{ rule } \ldots},$
$\underline{(\ e.101) : (Invalid\ \boxed{e.102}\ ) : (Dirty\ \ I\ ) : (Valid\ )}\ \ ),$
$Eval(\ \bullet,\ (Prog\ Synapse)\ )$

To the previous configuration:
$Match(\ [],\ \boxed{\boxed{e.136}}\ ,\ ([])\ ),$
$Match(\ \underline{(Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)},\ \underline{(Invalid\ \boxed{\boxed{e.137}}\ ) : (Dirty\ \ I\ ) : (Valid\ )},\ \bullet\ ),$
$Matching(\ \bullet,\ \underline{Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ )},\ \underline{\ldots\ \text{The 2-nd } Loop \text{ rule } \ldots},$
$\underline{(\ e.136) : (Invalid\ \boxed{\boxed{e.137}}\ ) : (Dirty\ \ I\ ) : (Valid\ )}\ \ ),$
$Eval(\ \bullet,\ (Prog\ Synapse)\ )$

The substitution:
$e.136\ :=\ e.101;\quad e.137\ :=\ e.102;$