Russian Academy of Sciences
Ailamazyan Program Systems Institute

First International Workshop on Verification and Program Transformation

Affiliated with the 25th International Conference
on Computer Aided Verification
CAV 2013

Pre-Proceedings
Saint Petersburg, Russia, July 13–14, 2013

Pereslavl-Zalessky

УДК 004.42(063)
ББК 22.18

В874

First International Workshop on Verification and Program Transformation // Pre-Proceedings of the first International Workshop on Verification and Program Transformation. Saint Petersburg, Russia, July 13-14, 2013 / *Edited by A. P. Lisitsa and A. P. Nemytykh.* — Pereslavl Zalessky: Ailamazyan University of Pereslavl, 2013, **119** p. — 978-5-901795-29-3

Первый международный семинар по верификации и преобразованиям программ // Сборник трудов Первого международного семинара по верификации и преобразованиям программ, г. Санкт-Петербург, 13-14 июля 2013 г. /*Под редакцией А. П. Лисицы и А. П. Немытых.* – Переславль-Залесский: «Университет города Переславля», 2013, **119** с. *(англ).* – 978-5-901795-29-3

© Ailamazyan Program Systems Institute of RAS 2013
Институт программных систем им. А.К. Айламазяна РАН

ISBN 978-5-901795-29-3

Workshop Organization

Program Committee Co-Chairs

Alexei Lisitsa, The Liverpool University, Great Britain
Andrei Nemytykh, Program Systems Institute of RAS, Russia

Program Committee

Maurice Bruynooghe, Katholieke Universiteit Leuven, Belgium
Geoff Hamilton, Dublin City University, Republic of Ireland
Boris Konev, The Liverpool University, Great Britain
Johan Nordlander, Luleå University of Technology, Sweden
Sven Schewe, The Liverpool University, Great Britain
Peter Sestoft, The IT University of Copenhagen, Denmark
Morten Sørensen, Formalit, Denmark
Simon Thompson, University of Kent, Great Britain

Invited Speakers

Amir M. Ben-Amram, Academic Colledge of Tel-Aviv-Yaffo, Israel
Bernd Finkbeiner, Universität des Saarlandes, Germany
Jérôme Leroux, ICNRS, LaBRI, Bordeaux, France
Alberto Pettorossi, Università di Roma Tor Vergata and
Istituto di Analisi dei Sistemi ed Informatica "A. Ruberti", Italy
Simon Thompson, University of Kent, Great Britain

Sponsoring Institutions

The Russian Academy of Sciences

Russian Foundation for Basic Research (№ 13-07-06028-г)

The *Centre National de la Recherche Scientifique*
of the French Ministry of Higher Education and Research

Contents

Cryptographic Protocol Verification via Supercompilation (A Case Study) 7
Abdulbasit Ahmed, Alexei Lisitsa, Andrei P. Nemytykh

Verification of Imperative Programs through Transformation of Constraint Logic Programs 26
Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, Maurizio Proietti

Ranking Functions for Linear-Constraint Loops 43
Amir M. Ben-Amram

Transforming Undecidable Synthesis Problems into Decidable Problems . 53
Bernd Finkbeiner

On the Termination of Positive Supercompilation 54
G. W. Hamilton

Acceleration For Presburger Petri Nets 74
Jérôme Leroux

Transforming Event B Models into Verified C# Implementations 78
Dominique Méry and Rosemary Monahan

Ping-Pong Protocols as Prefix Grammars and Turchin Relation 99
Antonina Nepeivoda

Program Transformation for Program Verification 116
Alberto Pettorossi, Maurizio Proietti

Building Trustworthy Refactoring Tools 118
Simon Thompson

Cryptographic Protocol Verification via Supercompilation

(A Case Study)

Abdulbasit Ahmed[1], Alexei P. Lisitsa[2] and Andrei P. Nemytykh[3]

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

Abstract

It has been known for a while [35, 36, 12] that program transformation techniques, in particular, program specialization, can be used to prove the properties of programs automatically. For example, if a program actually implements (in a given context of use) a constant function sufficiently powerful and semantics preserving program transformation may reduce the program to a syntactically trivial "constant" program, pruning unreachable branches and proving thereby the property. Viability of such an approach to verification has been demonstrated in previous works [16, 19, 18] where it was applied to the verification of parameterized cache coherence protocols and Petri Nets models [11, 20]. In this paper we further extend the method and present a case study on its application to the verification of a cryptographic protocol. The protocol is modeled by functional programs at different levels of abstraction and verification via program specialization is done by using Turchin's supercompilation method.

Keywords: Program verification, cryptographic protocols, program specialization, supercompilation, program analysis, program transformation.

1 Introduction

Progam specialization techniques traditionally have been applied for optimization purposes. For example, if in a given program $p(x, y)$ a value of the argument x is fixed to some x_0 then a specialization transformation can be applied to produce a program $q_{x_0}(y)$ such that for any value of y $p(x_0, y) = q_{x_0}(y)$. What is more, specialization exploits partial knowledge of the input and other syntactical

structures of the program p to make the specialized program more efficient, e.g. by pruning the *unreachable* fragments of code.

But program transformations can also be used for analysis of programs [22] and more specifically for their verification [14, 19]. In the experiments discussed below we use a specializer based on Turchin's *supercompilation* method [36, 35, 32, 33, 19, 4] as the program specialization technique. This paper extends the authors' *parameterized testing* method (see [19]) for modeling and verification of global safety properties of parameterized protocols via supercompilation. The idea of the method is very simple and natural. In order to verify a safety property of a (parameterized) non-deterministic protocol S the later is specified as a functional program $\phi(i, \bar{x})$, where input i takes an encoding of the initial state of S and input $\bar{x}$ takes an encoding of a sequence of actions of S. The program returns a state of the protocol after executing the sequence of actions $\bar{x}$ starting in the initial state i. Let $T_P(s)$ be a *testing program* which given a state s checks whether a (safety) property P holds on s ($True$ or $False$). Consider a composition $T_P(\varphi_S(i, \bar{x}))$. This program first simulates the execution of the protocol and then tests the required property. Now, provided we have used adequate encodings, the statement "the safety property P holds in any possible state reachable by the execution of the protocol S" is equivalent to the statement "the program $T_P(\varphi_S(i, \bar{x}))$ never returns the value $False$". The general method assumes that a semantics-preserving program transformation (e.g. supercompilation) is applied to $T_P(\varphi_S(i, \bar{x}))$ in order to transform it to a form from which one can easily establish the required property. For example if the statements of the form "`return False;`" have been eliminated during the transformation the required property holds. This approach has shown to be efficient for the verification of various (classes of) parameterized and infinite-state protocols and systems, including parameterized cache coherence protocols [19], Java MetaLock algorithm [17], coverablity for Petri Nets [20, 11]. The simplified theoretical model of verification via supercompilation approach is presented in [19] and the completeness of the method for the verification of coverability for Petri Nets can be found in [11, 20].

In this paper we show how to extend this approach to the verification of cryptographic protocols. We are interested in development of methods of both functional modeling of the cryptographic protocols and the capability of supercompilation (an automated program specialization method) to verify the corresponding program models.

Our case study here is a classical Needham-Schroeder Public Key authentication protocol [25]. We use the parameterized testing scheme outlined above and model the nondeterministic protocol in terms of a strict functional programming language, using an oracle guessing an evaluation path of this computing system. We supercompile the program model in a context of an initial protocol config-

uration and an unknown evaluation path as well as an intruder behavior. We hope for simple *syntactic* (explicit) properties of the resulting program, which allow us to conclude that the protocol satisfies a security property hidden in the semantics of the original program specification of the protocol (see above the remark on the "`return False;`" statement). Otherwise we interactively use the supercompiler (SCP4 [26, 27, 28]) for searching a possible counterexample (an attack on the protocol).

The limit on the length of this paper does not allow us to give an introduction to supercompilation. Here we should only enumerate properties of the method, which are crucial for the following discussion. The properties are: (1) a supercompiler tries to prune as many unreachable (in the context of specialization) program's branches (and more generally – formally possible evaluation's paths of the program being specialized) as it can; (2) in a sense the program is simplified in the superompilation process, so it becomes more amenable to the analysis.

The paper introducing the main original ideas of supercompilation as a program optimization method can be found in [36]: it is a paper written by the creator of supercompilation – V. F. Turchin. The most complete description of supercompilation ideas is given by V. F. Turchin in a report [34]. A simplified version of supercompilation is considered in [33]. See also [19, 27].

This paper assumes the reader has basic knowledge of concepts of functional programming, pattern matching, term rewriting systems and cryptographic protocols' specifications.

The paper is organized as follows. Sect. 2 provides an informal specification of the Needham-Schroeder public key (NSPK) protocol and the program presentation language. Sect. 3 introduces the developed basics ideas behind our functional modeling of the cryptographic protocols. In Sect. 4 these ideas are instantiated for a program model of NSPK, which is our case study. A verification attempt of the NSPK program model is discussed in Sect. 5, including of generating the classical Lowe attack on NSPK. A program model of a corrected version of NSPK successfully verified by supercompiler SCP4 is described in Sect. 6. In Sect. 7 we point out another NSPK program model verified by SCP4. Related work is discussed in Sect. 8. The residual programs generated by SCP4 have been relegated to appendices [1].

2 Preliminaries

This section deals with specification of the Needham-Schroeder public key (NSPK) protocol and the presentation language used in the protocol program model described below.

2.1 The Needham-Schroeder Public Key Protocol

Let us consider the following version of the NSPK authentication protocol [25]. NSPK involves two legal participants Alice (`A`) and Bob (`B`) who aim to authenticate each other by sending *encrypted and signed* messages via an *open* channel. The authentication is required even in the presence of an intruder. The intruder (`C`) is aware of the communication rules and tries to convince `B` that he (`C`) is `A`, observing the messages moving in the channel and sometimes replacing them. I.e. `C` may intercept a message and, if he is able to read it (e.g. after decryption), he may compose a fake message based on the read information, otherwise he may send the original message back in the channel.

We assume that all participants use public-key cryptography, that is every participant, legal or intruder, has a pair of private and public keys assigned to him/her. Everyone can use a public key of anyone else to encrypt a message, but only the holder of the corresponding private key may decrypt such an encrypted message. We denote the public keys used by the protocol participants as `EA`, `EB`, `EC` respectively.

Below we denote the clients' signatures with their names. Because the signatures are constant for a long time (they do not change from one session to another), additionally, to make the communications more secure the participants use random *unique* numbers (nonces) `rA`, `rB`, `rC` in the correspondence. The numbers depend on the sessions.

Normal (secure) NSPK evaluation (a sequence of steps/messages) maintaining of information protection is as follows:

1. `A` → `B : EB(rA,A) - A` initiates a communication session and sends (in the channel) a nonce `rA` signed with `A` and encrypted with the public key `EB`. By means of this message `A` asks for ensuring that he communicates with `B` rather than with someone else.

2. `B` → `A : EA(rA,rB) - B`, upon receiving the first message from `A`, decrypts it with his private key and in response to that sends the received nonce `rA` back to `A`. I.e. `B` confirms that he can decrypt the first message. In addition `B` signs his message with a nonce `rB` and encrypts it with the key `EA`. Now `B` asks for ensuring that he makes a contact with `A` rather than someone else. I.e. a person trying to communicate must know the private key `EA` and hence (s)he can read `B`'s reply.

3. `A` → `B : EB(rB) - A` certifies that he can read the message sent by `B`: he sends the received nonce `rB` back to `B`. This message is encrypted with `EB`.

4. After the three message exchange above participants `B` and `A` assume that they communicate with each other.

Upon completion of all the steps above the *connection between* `B` *and* `A` *is established.* The objective of the protocol session is achieved. Any other messages received by the legal participants cause concern of an unauthorized access and interrupting the session. In this case it is said about an attack on the protocol.

The NSPK specification above assumes the probability of guessing the numbers `rA`, `rB` is zero. Because an intruder may interfere in the communication between `A` and `B`, NSPK is parameterized with both an unknown number of the messages and the messages themselves. Indeed, the intruder is able to generate any number of unknown messages. Additionally, the set $\mathcal{K}$ of all public keys is also a NSPK parameter, because the concrete legal participants are unknown in advance. In the program model of the NSPK protocol below we assume that `A` may initiate an execution of the protocol by sending the first request to any participant whose public key belongs to $\mathcal{K}$. The numbers `rA`, `rB` are the parameters of a fixed communication session. Other parameters of NSPK's evolution are the messages generated by `C` - as a consequence of guessing or on the basis of analyzing the messages intercepted from the channel.

The verification objective is to prove that there exist no attacks on the protocol or otherwise to construct such an attack. What constitutes an attack here we understand as a session of the protocol has successfully compeleted but at the same time an intruder took a part in the session.[1]

Exact specification is embodied into a program model of the protocol and is presented and discussed in Section 4.

2.2 The Presentation Language

We present our program examples in a variant of a pseudocode for functional programs while the real supercompilation experiments with the programs were done in a strict (call by value) functional programming language REFAL [38, 39]. The programs given below are written as *strict* term rewriting systems based on pattern matching. The sentences in the programs are ordered from the top to the bottom to be matched. The data set is a free monoid of concatenation with an additional unary constructor, which is denoted only with its parentheses (that is without a name). The colon sign stands for the concatenation. The constant `[]` is the unit of the concatenation and may be omitted, others constants c are identifiers. The monoid of the data may be defined with the following grammar:
d `::=` `[]` `|` c `|` d_1 `:` d_2 `|` `(`d`)`
Thus a datum is a finite sequence (including the empty sequence). Let v, f denote a variable and a function name correspondingly, then the monoid of the

[1]It has turned out that this very conserative definition of tha attack does not lead to the generation of the spurious attacks on NSPK protocol. This surprising property of NSPK in the context of our modeling is interesting itself.

corrresponding terms may be defined as follows:
t `::= [] |` c `|` v `|` f`(` $args$ `) |` t_1 `:` t_2 `| (`t`)`
$args$ `::=` t `|` t`,` $args$
To be closer to REFAL we use three kinds of variables: s.variables range over *identifiers* (e.g. `True`), e.variables range over the whole data set, while t.variables range over the data set excluding `[]`.

Examples of identifiers are `B2, Message, refused, Ms`. Examples of the variables are `s.rA, t.1, e.cls, e.memory, e.A_st`. I.e. the variable's names may be the identifiers or natural numbers.

3 The Principles of Modeling

This section informally explains our basic principles of modeling of the cryptographic protocols with ordered term rewriting systems based on pattern matching. The next section applies the principles to a concrete program model.

Modeling of the dynamic of the protocols. A (parameterized) nondeterministic protocol S is specified as a functional program $\phi(i, \bar{x})$, where input i takes an encoding of the initial state of S and input $\bar{x}$ takes an encoding of a finite sequence of actions of S. More precisely, the program ϕ is a term rewriting system. Given an action x_0 and a current state of S, the program ϕ computes the following state of S and goes on to the next action. Repeating such steps ϕ returns a state of the protocol after executing the whole sequence of the actions $\bar{x}_0$ starting in the initial state i. These steps are the rewriting steps of the term rewriting system ϕ. In the case of cryptographic protocols (similar to NSPK) the state consists of the open channel's content and the protocol memory split into the memories of all participants of S. The open channel contains the only (current) message.

Privacy policy for the protocol participants' memory. The protocol memory is a sequence p_1, p_2, q_3, where p_i is the memory of the i-th legal protocol participant, while q_3 is the intruder memory. We encode this sequence with a term of the following form
`Memory:t.A:t.B:e.memory`. Here `t.A ::= (A:e.A)`, `t.B ::= (B:e.B)`, while `e.memory ::= (C:e.C) | []`. I.e. the intruder part of the memory may be omitted. The last is just a technical trick: the term `(C:e.C)` informs the intruder that the channel contains a message put by himself, while `[]` (no terms) means the channel's message was renewed by someone else.

The protocol specification requires a privacy policy for the memory of `A, B` and `C`. We achieve that by the following programming discipline. Given a partic-

ipant of the protocol, the participant performs his/her *given* step of the protocol with the only term rewriting step. The patterns of the sentences, which can realize the given step, are not allowed to specify the part of the memory belonging to the other participants not taking part in this step. For example, let the active participant be `B` then the following two protocol memory patterns `Memory:t.A:(B:[]):e.memory`, `Memory:(A:e.A):(B:B2):t.C` are allowed, while the pattern `Memory:(A:to:s.EB):(B:B2):t.C` is not. The reason is as follows. The pattern `(A:e.A)` specifies the name (i.e. `A`) to whom this memory part belongs, but it does not disclose any part of the content of `A`'s memory, which is `e.A`. The patterns `t.A, t.C, e.memory` hide even the names of the corresponding participants. The pattern `(A:to:s.EB)` specifies partly `A`'s memory, but that is forbidden for `B`.

A similar programming discipline based on (lack of) knowledge of encryption keys guarantees the (in)capability of reading the encrypted messages being transmitted through the communication channel.

Intruder behavior. According with the general privacy principle described above the intruder `C` is not able to spy in the memory of the legal participants. He cannot decrypt a message if he does not know the key decrypting the message. `C` never replaces his own message in the channel. `C` is able to synthesize messages from data taken from the pattern (through its variables' values) of a sentence corresponding to `C`, but he cannot split the variables' values. For example, if the pattern includes the variables `e.xy`, `t.A`, `s.rB`, then `C` may produce the term `e.xy t.A e.xy (s.rB)`, but cannot specify (even partly) the values of these variables. See Section 4.1 for additional explanations of the intruder behavior logic.

Tracing. The program model ϕ may trace some additional information on the protocol evaluation, leaving it step by step in the program result. Such a trace may be useful for constructing an attack on the protocol, if any.

In general the actions $\bar{x}$ may be abstracted whenever the chosen abstraction together with the channel content and the memory state allow to unambiguously reconstruct the corresponding action. Additionally the discrete dynamic system ϕ may compute some information on properties of the generated states and carry it to the final result of ϕ.

4 The Program Model of NSPK

Let us consider the NSPK program model (see Figure 1). It is supposed that if the protocol started, then the open channel can contain only one message:

sending a next message is replacement of the current message in the channel. The term `Key:s.EB` denotes the public key used by `A` to initiate execution of the protocol. Information on the messages sent in the channel is stored in the memory

`Memory:(A:e.A_st):(B:e.B_st):e.C_part`

The memory is a sequence of the protocol participants' storages. The information kept in a storage is available only to the participant corresponding to the storage. The storage for the participant `e.C_part` may be empty, in which case it is denoted by `[]`.

The function `mainNSPK` defines the input point of the protocol dynamic system. In its definition we assume that in any (correct) execution there exist at least two messages sent via the channel plus the initial message. Thus the sequence of the (abstract) messages without the initial message can be represented as the term `Ms:t.x1:t.x2:e.cls`.

The initial encrypted message sent by `A` is represented with `s.EB:(rA:A)` where the identifier `rA` represents a nonce, `A` is the name of the sender. The message is encrypted with the public key of an addressee with whom `A` would like to initiate the mutual authentication.

The function `Loop(Ms:e.cls, e.broadcast, Memory:e.memory)` models the protocol dynamic. Its first argument ranges over the finite sequences of the messages' *abstracts*. A message abstract includes its sender name and an abstract of the corresponding message (if needed). `Loop` puts a next message (`e.broadcast`) in the channel (the second argument), modifies the memory and returns a trace of the messages passed over the channel followed by a flag. The trace is a sequence of terms of the following form `(s.sender_name : e.ms_info)`. Below we use the trace for constructing an attack on NSPK. The flag is `connection` - the fact of authentication of the legal participants or `refused` - the fact of interruption of the current session. Let us consider the function sentences.

`(1):` The message sequence is empty. Mutual authentication is not established. The negotiation is interrupted.

`(2a-2b):` The first message of `B`. This fact is represented both in the first `Loop`'s argument and in the memory's part accessible to `B` which is empty `[]`. Notice that the other part of the memory is not available to `B` because the pattern `Memory:(A:e.A):(B:[]):e.memory` does disclose the content neither of `e.A` (`A`'s memory) nor `e.memory`. The `B` modifies his part of the memory, keeping in mind (recording the term `B2`) that the next his message will be the second. He traces his current message as a part of the returning result. This message is a response to the request of `A`. It confirms the fact that `B` can decrypt the message received from `A`: `B's` message contains the nonce `s.rA` sent by `A`.

Additionally, the memory is cleaned from possible intruder records. Note that *this removing of the intruder storage is just a technical trick and does not lead to loss of possible attacks* (see the case `(5)` below for explanations). In the case `(2a)`, from `B`'s point of view, all the protocol rules are respected, while in the case `(2b)` `B` suspects an attack and interrupts the session.

```
mainNSPK( Ms:t.x1:t.x2:e.cls, Key:s.EB ) =
    Test( (A:s.EB:(rA:A)): Loop( Ms:t.x1:t.x2:e.cls, Message:s.EB:(rA:A),
                                                Memory:(A:to:s.EB):(B:[])));
/*1.*/
Loop( Ms, e.message, e.memory ) = refused;
/*2a.*/
Loop(Ms:(B:[]):e.cls,Message:EB:(s.rA:A),Memory:(A:e.A):(B:[]):e.memory)
     = (B:EA:(s.rA:rB)) :
       Loop( Ms:e.cls, Message:EA:(s.rA:rB), Memory:(A:e.A):(B:B2) );
/*2b.*/
Loop( Ms:(B:[]):e.cls, t.message, t.memory ) = refused;
/*3a.*/
Loop( Ms:(A:e.A):e.cls, Message:EA:(rA:s.r),
                        Memory:(A:to:s.EB):t.B:e.memory )
     = (A:s.EB:(s.r)) :
       Loop( Ms:e.cls, Message:s.EB:(s.r), Memory:(A:to:s.EB):t.B );
/*3b.*/
Loop( Ms:(A:e.A):e.cls, t.message, t.memory ) = refused;
/*4a.*/
Loop( Ms:(B:B2):e.cls, Message:EB:(rB),
            Memory:(A:e.A):(B:B2):e.memory ) = (B:end):connection;
/*4b.*/
Loop( Ms:(B:B2):e.cls, t.message,  t.memory ) = refused;
/*5.*/
Loop( Ms:(C:e.xy):e.cls, Message:EC:t.r, Memory:(A:e.A):(B:e.B) )
  = (C:e.xy): Loop(Ms:e.cls, Message:e.xy,Memory:(A:e.A):(B:e.B):(C:C1));
/*6.*/
Loop( Ms:(C:e.xy):e.cls, t.message, Memory:(A:e.A):(B:e.B):(C:C1) )
     = Loop( Ms:e.cls, t.message, Memory:(A:e.A):(B:e.B):(C:C1) );

Test( connection ) = True;
Test( e.trace : refused ) = refused;
Test( (C:e.C):e.x:connection ) = (C:e.C):e.x:False;
Test( t.x1:e.trace:connection ) = t.x1 : Test( e.trace:connection );
```

Figure 1: The NSPK program model.

`(3a-3b)`: The second `A`'s message. `A` checks the fact that his first message

was read. That is confirmed with the nonce `rA`. `A` traces this message in the returning result. This `A`'s response returns the nonce `s.r` back to the communication partner. The response is encrypted with the same public key used for the first `A`'s message: the key was early stored in the `A`'s memory. The other part of the memory is not available to `A`. Additionally, the memory is cleaned from possible intruder records (see the case `(5)` below). In the case `(3a)`, from `A`'s point of view, all the protocol rules are respected, while in the case `(3b)` `A` suspects an attack and interrupts the session.

`(4a-4b):` `(4a)` The second `B`'s message: this fact is confirmed with the memory. `B` checks the fact that his first message was read. He does that by means of the nonce `rB` returned back to him. `B` decides that the person signed by `A` is the `A`. The authentication is established. `(4b)` `B` suspects an attack and interrupts the session.

`(5):` The intruder `C` checks in the memory that the last message sent in the channel was sent by someone else. That is the memory does not contain records written by `C`, which he enters in there and which the legal participants throw out from the memory (see above). The part of the memory belonging to the legal participants is not available to `C`. `C` knowing the open keys tries to impersonate a legal participant: by means of guessing a message `e.xy` and sending it in the channel. Here `e.xy` is an arbitrary message (*this free variable ranges over the whole data set*), therefore if `C` can decrypt the intercepted message `t.r`, then the message may include any information obtained from `t.r`. Notice that in this version of the specification we abstract away capabilities of `C` and allow him to send an arbitrary message. As a consequence, the cleaning of the intruder storage being done by the legal participants in the cases `(2)` and `(3)` does not restrict any possibility of `C` to generate messages. See further discussion of this point in the next subsection. Additionally, `C` records the term `C1` in the memory, which means that this current message is sent by himself rather than someone else. Otherwise the protocol evolution runs in an infinite loop, in which `C` should analyze the message sent by himself.

`(6):` `C` checks in the memory that the last message sent in the channel was sent by himself. If the memory contains the record `C1`, then `C` does not replace the current message in the channel: otherwise the protocol evolution passes in an infinite loop. This case is the last in the function `Loop`. That means if the message was sent by someone else, then the program goes into deadlock (pattern recognition impossible). Thus we model the NSPK evaluation deadlock (`C` cannot decrypt the intercepted message) with the program interpretation deadlock (abnormal stop).

The function `Test` checks correctness of a *fixed* NSPK evolution and if the *concrete* message sequence leads to completion of the *whole* negotiation session, in which `C` took a part, then the function returns the message sequence leading

to the attack on the protocol. The third pattern of the function detects that the intruder replaced a message in the channel, and therefore the protocol is not secure.

4.1 On Parameterization of the Intruder Behavior Logic

Modeling of the behavior logic of the intruder `C` is crucial for detecting an attack on any crypto-protocol. In our program model the case (6) of the function `Loop` is technical: it only transfers the turn of reading/sending a message to the legal protocol participants. The substantial intruder behavior logic is modeled in the fifth case. Guessing a message, particularly, may include *any subtle analyses and syntheses* of the messages (including their sequence), therefore our model completely parameterizes the intruder logic. It has a potential advantage that more concrete models of intruder behaviour may fail to find an attack. Our model above, being under supercompilation, provides for completely automatic generation of the intruder logic. On the other hand, potential disadvantage of fully parameterized intruder behaviour model is that "guessing a message" `e.xy` allows generating a message, which might be not relevant at all to the protocol being considered. That may lead to generation of spurious attacks. E.g. the intruder may guess a public key, which is not accessible in principle, and may encrypt (using this key) a message and initiate an attack. Speaking formally, a protocol is not secure if not only an attack was generated, but the attack is proven to be realized.

Furthemore, that may be considered as a "*potential attack*": if the intruder might guess something, then he might break the negotiation confidentiality of the legal protocol participants. Such an attack may also be very interesting for analyzing the protocols. Below we show that the NSPK logic model chosen above does not lead to the spurious attacks. As a result of a verification attempt we will construct the only classical attack [23] on the protocol.

5 A Verification Attempt of the NSPK Program Model

The NSPK program model `P` described above terminates on any input data, returning a result or falling into an abnormal state (pattern recognition impossible). That is because the number of the messages (see the first `P`'s argument) taking part in a given session is finite (but unbounded) and is exhausted step by step. In other words, `P` is primitive recursive with respect to its first argument. This important (*syntactic*) property of `P` allows us to conclude the following. In the case the supercompiler SCP4 is able to transform `P` to a residual program

`P'` such that `P'` has simple *syntactic* properties showing that `P'` is identically true on its domain, then the NSPK program model is secure.

E.g. such a syntactic property, both in REFAL terms and in the presentation language terms, is the lack of in `P`'s right-side top-level passive subexpressions without the identifier `False`. An expression is said to be top-level passive iff it does not contain any function call and it is not a part of a function call argument. If `P`' has not such a property, then that may mean both existence of an attack on NSPK and weakness of the supercompiler. In both cases the program model `P` has to be additionally analyzed.

As a result of specialization of our NSPK program model the supercompiler generates a program `P'`[2] containing two sentences with the identifier `False`. Let us consider one of them:

```
F122((B:B2):e.140, e.142, EB:(rB))
                         = (A:EC:(rA:A)):(C:EB:(rA:A)):(B:EA:(rA:rB)):
                           (A:EC:(rB)):e.142:(C:EB:(rB)):(B:end):False;
```

Its right-side expression is completely passive. The second sentence has the same property. Thus we are forced to suppose that the program model is not secure or the supercompiler is not able to solve the verification task. In both cases we have to continue analyzing the program model.

5.1 Interactive Search for an Attack on NSPK

Now we start an interactive search for attacks on the program model. It is natural to consider the number of the messages sent in the channel as a working time measure of NSPK. In our model terms this number is the length of the sequence `e.cls` plus 3 (see the term `Ms:t.1:t.2:e.cls` in the input point in Figure 1 and Section 4). We will interactively use the supercompiler. We input the sequence `e.cls` with a fixed length and unknown members. When the length equals 1 or 2 the corresponding residual programs are identically true on their domains. The following input point ($ln($`e.cls`$) = 3$) given as a supercompilation task leads to the generation of an attack on NSPK:

```
mainNSPK( Ms:t.1:t.2:t.3:t.4:t.5, Key:s.EB )
```

The corresponding residual program can be found in Appendix A in [1]. We see the only sentence containing `False`. It is labeled with a comment. `False` stays in the completely passive right side. The residual program is a one-step program: it does not contain formal syntax loops. The sentence we are interested in is:

[2]The residual program can be found in [21].

```
mainNSPK'( Ms:(C:EB:(rA:A)):(B:[]):(A:e.129):(C:EB:(rB)):(B:B2), Key:EC )
               = (A:EC:(rA:A)):(C:EB:(rA:A)):(B:EA:(rA:rB)):(A:EC:(rB))
                                                   :(C:EB:(rB)):(B:end):False;
```

According to the semantics of the result returned by the program model we conclude that a likely attack on NSPK is the following message sequence

```
(A:EC:(rA:A)):(C:EB:(rA:A)):(B:EA:(rA:rB)):(A:EC:(rB)):(C:EB:(rB)
                            ):(B:end)
```

We are writing a *likely* attack for two reasons: (1) the considered sentence may be unreachable during interpretation of the residual program `P'`; (2) the constructed attack may be spurious (see 4.1). The first doubt may be easily dissolved: it is a one-step program[3], hence the only input data, which may lead to the sentence, must be matched with this sentence pattern. But the pattern has no variables, therefore this input data has to be:

```
Ms:(C:EB:(rA:A)):(B:[]):(A:e.129):(C:EB:(rB)):(B:B2), Key:EC
```

We input this data both to the residual and to the source program and run both programs (by the interpreter). In such a way we make sure that both programs return the right-side expression of the sentence considered, i.e. the chosen data belongs to the domains of both the residual and the *source* program.

The second doubt above is canceled with a paper published in 1995 [23]: the constructed attack is the classical attack detected by G. Lowe. In this "man-in the-middle" attack the intruder `C` using an authentication request from a participant `A` impersonates `A` in an authentication exchange with `B`. The attack runs as follows.

1. `(A:EC:(rA:A))` - `A` initiates a session with `C`;

2. `(C:EB:(rA:A))` - `C` receives the first message, decrypts it with his key, encrypts the result with `EB` and replaces in the channel the first message with the changed one;

3. `(B:EA:(rA:rB))` - `B` taking into account that the second message signed by `A` sends the current message signed with the nonce `rB` and encoded with `EA` in the channel;

4. `(A:EC:(rB))` - `A` seeing that his previous message successfully decoded decides that `B'` key is really the key used for the first message and sends (in the channel) confirmation of reading the third message, once again using the key `EC`;

[3] I.e. it is the only rewriting step necessary to produce the program's result.

5. `(C:EB:(rB)))` - `C` intercepts and decodes the forth message, he sends the (re)encoded message ensuring `B` that the last `B's` message was read and he is a legal protocol participant;

6. the technical `(B:end)` term just means that `B` receives the fifth message and falsely decides that `C` is `A`.

6 A Corrected Version of NSPK

Let us consider a corrected version of the protocol suggested G. Lowe [23]. The second step described above (Section 2.1) can be specified more accurately: `B` $\rightarrow$ `A : EA(rA,rB,EB)`. I.e. additionally, `B` sends his public key `EB` to `A`. As a consequence, at the third step, `A` may compare the received key with the key used for encoding his initial message. She interrupts the session if the keys do not coincide. Now the corresponding cases of the program model (the function `Loop`) must be corrected as follows.

```
...
/*2a.*/
Loop(Ms:(B:[]):e.cls, Message:EB:(s.rA:A),Memory:(A:e.A):(B:[]):e.memory)
     = (B:EA:(s.rA:rB:EB)) :
       Loop( Ms:e.cls, Message:EA:(s.rA:rB:EB), Memory:(A:e.A):(B:B2) );
...
/*3a.*/
Loop( Ms:(A:e.A):e.cls, Message:EA:(rA:s.r:s.EB),
                        Memory:(A:to:s.EB):t.B:e.memory )
     = (A:s.EB:(s.r)) :
       Loop( Ms:e.cls, Message:s.EB:(s.r), Memory:(A:to:s.EB):t.B );
...
```

The supercompilation result (see Appendix B in [1]) of the corrected program model `P` never returns `False`. Thus we conclude the model `P` is secure. The corrected version of NSPK has been successfully verified by the supercompiler SCP4.

7 Explicit Capabilities Intruder Model

Another version of the NSPK protocol specification in terms of the Refal language has been given in the MSc Dissertation of the first author [2] where the supercompiler SCP4 has been applied for finding an attack on the original protocol and for verification of the corrected version. The main difference with the version presented here is that in [2] the capabilities of the intruder have been

explicitly specified within the program model. That led to much longer specification which nevertheless was sucessfully analysed by the interactive use of SCP4. Full details can be found in [2].

8 Related Work and Concluding Remarks

The history of verification of cryptographic protocols spans more than twenty years. Needham and Schroeder [25] mentioned that security protocols are prone to extremely subtle errors, and the need for techniques to verify the correctness of such protocols is great. The work of Dolev and Yao [6] was the first to address the verification of cryptographic protocols by utilizing a formal model of the protocols and the environment. Since then the various routes in the development of the verification techniques have been taken. Model checking approach, e.g. [24, 23] has been particularly successful in demonstrating the power of formal methods by discovering the flaws in the protocols by using finite state abstractions. Theorem proving, both *interactive* [30] and automated [5] allowed to approach the verification of parameterized and infinite state protocols. The technique utilizing declarative logic programming has been developed and implemented in *ProVerif* tool [3] which is capable of the efficient automated verification of large variety of cryptographic protocols. To make it possible a special modification of the standard Prolog semantics has been implemented.

Another line of work which led to the research presented in this paper is the development of the verification methods based on program transformation techniques and in particular on supercompilation.

M. Leuschel with his coauthors [14, 13] were the pioneers who suggested to apply a program specialization method for verification of various infinite state computing systems. The systems were modeled in terms of logic programs. The used method is known as partial deduction. A. Roychoudhury and C.R. Ramakrishnan in [31] have used fold/unfold transformations of logic programs for the verification of parameterized concurrent systems. F. Fioravanti, A. Pettorossi, M. Proietti and V. Senni [7, 8] proposed to use constraint logic programs, which give more powerful means for dealing with infinite sets of states. They also studied various strategies of generalization used for verification [9].

In [10] G. W. Hamilton described using of his distillation algorithm as a proof assistant for transformation of programs into a tail recursive form in which some properties of the programs can be easily verified by the application of inductive proof rules.

As mentioned above, functional modeling and verification (by supercompilation) of global *safety* properties of nondeterministic parameterized (i.e. infinite state) cache coherence protocols was studied by A. Lisitsa and A. Nemytykh

[16, 19, 15, 17]. A. Lisitsa and A. Nemytykh in [20] and A. Klimov in [11] apply supercompilation to verification of Petri Nets models.

Modeling of cryptographic protocols is more subtle. The work of A. Ahmed [2] was the first to address the verification of the cryptographic protocols via supercompilation using the functional modeling of a variant of the Dolev Yao model. Antonina Nepeivoda [29] considers modeling and verifying of the ping-pong cryptographic protocols. Verification of her program models essentially uses generalization based on Turchin's relation [37].

In this paper we have presented a method for modeling of cryptographic protocols by functional programs and their exploration via program optimization.

The method was demonstrated on the Needham-Schroeder public key protocol. Using the supercompiler SCP4 we have explored the NSPK protocol and interactively detected the classical attack on the protocol. Then we have automatically verified the corrected version of the protocol. This case study provides with the guidelines to the design of a semi-decision procedure for the verification of cryptographic protocols based on supercompilation. The procedure would either terminate with the proof of the correctness of a protocol, or generate the attacks on the protocol, or in the worst case would not terminate, if the protocol is correct, but supercompiler is not powerful enough to prove that.

The paper length limit does not allow us to provide some details of the supercompilation technique and we refer the reader to [36, 35, 32, 33, 19, 4].

The fact of the successful using of the completely parameterized intruder behavior logic in the presented model reflects some properties of the NSPK protocol. It will be very interesting to describe a class of cryptographic protocols which can be verified with such an intruder model, using the presented approach without producing of spurious attacks.

The approach we advocate in this paper is very flexible and due to the use of the expressive programming language for the specification of the models and properties can cover the wide spectrum of models - from completely parameterized to more definite, such as the Dolev-Yao model and their variants [6, 2]. Exploration of various directions, their formal representation and comparison with other approaches such as ProVerif is a topic of ongoing and future work.

Acknowlegements

We are grateful to the reviewers of the paper for their generous and constructive comments, which both improved the presentation in this paper and will influence our future work.

References

[1] A. Ahmed, A. P. Lisitsa, and A. P. Nemytykh. Appendices to the paper: Cryptographic protocol verification via supercompilation (a case study). [online], 2013. Available at URL `http://www.botik.ru/pub/local/scp/refal5/NSPK-appendices.pdf`.

[2] Abdulbasit M. Ahmed. Verification of cryptographic protocols via supercompilation. Master's thesis, Department of Computer Science, University of Liverpool, 2008. 76pp, Available at URL `http://www.csc.liv.ac.uk/~alexei/A.Ahmed.dissertation.pdf`.

[3] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *In 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001. `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`.

[4] M. Bolingbroke and S. Peyton-Jones. Supercompilation by evaluation. In *the third ACM Haskell symposium on Haskell (Haskell '10)*, pages 135–146, NY, USA, 2010. ACM New York.

[5] Ernie Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[6] D. Dolev and A. C. Yao. On security of public key protocols. *IEEE trans. on Information Theory*, IT-29:198–208, 1983.

[7] 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.

[8] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems: An experimental evaluation. In M. Alpuente, editor, *the Proc. of LOPSTR 2010*, volume 6564 of *LNCS*, pages 164–183. Springer, 2011.

[9] 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.

[10] G. W. Hamilton. Distilling programs for verification. *Electronic Notes in Theoretical Computer Science*, 190(4):17–32, 2007. The Proc. of the International Conference on Compiler Optimization Meets Compiler Verification.

[11] 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.

[12] 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.

[13] M. Leuschel and H. Lehmann. Coverability of reset petri nets and other wellstructured transition systems by partial deduction. In *Proc. CL 2000*, volume 1861 of *LNAI*, pages 101–115. Springer, 2000.

[14] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *LNCS*, pages 63–82, Venice, Italy, 2000.

[15] 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.

[16] 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.

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

[18] A. P. Lisitsa and A. P. Nemytykh. Extracting bugs from the failed proofs in verification via supercompilation. In Bernhard Beckert and Reiner Hahnle, editors, *Tests and Proofs: Papers Presented at the Second International Conference TAP 2008*, number 5/2008 in Reports of the Faculty of Informatics, Univesitat Koblenz-Landau, pages 49–65, April 2008.

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

[20] A. P. Lisitsa and A. P. Nemytykh. Solving coverability problems by supercompilation. Presentation on the Workshop on Reachability Problems - RP'08, 2008.

[21] A. P. Lisitsa and A. P. Nemytykh. A fail result of an experiment on verification (via supercompilation) of a NSPK program model. [online], 2012. `http://refal.botik.ru/protocols/r_False_NSPK.ref`.

[22] A. P. Lisitsa and A. P. Nemytykh. A note on program specialization. what can syntactical properties of residual programs reveal? arXiv:1209.5407, 2012.

[23] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 1294(3):131–133, 1995.

[24] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur-phi. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[25] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of ACM*, 21(12):993–999, 1978.

[26] A. P. Nemytykh. The supercompiler Scp4: General structure. (extended abstract). In *the Proc. of PSI'03*, volume 2890 of *LNCS*, pages 162–170, 2003.

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

[28] A. P. Nemytykh and V. F. Turchin. The supercompiler Scp4: Sources, on-line demonstration. [online], 2000. http://www.botik.ru/pub/local/scp/refal5/.

[29] Antonina Nepeivoda. Ping-pong protocols as prefix grammars and Turchin relation. In *Proc. of Verification and Program Transformation*, 2013.

[30] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[31] Abhik Roychoudhury and C. R. Ramakrishnan. Unfold/fold transformations for automated verification of parameterized concurrent systems. In *Program Development in Computational Logic*, pages 261–290, 2004.

[32] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, pages 486–479. The MIT Press, 1995.

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

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

[35] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, volume 85 of *LNCS*, pages 645–657. Springer-Verlag, 1980.

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

[37] 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.

[38] 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.

[39] 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/.

Verification of Imperative Programs through Transformation of Constraint Logic Programs

Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2] and Maurizio Proietti[3]

[1] DEC, University 'G. D'Annunzio', Pescara, Italy
{emanuele.deangelis,fioravanti}@unich.it
[2] DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[3] IASI-CNR, Rome, Italy maurizio.proietti@iasi.cnr.it

1 Introduction

In the last decade formal techniques have received a renewed attention as the basis of a methodology for increasing the reliability of software artifacts and reducing the cost of software production. In particular, great efforts have been made to devise automatic techniques such as *software model checking* [19], for verifying the correctness of programs with respect to their specifications.

In many software model checking techniques, the use of *constraints* has been very effective both for constructing models of programs and for reasoning about them [1, 6, 7, 9, 14, 16, 18, 30, 31]. Several kinds of constraints have been considered, such as equalities and inequalities over booleans, integers, reals, and finite or infinite trees. By using constraints we can represent in a symbolic, compact way the (possibly infinite) sets of values computed by programs and, in general, the sets of states which are reached during program executions. Then, by using powerful solvers specifically designed for the classes of constraints we have mentioned above, we can reason about program properties in an efficient way.

In this paper we consider a simple imperative programming language with integer and array variables and we use Constraint Logic Programming (CLP) [17] as a metalanguage for representing imperative programs, their executions, and the properties to be verified. We use constraints consisting of linear equalities and inequalities over integers. Note, however, that the method presented here is parametric with respect to the constraint domain which is used. By following an approach originally presented in [30], a given imperative program *prog* and its interpreter are first encoded as a CLP program. Then, the proofs of the properties of interest about the program *prog* are sought by analyzing that derived

CLP program. In order to improve the efficiency of that analysis, it is advisable to first *compile-away* the CLP interpreter of the language in which *prog* is written. This is done by specializing the interpreter with respect to the given program *prog* using well-known *program specialization* techniques [20, 30].

In previous papers [7, 12] we have shown that program specialization can be used not only as a preprocessing step to improve the efficiency of program analysis, but also as a means of analysis on its own. In this paper, we extend that approach and we propose a verification method based on more general *unfold/fold transformation rules* for CLP programs [4, 10, 34].

Transformation-based verification techniques are very appealing because they are parametric with respect to both the programming languages in which programs are written, and the logics in which the properties of interest are specified. Moreover, since the output of a transformation-based verification method is a program which is *equivalent* to the given program with respect to the properties of interest, we can apply a *sequence* of transformations, thereby refining the analysis to the desired degree of precision (see, for instance, [7]).

The specific contributions of this paper are the following. We present a verification method based on a set of transformation rules which includes the rules for performing *conjunctive definition*, *conjunctive folding*, and *goal replacement*, besides the usual rules for *unfolding* and *constraint manipulation* which are used during program specialization. The rules for conjunctive definition and conjunctive folding allow us to introduce and transform new predicates defined in terms of *conjunctions* of old predicates, while program specialization can only deal with new predicates that correspond to specialized versions of exactly *one* old predicate. The goal replacement rule allows us to replace conjunctions of predicates and constraints by applying equivalences that hold in the least model of the CLP program at hand, while program specialization can only replace conjunctions of constraints.

By using these more powerful definition and folding rules, we extend the specialization-based verification method in the following two directions: (i) we verify programs with respect to specifications given by sets of CLP clauses (for instance, recursively defined relations among program variables), whereas program specialization can only deal with specifications given by constraints, and (ii) we verify programs manipulating arrays and other data structures by applying equivalences between predicates that axiomatize suitable properties of those data structures (for instance, the ones deriving from the axiomatization of the theory of arrays [27]).

The paper is organized as follows. In Section 2 we present our transformation-based verification method. First, we introduce a simple imperative language and we describe how correctness properties of imperative programs can be translated into predicates defined by CLP programs. We also present a general strategy

for applying the transformation rules to CLP programs, with the objective of verifying the properties of interest. Next, we present two examples of application of our verification method. In particular, in Section 3 we show how we deal with specifications given by recursive CLP clauses, and in Section 4 we show how we deal with programs which manipulate arrays. Finally, in Section 5 we discuss the related work which has been recently done in the area of automatic program verification.

2 The Transformation-Based Verification Method

We consider an imperative C-like programming language with integer and array variables, assignments (`=`), sequential compositions (`;`), conditionals (`if` and `if else`), while-loops (`while`), and jumps (`goto`). A program is a sequence of (labeled) commands, and in each program there is a unique `halt` command which, when executed, causes program termination.

The semantics of our language is defined by a *transition relation*, denoted $\Longrightarrow$, between *configurations*. Each configuration is a pair $\langle\!\langle c, \delta \rangle\!\rangle$ of a command c and an *environment* δ. An environment δ is a function that maps: (i) every integer variable identifier x to its value v, and (ii) every integer array identifier a to a *finite* function from the set $\{0, \ldots, dim(a){-}1\}$, where $dim(a)$ is the dimension of the array a, to the set of the integer numbers. The definition of the relation $\Longrightarrow$ is similar to the 'small step' operational semantics given in [32], and is omitted.

Given an imperative program *prog*, we address the problem of verifying whether or not, starting from any *initial configuration* that satisfies the property φ_{init}, the execution of *prog* eventually leads to a *final configuration* that satisfies the property φ_{error}, also called an *error configuration*. This problem is formalized by defining an *incorrectness triple* of the form $\{\!\{\varphi_{init}\}\!\}$ *prog* $\{\!\{\varphi_{error}\}\!\}$, where φ_{init} and φ_{error} are encoded by CLP predicates defined by (possibly recursive) clauses. We say that a program *prog* is *incorrect* with respect to φ_{init} and φ_{error}, whose free variables are assumed to be among $z_1, \ldots, z_r$, if there exist environments δ_{init} and δ_h such that: (i) $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$ holds, (ii) $\langle\!\langle \ell_0 : c_0,\ \delta_{init} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \mathtt{halt},\ \delta_h \rangle\!\rangle$, and (iii) $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$ holds, where $\ell_0 : c_0$ is the first labeled command of *prog* and $\ell_h : \mathtt{halt}$ is the unique `halt` command of *prog*. A program is said to be *correct* with respect to φ_{init} and φ_{error} iff it is not incorrect with respect to φ_{init} and φ_{error}. Note that this notion of correctness is equivalent to the usual notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$.

Our verification method is based on the formalization of the notion of program incorrectness by using a predicate `incorrect` defined by a CLP program.

In this paper a CLP program is a finite set of clauses of the form `A :- c,B`, where `A` is an atom, `c` is a constraint (that is, a possibly empty conjunction of linear equalities and inequalities over the integers), and `B` is a goal (that is, a possibly empty conjunction of atoms). The conjunction `c,B` is called a *constrained goal*. A clause of the form: `A :- c` is called a *constrained fact*. We refer to [17] for other notions of CLP with which the reader might be not familiar.

We translate the problem of checking whether or not the program *prog* is incorrect with respect to the properties φ_{init} and φ_{error} into the problem of checking whether or not the predicate `incorrect` is a consequence of the CLP program T defined by the following clauses:

```
incorrect :- initConf(X), reach(X).
reach(X) :- tr(X,X1), reach(X1).
reach(X) :- errorConf(X).
```

together with the clauses for the predicates `initConf(X)`, `errorConf(X)`, and `tr(X,X1)`. They are defined as follows: (i) `initConf(X)` encodes an initial configuration satisfying the property φ_{init}, (ii) `errorConf(X)` encodes an error configuration satisfying the property φ_{error}, and (iii) `tr(X,X1)` encodes the transition relation $\Longrightarrow$. (Note that in order to define `initConf(X)`, `errorConf(X)`, and `tr(X,X1)` and, in particular, to represent operations over the integer variables and the elements of arrays, we need constraints.) The predicate `reach(X)` holds if an error configuration `Y` such that `errorConf(Y)` holds, can be reached from the configuration `X`.

The imperative program *prog* is correct with respect to the properties φ_{init} and φ_{error} iff `incorrect` $\notin M(T)$, where $M(T)$ denotes the *least model* of program T [17]. Due to the presence of integer variables and array variables, $M(T)$ is in general an infinite model, and both the bottom-up and top-down evaluation of the query `incorrect` may not terminate. In order to deal with this difficulty, we propose an approach to program verification which is symbolic and, by using program transformations, allows us to avoid the exhaustive exploration of the possibly infinite space of reachable configurations.

Our verification method consists in applying to program T a sequence of program transformations that preserve the least model $M(T)$ [10]. In particular, we apply the following *transformation rules*, collectively called *unfold/fold rules*: (i) (*conjunctive*) *definition*, (ii) *unfolding*, (iii) *goal replacement*, (iv) *clause removal*, and (v) (*conjunctive*) *folding*. Our verification method is made out of the following two steps.

Step (A): *Removal of the Interpreter.* Program T is *specialized* with respect to the given *prog* (on which `tr` depends), `initConf`, and `errorConf`, thereby deriving a new program $T1$ such that: (i) `incorrect` $\in M(T)$ iff `incorrect` $\in M(T1)$, and (ii) `tr` does not occur explicitly in $T1$ (in this sense we say that the interpreter

is removed or compiled-away).

Step (B): *Propagation of the Initial and Error Properties.* By applying a sequence of unfold/fold transformation rules, the CLP program $T1$ is transformed into a new CLP program $T2$ such that `incorrect` holds in $M(T2)$ iff *prog* is incorrect with respect to the given initial and error properties. The objective of Step (B) is to propagate the initial and the error properties so as to derive a program $T2$ where the predicate `incorrect` is defined by either (i) the fact '`incorrect.`' (in which case *prog* is incorrect), or (ii) the empty set of clauses (in which case *prog* is correct). In the case where neither (i) nor (ii) holds, that is, in program $T2$ the predicate `incorrect` is defined by a non-empty set of clauses not containing the fact '`incorrect.`', we cannot conclude anything about the correctness of *prog* and, similarly to what has been proposed in [7], we iterate Step (B) in the hope of deriving a program where either (i) or (ii) holds. Obviously, due to undecidability limitations, it may be the case that we never get a program where either (i) or (ii) holds.

Steps (A) and (B) are both instances of the *Transform* strategy outlined in Figure 1 below.
In particular, the application of the *Transform* strategy for performing Step (A) coincides with the fully automatic specialization strategy presented in [7]. In the *Transform* strategy we make use of the following rules, where P is the input CLP program, and *Defs* is a set of clauses, called *definition clauses*, constructed as we indicate in that strategy.

Definition Rule. By this rule we introduce a clause of the form `newp(X) :- c,G`, where `newp` is a new predicate symbol, `X` is a tuple of variables occurring in (`c,G`), `c` is a constraint, and `G` is a non-empty conjunction of atoms.

Unfolding Rule. Given a clause C of the form `H :- c,L,A,R`, where `H` and `A` are atoms, `c` is a constraint, and `L` and `R` are (possibly empty) conjunctions of atoms, let us consider the set $\{\mathtt{K}_i \text{ :- } \mathtt{c}_i,\mathtt{B}_i \mid i = 1,\ldots,m\}$ made out of the (renamed apart) clauses of P such that, for $i=1,\ldots,m$, `A` is unifiable with $\mathtt{K}_i$ via the most general unifier ϑ_i and $(\mathtt{c},\mathtt{c}_i)\,\vartheta_i$ is satisfiable (thus, the unfolding rule performs some constraint solving operations). By unfolding C w.r.t. A using P, we derive the set $\{(\mathtt{H} \text{ :- } \mathtt{c},\mathtt{c}_i,\mathtt{L},\mathtt{B}_i,\mathtt{R})\,\vartheta_i \mid i = 1,\ldots,m\}$ of clauses.

Goal Replacement Rule. If a constrained goal $\mathtt{c}_1, \mathtt{G}_1$ occurs in the body of a clause C, and $M(P) \models \forall\,(\mathtt{c}_1, \mathtt{G}_1 \leftrightarrow \mathtt{c}_2, \mathtt{G}_2)$, then we derive a new clause D by replacing $\mathtt{c}_1, \mathtt{G}_1$ by $\mathtt{c}_2, \mathtt{G}_2$ in the body of C.

The equivalences which are needed for goal replacements are called *laws* and their validity in $M(P)$ can be proved once and for all, before applying the *Transform* strategy.

Folding Rule. Given a clause E of the form: `H :- e, L, Q, R` and a clause D in *Defs* of the form `K :- d, D` such that: (i) for some substitution ϑ, $\mathtt{Q} = \mathtt{D}\,\vartheta$, and

Input: Program P.
Output: Program *TransfP* such that `incorrect` $\in M(P)$
iff `incorrect` $\in M(\mathit{TransfP})$.

INITIALIZATION:
TransfP $:= \emptyset$; *InDefs* $:= \{$`incorrect:- c, G`$\}$; *Defs* $:=$ *InDefs*;

while in *InDefs* there is a clause C *do*

UNFOLDING: Apply the unfolding rule at least once, and derive from C a set $U(C)$ of clauses;

GOAL REPLACEMENT: Apply a sequence of goal replacements, and derive from $U(C)$ a set $R(C)$ of clauses;

CLAUSE REMOVAL: Remove from $R(C)$ all clauses whose body contains an unsatisfiable constraint;

DEFINITION & FOLDING: Introduce a (possibly empty) set *NewDefs* of new predicate definitions and add them to *Defs* and to *InDefs*;
Fold the clauses in $R(C)$ different from constrained facts by using the clauses in *Defs*, and derive a set $F(C)$ of clauses;

InDefs $:=$ *InDefs* $- \{C\}$; *TransfP* $:=$ *TransfP* $\cup F(C)$;

end-while;

REMOVAL OF USELESS CLAUSES:
Remove from *TransfP* all clauses whose head predicate is useless.

Figure 1: The *Transform* strategy.

(ii) $\forall\,(\mathtt{e} \rightarrow \mathtt{d}\,\vartheta)$ holds, then by folding E using D we derive `H :- e, L, K`ϑ`, R`.

Removal of Useless Clauses. The set of *useless predicates* in a given program Q is the greatest set U of predicates occurring in Q such that `p` is in U iff every clause with head predicate `p` is of the form $\mathtt{p(X)}$ `:-` $\mathtt{c, G_1, q(Y), G_2}$, for some `q` in U. A clause in a program Q is *useless* if the predicate of its head is useless in Q.

The termination of the *Transform* strategy is guaranteed by suitable techniques for controlling the unfolding and the introduction of new predicates. We refer to [24] for a survey of techniques which ensure the finiteness of unfolding. The introduction of new predicates is controlled by applying *generalization operators* based on various notions, such as *widening*, *convex hull*, *most specific generalization*, and *well-quasi ordering*, which have been proposed for analyzing and transforming CLP programs (see, for instance, [6, 8, 13, 29]).

The correctness of the strategy with respect to the least model semantics directly follows from the fact that the application of the transformation rules

complies with some suitable conditions that guarantee the preservation of that model [10].

Theorem 1. (Termination and Correctness of the *Transform* strategy) (i) *The Transform strategy terminates.* (ii) *Let program TransfP be the output of the Transform strategy applied on the input program P. Then,* `incorrect` $\in M(P)$ *iff* `incorrect` $\in M(\mathit{TransfP})$.

3 Verification of Recursively Defined Properties

In this section we will show, through an example, that our verification method can be used when the initial properties and the error properties are specified by (possibly recursive) CLP clauses, rather than by constraints only (as done, for instance, in [7]). In order to deal with that kind of properties, during the DEFINITION & FOLDING phase of the *Transform* strategy, we allow ourselves to introduce new predicates which are defined by clauses of the form: `Newp :- c, G`, where `Newp` is an atom with a new predicate symbol, `c` is a constraint, and `G` is a conjunction of *one or more* atoms. This kind of predicate definitions allows us to perform program verifications that cannot be done by the technique presented in [7], where the goal `G` is assumed to be a single atom.

Let us consider the following program *GCD* that computes the greatest common divisor z of two positive integers m and n, denoted $gcd(m,n,z)$.

GCD: ℓ_0: $x = m$;
ℓ_1: $y = n$;
ℓ_2: `while` $(x \neq y)$ $\{$ `if` $(x > y)$ $x{=}x{-}y$; `else` $y{=}y{-}x$; $\}$;
ℓ_3: $z = x$;
ℓ_h: `halt`

We also consider the incorrectness triple $\{\!\{\varphi_{init}(m,n)\}\!\}$ *GCD* $\{\!\{\varphi_{error}(m,n,z)\}\!\}$, where:
(i) $\varphi_{init}(m,n)$ is $m \geq 1 \wedge n \geq 1$, and (ii) $\varphi_{error}(m,n,z)$ is $\exists\, d\, (gcd(m,n,d) \wedge d \neq z)$. These properties φ_{init} and φ_{error} are defined by the following CLP clauses 1 and 2–5, respectively:

```
1. phiInit(M,N) :- M≥1, N≥1.
2. phiError(M,N,Z) :- gcd(M,N,D), D≠Z.
3. gcd(X,Y,D) :- X>Y, X1=X−Y, gcd(X1,Y,D).
4. gcd(X,Y,D) :- X<Y, Y1=Y−X, gcd(X,Y1,D).
5. gcd(X,Y,D) :- X=Y, Y=D.
```

The predicates `initConf` and `errorConf` specifying the initial and the error configurations, respectively, are defined by the following clauses:

```
6. initConf(cf(cmd(0,asgn(int(x),int(m))),
       [[int(m),M],[int(n),N],[int(x),X],[int(y),Y],[int(z),Z]])) :- phiInit(M,N).
7. errorConf(cf(cmd(h,halt),
       [[int(m),M],[int(n),N],[int(x),X],[int(y),Y],[int(z),Z]])) :- phiError(M,N,Z).
```

Thus, the CLP program encoding the given incorrectness triple consists of clauses 1–7 above, together with the clauses defining the predicates `incorrect`, `reach`, and `tr`.

Now we perform Step (A) of our verification method, which consists in the removal of the interpreter, and we derive the following CLP program:

```
 8. incorrect :- M≥1, N≥1, X=M, Y=N, new1(M,N,X,Y,Z).
 9. new1(M,N,X,Y,Z) :- X>Y, X1=X−Y, new1(M,N,X1,Y,Z).
10. new1(M,N,X,Y,Z) :- X<Y, Y1=Y−X, new1(M,N,X,Y1,Z).
11. new1(M,N,X,Y,Z) :- X=Y, Z=X, Z≠D, gcd(M,N,D).
```

By moving the constrained atom '`Z≠D, gcd(M,N,D)`' from the body of clause 11 to the body of clause 8, we can rewrite clauses 8 and 11 as follows (this rewriting is correct because in clauses 9 and 10 the predicate `new1` modifies neither the value of `M` nor the value of `N`):

```
8r.  incorrect :- M≥1,N≥1, X=M, Y=N, Z≠D, gcd(M,N,D), new1(M,N,X,Y,Z).
11r. new1(M,N,X,Y,Z) :- X=Y, Z=X.
```

Note that we could avoid performing the above rewriting and obtain a similar program where the constraints characterizing the initial and the error properties occur in the same clause by starting our derivation from a more general definition of the reachability relation. However, an in-depth analysis of this variant of our verification method is beyond the scope of this paper.

Now we will perform Step (B) of the verification method by applying the *Transform* strategy to the derived program consisting of clauses {3, 4, 5, 8r, 9, 10, 11r}. Initially, we have that the sets *InDefs* and *Defs* of definition clauses are both equal to {8r}.

Unfolding. We start off by unfolding clause 8r w.r.t. the atom `new1(M,N,X,Y,Z)`, and we get:

```
12. incorrect :- M≥1, N≥1, X=M, Y=N, X>Y, X1=X−Y, Z≠D,
                 gcd(M,N,D), new1(M,N,X1,Y,Z).
13. incorrect :- M≥1, N≥1, X=M, Y=N, X<Y, Y1=Y−X, Z≠D,
                 gcd(M,N,D), new1(M,N,X,Y1,Z).
14. incorrect :- M≥1, N≥1, X=M, Y=N, X=Y, Z=X, Z≠D, gcd(M,N,D).
```

By unfolding clauses 12, 13, and 14 w.r.t. the atom `gcd(M,N,D)`, we derive:

15. $\texttt{incorrect :- M}\geq\texttt{1, N}\geq\texttt{1, M>N, X1=M-N, Z}\neq\texttt{D,}$
$\quad\texttt{gcd(X1,N,D), new1(M,N,X1,N,Z).}$
16. $\texttt{incorrect :- M}\geq\texttt{1, N}\geq\texttt{1, M<N, Y1=N-M, Z}\neq\texttt{D,}$
$\quad\texttt{gcd(M,Y1,D), new1(M,N,M,Y1,Z).}$

(The unfolding of clause 14 produces the empty set of clauses because the constraint '$\texttt{X=M, Z=X, Z}\neq\texttt{D, M=D}$' is unsatisfiable.) The GOAL REPLACEMENT and CLAUSE REMOVAL phases leave the set of clauses produced by the UNFOLDING phase unchanged, because no laws are available for the predicate `gcd`.

DEFINITIONS & FOLDING. In order to fold clauses 15 and 16, we perform a generalization step and we introduce a new predicate defined by the following clause:

17. $\texttt{new2(M,N,X,Y,Z,D) :- M}\geq\texttt{1, N}\geq\texttt{1, Z}\neq\texttt{D, gcd(X,Y,D), new1(M,N,X,Y,Z).}$

The body of this clause 17 is the most specific generalization of the bodies of clause 8r (which is the only clause in *Defs*), and clauses 15 and 16 (which are the clauses to be folded). Now, clauses 15 and 16 can be folded by using clause 17, thereby deriving:

18. $\texttt{incorrect :- M}\geq\texttt{1, N}\geq\texttt{1, M>N, X1=M-N, Z}\neq\texttt{D, new2(M,N,X1,N,Z,D).}$
19. $\texttt{incorrect :- M}\geq\texttt{1, N}\geq\texttt{1, M<N, Y1=N-M, Z}\neq\texttt{D, new2(M,N,M,Y1,Z,D).}$

Clause 17 defining the new predicate `new2` is added to *Defs* and *InDefs* and, since the latter set is not empty, we perform a new iteration of the while-loop body of the *Transform* strategy.

UNFOLDING. By unfolding clause 17 w.r.t. `new1(M,N,X,Y,Z)` and then unfolding the resulting clauses w.r.t. `gcd(X,Y,Z)`, we derive:

20. $\texttt{new2(M,N,X,Y,Z,D) :- M}\geq\texttt{1, N}\geq\texttt{1, X>Y, X1=X-Y, Z}\neq\texttt{D,}$
$\quad\texttt{gcd(X1,Y,D), new1(M,N,X1,Y,Z).}$

21. $\texttt{new2(M,N,X,Y,Z,D) :- M}\geq\texttt{1, N}\geq\texttt{1, X<Y, Y1=Y-X, Z}\neq\texttt{D,}$
$\quad\texttt{gcd(X,Y1,D), new1(M,N,X,Y1,Z).}$

DEFINITION & FOLDING. Clauses 20 and 21 can be folded by using clause 17, and we derive:

22. $\texttt{new2(M,N,X,Y,Z,D) :- M}\geq\texttt{1, N}\geq\texttt{1, X>Y, X1=X-Y, Z}\neq\texttt{D, new2(M,N,X1,Y,Z).}$
23. $\texttt{new2(M,N,X,Y,Z,D) :- M}\geq\texttt{1, N}\geq\texttt{1, X<Y, Y1=Y-X, Z}\neq\texttt{D, new2(M,N,X,Y1,Z).}$

No new predicate definition is introduced, and the *Transform* strategy exits the while-loop. The final program *TransfP* is the set $\{18, 19, 22, 23\}$ of clauses, which contains no constrained facts. Hence both predicates `incorrect` and `new2` are useless and all clauses of *TransfP* can be removed. Thus, the *Transform* strategy terminates with *TransfP*$=\emptyset$ and we conclude that the imperative program *GCD* is correct w.r.t. the given initial and error properties.

4 Verification of Array Programs

In this section we apply our verification method to the following program *ArrayMax* which computes the maximal element of an array:

ArrayMax: ℓ_0: `while` $(i<n)$ { `if` $(a[i] > max)$ $max = a[i]$; $i = i+1$; };
ℓ_h: `halt`

We consider the following incorrectness triple:
$\{\!\{\varphi_{init}(i,n,a,max)\}\!\}$ *ArrayMax* $\{\!\{\varphi_{error}(n,a,max)\}\!\}$ where: (i) $\varphi_{init}(i,n,a,max)$ is $i = 0 \wedge n = dim(a) \wedge n \geq 1 \wedge max = a[i]$, and (ii) $\varphi_{error}(n,a,max)$ is $\exists k\ (0 \leq k < n \wedge a[k] > max)$.

First, we construct a CLP program T which encodes the above incorrectness triple, similarly to what has been done in Section 3. The predicates `initConf(X)` and `errorConf(X)` specifying the initial and the error configurations, respectively, are defined by the following clauses:

```
1. initConf(cf(cmd(0, asgn(int(x), int(0))),
        [[int(i), I], [int(n), N], [array(a), (A, N)], [int(max), Max]]))
                                                   :- phiInit(I, N, A, Max).
2. errorConf(cf(cmd(h, halt),
        [[int(i), I], [int(n), N], [array(a), (A, N)], [int(max), Max]]))
                                                   :- phiError(N, A, Max).
3. phiInit(I, N, A, Max) :- I=0, N≥1, read((A, N), I, Max).
4. phiError(N, A, Max) :- K≥0, N>K, Z>Max, read((A, N), K, Z).
```

Now we start off by applying Step (A) of our verification method which consists in the removal of the interpreter. From program T we obtain the following program $T1$:

```
5. incorrect :- I=0, N≥1, read((A, N), I, Max), new1(I, N, A, Max).
6. new1(I, N, A, Max) :- I1=I+1, I<N, I≥0, M>Max,
                         read((A, N), I, M), new1(I1, N, A, M).
7. new1(I, N, A, Max) :- I1=I+1, I<N, I≥0, M≤Max,
                         read((A, N), I, M), new1(I1, N, A, Max).
8. new1(I, N, A, Max) :- I≥N, K≥0, N>K, Z>Max, read((A, N), K, Z).
```

As indicated in [7], in order to propagate the error property, we 'reverse' the derived program $T1$ and we get the following program $T1_{rev}$:

```
rev1. incorrect :- b(U), r2(U).
rev2. r2(V) :- trans(U, V), r2(U).
rev3. r2(U) :- a(U).
```

where the predicates `a`, `b`, and `trans` are defined as follows:

```
s4. a([new1, I, N, A, Max]) :- I=0, N≥1, read((A, N), I, Max)
```

s5. `trans([new1, I, N, A, Max], [new1, I1, N, A, M]) :-`
`I1 = I+1, I < N, I ≥ 0, M > Max, read((A, N), I, M).`

s6. `trans([new1, I, N, A, Max], [new1, I1, N, A, Max]) :-`
`I1 = I+1, I < N, I ≥ 0, M ≤ Max, read((A, N), I, M).`

s7. `b([new1, I, N, A, Max]) :- I ≥ N, K ≥ 0, K < N, Z > Max, read((A, N), K, Z).`

The transformation from $T1$ to $T1_{rev}$ is correct in the sense that `incorrect` $\in M(T1)$ iff `incorrect` $\in M(T1_{rev})$. This equivalence holds because: (i) in program $T1$ the predicate `incorrect` is defined in terms of the predicate `new1` that encodes the reachability relation from an error configuration to an initial configuration, and (ii) in program $T1_{rev}$ the predicate `incorrect` is defined in terms of the predicate `r2` that also encodes the reachability relation, but this time the encoding is 'in the reversed direction', that is, from an initial configuration to an error configuration.

Now let us apply Step (B) of our verification method starting from the program $T1_{rev}$.

Unfolding. First we unfold clause rev1 w.r.t. the atom `b(U)`, and we get:

9. `incorrect :- I ≥ N, K ≥ 0, K < N, Z > Max,`
`read((A, N), K, Z), r2([new1, I, N, A, Max]).`

Neither Goal Replacement nor Clause Removal is applied.

Definition & Folding. In order to fold clause 9 we introduce the following clause:

10. `new2(I, N, A, Max, K, Z) :- I ≥ N, K ≥ 0, K < N, Z > Max,`
`read((A,N), K, Z), r2([new1, I, N, A, Max]).`

By folding clause 9 using clause 10, we get:

11. `incorrect :- I ≥ N, K ≥ 0, K < N, Z > Max, new2(I, N, A, Max, K, Z).`

Now we proceed by performing a second iteration of the body of the while-loop of the *Transform* strategy because *InDefs* is not empty (indeed, clause 10 belongs to *InDefs*).

Unfolding. After some unfoldings from clause 10 we get the following clauses:

12. `new2(I1, N, A, M, K, Z) :- I1 = I+1, N = I1, K ≥ 0, K < I1, M > Max, Z > M,`
`read((A, N), K, Z), read((A, N), I, M), r2([new1, I, N, A, Max]).`

13. `new2(I1, N, A, Max, K, Z) :- I1 = I+1, N = I1, K ≥ 0, K < I1, M ≤ Max, Z > Max,`
`read((A, N), K, Z), read((A, N), I, M), r2([new1, I, N, A, Max]).`

Goal Replacement. We use the following law which is a consequence of the fact that arrays are finite functions:

(GR) `read((A, N), K, Z), read((A, N), I, M)` $\leftrightarrow$
`(K = I, Z = M, read((A, N), K, Z))` $\vee$ `(K ≠ I, read((A, N), K, Z), read((A, N), I, M))`

Thus, (i) we replace the conjunction of atoms '`read((A,N),K,Z),read((A,N),I,M)`' occurring in the body of clause 12 by the right hand side of law (GR), and then (ii) we split the derived clause with disjunctive body into the following two clauses, each of which corresponds to a disjunct of the right hand side of (GR). We get the following clauses:

12.1 `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
`K=I, M=Z, read((A,N),K,Z), r2([new1,I,N,A,Max]).`
12.2 `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
`K≠I, read((A,N),K,Z), read((A,N),I,M), r2([new1,I,N,A,Max]).`

Clause Removal. The constraint '`Z>M,M=Z`' in the body of clause 12.1 is unsatisfiable. Hence, this clause is removed from *TranfP*. By simplifying the constraints in clause 12.2 we get:

14. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M>Max, Z>M,`
`read((A,N),K,Z), read((A,N),I,M), r2([new1,I,N,A,Max]).`

By applying similar goal replacements and clause removals, from clause 13 we get:

15. `new2(I1,N,A,Max,K,Z) :- I1≤I+1, N=I1, K≥0, K<I, M≤Max, Z>Max,`
`read((A,N),K,Z), read((A,N),I,M), r2([new1,I,N,A,Max]).`

Definition & Fold. In order to fold clause 14, we introduce the following definition:

16. `new3(I,N,A,Max,K,Z) :- K≥0,K<N,K<I,Z>Max,`
`read((A,N),K,Z)), r2([new1,I,N,A,Max]).`

Clause 16 is obtained from clauses 10 and 14 by applying a generalization operator called *WidenSum* [13], which is a variant of the classical widening operator [5]. Clause 16 can be used also for folding clause 15, and by folding clauses 14 and 15 using clause 16, we get:

17. `new2(I1,N,A,Max,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M>Max, Z>M,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`
18. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M≤Max, Z>Max,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`

Now we perform the third iteration of the body of the while-loop of the strategy. After some unfolding, goal replacement, clause removal, and folding steps, from clause 16 we get:

19. `new3(I1,N,A,M,K,Z) :- I1=I+1, K≥0, K+1<I1, N≥I1, M>Max, Z>M,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`
20. `new3(I1,N,A,Max,K,Z) :- I1=I+1, K≥0, K+1<I1, N≥I1, M≤Max,`
`Z>Max, read((A,N),I,M), new3(I,N,A,Max,K,Z).`

Since we did not introduce any new definition, and no clause remains to be processed (indeed, the set *InDefs* of definitions is empty), the *Transform* strategy exits the while-loop and we get the program consisting of the set $\{11, 17, 18, 19, 20\}$ of clauses.

Since no clause in this set is a constrained fact, by the final phase of removing the useless clauses we get a final program consisting of the empty set of clauses. Thus, the program *ArrayMax* is correct with respect to the given φ_{init} and φ_{error} properties.

5 Related Work and Conclusions

The verification method presented in this paper is an extension of the one introduced in [7], where Constraint Logic Programming (CLP) and iterated specialization have been used to define a general verification framework that is parametric with respect to the programming language and the logic used for specifying the correctness properties. The main novelties of this paper are the following ones: (i) we have considered imperative programs acting on integer variables as well as array variables, and (ii) we have allowed a more expressive specification language, in which one can write properties about elements of arrays and, in general, elements of complex data structures.

In order to deal with this more general setting, we have defined the operational semantics of array manipulation, and we have also considered powerful transformation rules, such as conjunctive definition, conjunctive folding, and goal replacement. These transformation rules together with some strategies for guiding their application, have been implemented in the MAP transformation system [25], so that the proofs of program correctness have been performed in a semi-automatic way.

The use of constraint-based techniques for program verification is not novel. Indeed, CLP programs have been successfully applied to perform model checking of both finite and infinite state systems [9, 11, 13] because through CLP programs one can express in a simple manner both (i) the symbolic executions of imperative programs and (ii) the invariants which hold during their executions. Moreover, there are powerful CLP-based tools, such as ARMC [31], TRACER [18], and HSF [16], that can be used for performing model checking of imperative programs. These tools are fully automatic, but they are applicable to classes of programs and properties that are much more limited than those considered in this paper. We have shown in [7] that, by focusing on verification tasks similar to those considered by ARMC, TRACER, and HSF, we can design a fully automatic, transformation-based verification technique whose effectiveness is competitive to the one of the above mentioned tools.

Our rule-based program transformation technique is also related to *conjunc-*

tive partial deduction (CPD) [8], a technique for the specialization of logic programs with respect to conjunctions of atoms. There are, however, some substantial differences between CPD and the approach we have presented here. First, CPD is not able to specialize logic programs with constraints and, thus, it cannot be used to prove the correctness of the *GCD* program where the role of constraints is crucial. Indeed, using the ECCE conjunctive partial deduction system [23] for specializing the program consisting of clauses {3, 4, 5, 8r, 9, 10, 11r} with respect to the query `incorrect`, we obtain a residual program where the predicate `incorrect` is not useless. Thus, we cannot conclude that the atom `incorrect` does not belong to the least model of the program, and thus we cannot conclude that the program is correct. One more difference between CPD and our technique is that we may use goal replacement rules which allow us to evaluate terms over domain-specific theories. In particular, we can apply the goal replacement rules using well-developed theories for data structures like arrays, lists, heaps and sets (see [3, 26, 15, 2, 33, 36] for some formalizations of these theories).

An alternative, systemic approach to program transformation is supercompilation [35], which considers programs as machines. A supercompiler runs a program and, while it observes its behavior, produces an equivalent program without performing stepwise transformations of the original program.

The verification method presented in this paper is also related to several other methods for verifying properties of imperative programs acting on arrays. Those methods use techniques based on abstract interpretation, theorem proving and, in particular, Satisfiability Modulo Theory (see, for instance, [28, 21, 22]).

The application of the powerful transformation rules we have considered in this paper enables the verification of a larger class of properties, but it does not entirely fit into the automated strategy used in [7]. In the future we intend to consider the issue of designing fully mechanizable strategies for guiding the application of our program transformation rules. In particular, we want to study the problem of devising suitable unfolding strategies and generalization operators, by adapting the techniques already developed for program transformation. We also envisage that the application of the laws used by the goal replacement rule can be automated by importing in our framework the techniques used in the fields of Theorem Proving and Term Rewriting. For some specific theories we could also apply the goal replacement rule by exploiting the results obtained by external theorem provers or Satisfiability Modulo Theory solvers.

We also plan to address the issue of proving correctness of programs acting on *dynamic data structures* such as lists or heaps, looking for a set of suitable goal replacement laws which axiomatize those structures.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments and constructive criticism.

References

[1] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI '07*, volume 4349 of *LNCS*, pages 378–394. Springer, 2007.

[2] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42. Springer-Verlag New York, Inc., March 1987.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI'06*, volume 3855 of *LNCS*, Charleston, SC, USA, January 8-10 2006. Springer.

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 4th ACM-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978.

[7] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM '13*, pages 43–52, 2013.

[8] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.

[9] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 223–239. Springer-Verlag, 1999.

[10] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[11] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM SIGPLAN Workshop on Verification and Computational Logic VCL'01, Florence*

(Italy), pages 85–96, University of Southampton, UK, 2001. Technical Report DSSE-TR-2001-3.

[12] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. In G. Delzanno and I. Potapov, editors, *Proceedings of the 5th International Workshop on Reachability Problems (RP 2011), September 28-30, 2011*, volume 6945 of *LNCS*, pages 165–179, Genova, Italy, 2011. Springer.

[13] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming. Special Issue on the 25th Annual GULP Conference*, 13(2):175–199, 2013.

[14] C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1–3):253–270, 2004.

[15] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3–4):231–254, 2007.

[16] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. Hsf(c): A software verifier based on horn clauses. In *Proc. of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '12*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[17] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[18] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV'12*, volume 5732 of *LNCS*, pages 758–766. Springer, 2012.

[19] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[21] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering. FASE 2009*, volume 5503 of *LNCS*, pages 470–485, 2009.

[22] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013, Rome, Italy, January 20-22, 2013*, volume 7737 of *LNCS*, pages 169–188, 2013.

[23] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks, release 3, November 2000. Available from `http://www.ecs.soton.ac.uk/~mal`.

[24] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–

515, 2002.

[25] MAP. The MAP transformation system, 2000. `http://www.iasi.cnr.it/~proietti/system.html`. Also available via a WEB interface from `http://www.map.uniroma2.it/mapweb`.

[26] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70, North-Holland, 1963.

[27] J. McCarthy. Towards a mathematical science of computation. In C. Popplewell, editor, *Information Processing. Proceedings of IFIP 1962*, pages 21–28, Amsterdam, North-Holland, 1963.

[28] R. Cousot P. Cousot and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM Symposium on Principles of programming languages. POPL'11*, pages 105–118, 2011.

[29] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17–20, 2002, Revised Selected Papers*, volume 2664 of *LNCS*, pages 90–108, 2003.

[30] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, editor, *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998*, volume 1503 of *LNCS*, pages 246–261. Springer, 1998.

[31] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In M. Hanus, editor, *Practical Aspects of Declarative Languages, PADL '07*, volume 4354 of *LNCS*, pages 245–259. Springer, 2007.

[32] C. J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS'02*, pages 55–74. IEEE Computer Society, 2002.

[34] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming (ICLP'84)*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.

[35] V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

[36] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, 1990.

Ranking Functions for Linear-Constraint Loops

Amir M. Ben-Amram

The Academic College of Tel-Aviv Yaffo
amirben@mta.ac.il

Abstract

Ranking functions are a tool successfully used in termination analysis, complexity analysis, and program parallelization. Among the different types of ranking functions and approaches to finding them, this talk will concentrate on functions that are found by linear programming techniques. The setting is that of a loop that has been pre-abstracted so that it is described by linear constraints over a finite set of numeric variables. I will review results (more or less recent) regarding the search for ranking functions which are either linear or lexicographic-linear.

1 Ranking Functions

Consider a program, viewed abstractly as a "single step" relation $\vdash$ mapping a state s to the next state s'. Thus a computation of the program is a chain $s_0 \vdash s_1 \vdash s_2 \vdash \dots$. In termination analysis, we wish to prove that there are no infinite chains. The *ranking function method* is to find a function ρ that maps program states into a well-founded ordered set W, such that $\rho(s) > \rho(s')$ whenever $s \vdash s'$. Well-founded orders have no infinite strictly-descending chains, so all computations must be finite. In practice, the granularity of the "next step" relation may vary. In many applications, the step in question represents a single application of a loop's body.

The uses of a ranking function are several. Termination *per se* is a basic concern in program verification. Secondly, in program transformation, one wants to ensure that generated code does not breach termination properties, and moreover, that the transformation process itself (e.g., a partial evaluation) is terminating. When the co-domain of the ranking function is the non-negative integers, the function's value on entrance to the loop bounds the number of loop iterations. Thus, such ranking functions are useful in time-complexity analysis [1, 20]. More complex ranking functions, in particular, *lexicographic*, can also be used for this purpose [3]. An estimate of the running time of a program (or just one loop) can be used in several program transformation tasks, such as optimization and parallelization. Some of the techniques mentioned below

have been developed for loop parallelization, concurrently or even before their adoption in termination analysis [16, 17].

The main sources for more details on results mentioned in this talk are [4] which summarizes several results on linear ranking functions, [3] on lexicographic ranking functions, and, for recent results, my joint work with Samir Genaim [5, 6].

2 Program Representation

Very often, to generate ranking functions, the semantics of the loop is represented (possibly over-approximated) by linear constraints. A *single-path* linear-constraint loop (*SLC* for short) over variables $x_1, \ldots, x_n$ has the form

$$\textit{while } (B\mathbf{x} \leq \mathbf{b}) \textit{ do } A \begin{pmatrix} \mathbf{x} \\ \mathbf{x}' \end{pmatrix} \leq \mathbf{c} \tag{1}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are column vectors, and for some $p, q > 0$, $B \in \mathbb{Q}^{p\times n}$, $A \in \mathbb{Q}^{q\times 2n}$, $\mathbf{b} \in \mathbb{Q}^p$, $\mathbf{c} \in \mathbb{Q}^q$. Note that the coefficients are rationals. The loop variables range over the integers or over the rationals[1]. For notational convenience, we let $\mathbf{x}''$ stand for $\left(\begin{smallmatrix} \mathbf{x} \\ \mathbf{x}' \end{smallmatrix}\right)$. In (1), the constraint $B\mathbf{x} \leq \mathbf{b}$ is the *loop guard*, and $a\mathbf{x}'' \leq \mathbf{c}$ is the *loop update*. A case of importance is *deterministic linear update*, which can be expressed as $\mathbf{x}' = A'\mathbf{x} + \mathbf{c}'$ for $A', \mathbf{c}'$ of appropriate dimensions.

The transition relation of an *SLC* loop is a convex polyhedron in $\mathbb{Q}^{2n}$, usually denoted by $\mathcal{Q}$ (for definitions of convex polyhedra and their essential properties, see [22]).

Whereas the loop update above is a conjunction of inequalities, a *multipath* linear-constraint loop (*MLC* for short) is described by a disjunction of such conjunctions. It is written as

$$\textit{loop} \bigvee_{i=1}^{k} \left[B_i\mathbf{x} \leq \mathbf{b}_i \,\wedge\, A_i \begin{pmatrix} \mathbf{x} \\ \mathbf{x}' \end{pmatrix} \leq \mathbf{c}_i \right] \tag{2}$$

Or (which amounts to the same) as a list $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ of transition polyhedra. The *MLC* form more faithfully represents loops that have a body which is not straight-line but involves branching, or where a non-linear operation has been abstracted to a disjunction of linear constraints.

A third form of program representation, generalizing *MLC* loops, is a control-flow graph (CFG) annotated with linear constraints on its arcs, as used in [3] (and, with somewhat different terminology, in [18, 23]). Generally speaking, one

[1] assuming that the variables range over the reals makes no difference from the rational case in the problems considered here.

may say that this form is used when a whole program module is translated to a constraint representation before calling the termination algorithm (examples of tools that has this flow are JULIA [24] for Java bytecode programs, Termilog [15] and Terminweb [10] for Prolog). On the other hand, in a tool such as Terminator [14], only *SLC* loops are used: basically, the tool looks for potentially non-terminating cycles, and applies the ranking-function generation algorithm to one at a time. COSTA [2] handles one *MLC* loop at a time. In the next two sections, we concentrate on *SLC* and *MLC* loops.

3 Linear Ranking Functions

A *linear ranking function* (*LRF*) has the form $\rho(\mathbf{x}) = \vec{\lambda} \cdot \mathbf{x} + \lambda_0$, with $(\lambda_0, \vec{\lambda}) \in \mathbb{Q}^{n+1}$, and is required to satisfy: if $\mathbf{x} \vdash \mathbf{x}'$, then $\rho(\mathbf{x}) \geq \rho(\mathbf{x}') + 1$ and $\rho(\mathbf{x}) \geq 0$. Written explicitly, the conditions are:

$$\vec{\lambda} \cdot \mathbf{x} + \lambda_0 \geq 0 \tag{3}$$

$$\vec{\lambda} \cdot (\mathbf{x} - \mathbf{x}') \geq 1 \tag{4}$$

In general, the co-domain of ρ will be $\mathbb{Q}$, but to justify its use as a termination proof, one can convert it to a function with co-domain ω, namely $1+\max(\rho(\mathbf{x}), 0)$.

The decision problem—does a *LRF* exist for a given linear-constraint loop? Will be denoted by either LINRF($\mathbb{Q}$) or LINRF($\mathbb{Z}$), depending on whether the variables are allowed to assume any rational value satisfying the constraints, or just integer ones. To the problem of computing an explicit representation of the ranking function, if one exists, we refer as the *synthesis* problem.

THEOREM 3.1. LINRF($\mathbb{Q}$) $\in$ *PTIME, and LRF synthesis can be done in polynomial time, for both SLC and MLC loops.*

This is perhaps the best-known result in this area. This result is based on using Farkas' lemma to transform the search for a *LRF* into a linear programming problem. Such a solution has been found by multiple researchers in different places and times and in some alternative versions: Sohn and van Gelder [23] did it (for *MLC* loops, restricted to non-negative integers) in termination analysis of logic programs; Feautrier [16], for *MLC* loops, for scheduling parallel computations; Podelski and Rybalchenko [21], for *SLC* loops, in termination analysis of imperative programs; finally, Mesnard and Serebrenik [18] extended Sohn and van Gelder's solution to the rationals. A related technique, in [11], is based on similar considerations but is not polynomial-time.

The fact that this solution is incomplete when the true domain of the variables is $\mathbb{Z}$ is illustrated by examples of loops which terminate (and have a *LRF*)

over the integers, whereas over the rationals they are not even terminating. One such loop is

$$\begin{array}{l} \textit{while } (x_2 - x_1 \le 0, x_1 + x_2 \ge 1) \textit{ do} \\ \qquad\qquad x_2' = x_2 - 2x_1 + 1, x_1' = x_1 \end{array} \tag{5}$$

This discrepancy has been noted, for example in [9, 13, 16]; the latter two point out that the discrepancy disappears if all polyhedra in the program are integral. An integral polyhedron is equal to the convex hull of its integer points [22]. The reason that this works is, intuitively, *convexity*: a function that satisfies conditions (3,4) for a given set V of points $\mathbf{x}''$ also satisfies them throughout the covex hull of V. Hence, a complete and sound, but unfortunately exponential, algorithm to decide LinRF($\mathbb{Z}$) is to compute the *integer hull* of $\mathcal{Q}$ (or each $\mathcal{Q}_i$), and proceed to solve LinRF($\mathbb{Q}$). In [6], we undertook a closer study of the complexity of LinRF($\mathbb{Z}$). The main results follow.

THEOREM 3.2. LinRF($\mathbb{Z}$) $\in$ coNP *(referring to MLC loops); this problem is strongly coNP-hard, even for deterministic SLC loops.*

An outline of the proofs. *coNP-hardness* follows from the hardness of the problem: given a polyhedron (in constraint representation), does it contain any integer point? The reduction constructs a loop that has a *LRF* if and only if the given polyhedron has no integer points.

For *inclusion in coNP*, we have to show that *non-existence* of a *LRF* is a property for which there is a polynomially-verifiable witness. For simplicity, consider a *SLC* loop $\mathcal{Q}$. We start by observing that a single transition $\mathbf{x}'' \in \mathcal{Q}$ that does not satisfy (3,4) for a given vector $(\lambda_0,\vec{\lambda})$ suffices to show that $(\lambda_0,\vec{\lambda})$ are not the coefficients of a ranking function. We thus define $W(\mathbf{x}'')$ to be the set of coefficient vectors $(\lambda_0,\vec{\lambda}) \in \mathbb{Q}^{n+1}$ that do not satisfy (3,4); we say that they are *witnessed against* by $\mathbf{x}''$. It immediately follows that there is no *LRF* for $\mathcal{Q}$ if and only if $\bigcup_{\mathbf{x}''\in\mathcal{Q}} W(\mathbf{x}'') = \mathbb{Q}^{n+1}$. But this is not an effective condition, since there may be infinitely integer points in $\mathcal{Q}$.

The goal now is to obtain a finite (and polynomial) witness set. To this end, we rely on convexity arguments, and on the *generator representation* of a polyhedron $\mathcal{Q}$. The generator representation consists of *vertices*, which are points of the polyhedron, and *rays*, also known as *recession directions*; a ray $\mathbf{y}'' \in \mathbb{Q}^{2n}$ is a vector such that for $\mathbf{x}'' \in \mathcal{Q}$, $\mathbf{x}'' + a\mathbf{y}''$ is in $\mathcal{Q}$ for all $a \ge 0$. The set of rays is denoted by $\mathcal{R}_{\mathcal{Q}}$. We call such a vector $\mathbf{y}''$ a *homogenous witness* (h-witness for short) against $(\lambda_0,\vec{\lambda})$ if one of the following requirements is not

satisfied:

$$\vec{\lambda} \cdot \mathbf{y} \geq 0 \tag{6}$$

$$\vec{\lambda} \cdot (\mathbf{y} - \mathbf{y}') \geq 0 \tag{7}$$

It can be shown that in this case, $(\lambda_0,\vec{\lambda})$ cannot be the coefficient vector of a ranking function. Now, our witness against the existence of a *LRF* is a pair X, Y of sets, such that $X \subseteq \mathcal{Q}, Y \in \mathcal{R}_{\mathcal{Q}}$, $X \neq \emptyset$ and X, Y witness, all together, against every possible coefficient vector. To verify this we only need to verify that the witnesses are kosher (i.e., inclusion in $\mathcal{Q}$ and $\mathcal{R}_{\mathcal{Q}}$, respectively), and that there is no solution to the combined set of requirements (i.e., (3,4) for the members of X and (6,7) for members of Y). This is a polynomial-time procedure, if the size of X and Y (in bits) is polynomial. Fortunately, the existence of a polynomial-size witness set can be guaranteed; basically, we show that a polynomially big subset of the generators constitutes such a witness set.

Synthesis. By computing the generator representation of a polyhedron (if not available), we obtain an algorithm to find ranking functions—namely by finding a coefficient vector that none of the generators witnesses againt. Computation of the generators is, in general, of exponential time and space complexity. The resulting ranking-function coefficients are, however, of polynomial bit-size, a result based on the considerations involved in proving the inclusion of the decision problem in coNP. Interestingly, this sheds a new light on an algorithm proposed in [9], which searches for *LRF* coefficients by a kind of bisection search on the space of coefficients. Bounding the bit-size of the coefficients turns this search (which is unbounded, and hence only a semi-decision procedure) into a decision procedure.

Polynomially solvable cases. If coNP-hardness is considered as bad news, a positive news is that some special cases of interest can be solved in polynomial time. Basically, this happens when the transition polyhedra are either integral to begin with, or of such a kind that their integer hull is easy to compute. In [6] we characterize some benign cases, including loops in which the body is a sequence of linear affine updates with integer coefficients (as in loop (5) above) and the condition is defined by either an extended form of *difference constraints*, a restricted form of *Two Variables Per Inequality* constraints, or a cone (constraints where the free constant is zero). Some cases in which the body involves linear constraints are also presented. A somewhat surprising *hardness* result is that for octagonal guards [19, 7] and deterministic udpates, LinRF($\mathbb{Z}$) is already coNP-hard.

4 Lexicographic-Linear Ranking Functions

A *lexicographic-affine* ranking function (*LLRF*) has the form $\tau(\mathbf{x}) = \langle \rho_1(\mathbf{x}), \dots, \rho_d(\mathbf{x}) \rangle$ where each $\rho_i(\mathbf{x}) = \lambda^{(i)} \cdot \mathbf{x} + \lambda_0^{(i)}$. It has to descend in lexicographic order. The constant d is called the *dimension* of the function.

Lexicographic ranking functions are a very natural concept: the oldest example of a (manual) termination proof using ranking functions [25] uses lexicographic descent. Several automatic methods of finding *LLRF*s have appeared, and we could point out that any method that finds a *LRF* for each loop in a nested set of loops implicitly constructs a *LLRF*. An algorithm in the style of Terminator [14] can also arrive at a lexicographic ranking function, but implicitly and in a roundabout way.

Before discussing algorithms and complexity, there are some things to note, and first the very definition, as there are different variants of the concept. Our definition follows. We use the notation $\Delta\rho(\mathbf{x}'')$ for $\rho(\mathbf{x}) - \rho(\mathbf{x}')$.

Definition 4.1. Let $\tau = \langle \rho_1, \dots, \rho_d \rangle$. We say that τ is a *LLRF* for a set $T \subseteq \mathbb{Q}^{2n}$ of transitions if and only if for every $\mathbf{x}'' \in T$ there exists $i \le d$ for which the following hold:

$$\forall j < i \ . \ \Delta\rho_j(\mathbf{x}'') \ge 0 \tag{8}$$

$$\forall j \le i \ . \ \rho_j(\mathbf{x}) \ge 0 \tag{9}$$

$$\Delta\rho_i(\mathbf{x}'') \ge 1 \tag{10}$$

When this holds, we write $\tau(\mathbf{x}) \succ_{lex} \tau(\mathbf{x}'')$.

As for *LRFs*, the co-domain of the function is not really a well-founded set, but it can be easily converted to a function into ω^d in order to prove termination. We note that the definition of [3] is more restrictive since it requires (9) to hold for all $1 \le j \le d$. In contrast the definition in [8] is more general since it requires (9) to hold only for $j = i$. Examples can be given to show that these three classes of function are really different. In terms of complexity, [3] give a polynomial-time algorithm, based on repeated finding of *LRFs* using linear programming, whereas [8] use a (possibly exponential) combinatorial search. Our work in [5] is inspired by [3], but extends it in certain ways: most importantly, we consider the complexity of the problem when the variables are integer (that is, the set T of transitions is specified as the integer points in a set $\mathcal{Q}_1, \dots, \mathcal{Q}_k$ of transition polyhedra).

In fact, [3] includes a (quite clever) completeness proof, but as for previous *LRF* generation procedures, this completeness is for rational-valued state space. Even over the rationals, their solution is incomplete when referring to our more flexible class of functions. An interesting property of our class is that an *SLC*

loop may require a lexicographic ranking function, as the following example demonstrates:

$$\textit{while}(x_1 \geq 0, x_2 \geq 0, x_3 \geq -x_1)\ \textit{do}\ x_2' = x_2 - x_1, x_3' = x_3 + x_1 - 2\,. \tag{11}$$

It has a *LLRF* $\tau_1 = \langle x_2, x_3 \rangle$ as in Definition 4.1 (over both rationals and integers), while according to the more restricted definition in [3], it has no *LLRF* at all.

As for linear ranking functions, we consider both the complexity of the decision problem over the integers (denoted LexLinRF($\mathbb{Z}$)), and the synthesis problem. Our main results are:

THEOREM 4.2. LexLinRF($\mathbb{Z}$) *is coNP-complete.*

THEOREM 4.3. *For synthesis of LLRFs, when variables range over the integers, there is a complete synthesis algorithm that runs in polynomial time,* given integer polyhedra *(otherwise, integer hulls have to be computed, which increases the running time to exponential.)*

An essential component in our proofs is the notion of a *quasi-LRF*. We say that ρ is a quasi-*LRF* for a set T of transitions if for every $\mathbf{x}'' \in T$ the following holds:

$$\rho(\mathbf{x}) \geq 0 \tag{12}$$

$$\Delta\rho(\mathbf{x}'') \geq 0 \tag{13}$$

We say that it is a *non-trivial* if, in addition, $\Delta\rho(\mathbf{x}'') > 0$, for at least one $\mathbf{x}'' \in X$.

Our synthesis algorithm constructs a *LLRF* $\langle \rho_1(\mathbf{x}), \ldots, \rho_d(\mathbf{x}) \rangle$ in this way: first a non-trivial quasi-*LRF* ρ_1 is found; then the transitions where ρ_1 strictly descends are eliminated; if some are left, the algorithm is repeated to generate the next components. The search for a quasi-*LRF* uses linear programming, resembling the *LRF* procedures based on the Farkas lemma. The passage to the next iteration is based on the fact that the *set of transitions where ρ_1 does not descend, out of a transition polyhedron $\mathcal{Q}_i$, is either empty or a face of $\mathcal{Q}_i$.* This leads to an efficient algorithm to compute the polyhedra for the next iteration, keeping them integral, and not blowing up the size of the coefficients.

For the *decision problem*, our results rest on the following proposition, which in turn is proved (in part) using the algorithm:

PROPOSITION 4.4. *There is no LLRF for a set of transitions $T \subseteq \mathbb{Z}^{2n}$ if and only if there is $W \subseteq T$ for which there is no non-trivial quasi-LRF.*

Now, inclusion in coNP follows by showing that non-existence of a non-trivial quasi-*LRF* has a polynomially checkable witness. The considerations in proving this claim are similar to those outlines for *LRF*s, involving witnesses which are vertices and h-witnesses which are rays, though slightly more complicated since the witness has to rule out only non-trivial quasi-*LRF*s (a trivial quasi-*LRF*, namely $\rho(\mathbf{x}) = 0$, always exists, and there may be others).

Further observations. Two interesting results that followed from our investigation are these:

- The dimension d of our ranking functions is always at most n, for an *MLC* loop of any number of alternatives.
- If an *SLC* loop has such a ranking function, its number of iterations can be linearly bounded (more precisely, it is linear in the absolute values of the variables in the initial state) even if the dimension of the ranking function is larger than 1.

5 General Control-Flow Graphs

MLC loops can be extended to general CFGs annotated with linear constraints. This is the program abstraction used in [3, 18, 12, 23]. In [3], termination certificates for such programs consists of a lexicographic affine function ρ_ℓ associated with every program location ℓ, such that in a transition $s \vdash s'$ from location ℓ to ℓ', we have $\rho_\ell(s) \succ_{lex} \rho_{\ell'}(s')$. As their algorithm shows (and also some previous works), the generalization from finding a *LLRF* for an *MLC* loop to this model is quite smooth, and our results transfer in the same way.

A predecessor to [3], in a sense, is the algorithm in [12] (extending [11]). It is somewhat similar in structure to that of [3] (and also ours), finding linear quasi-ranking functions one by one, but it does not construct *LLRF*s explicitly, and its space of ranking functions is more restricted because it handles a strongly-connected component S of the program by looking for a ρ which is bounded and non-increasing throughout the component (that is, ρ_ℓ is the same for all $\ell \in S$). However, we may note that the last difference disappears if one considers only *MLC* loops.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2010.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, FMCO'07*, volume 5382 of *LNCS*, pages 113–132. Springer, 2007.

[3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In R. Cousot and M. Martel, editors, *Static Analysis Symposium, SAS'10*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.

[4] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.*, 215:47–67, 2012.

[5] A. M. Ben-Amram and S. Genaim. The linear ranking problem for integer linear-constraint loops. Technical report, 2013.

[6] A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In *Proc. of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'13*, pages 51–62, NY, USA, 2013. ACM.

[7] M. Bozga, C. Gîrlea, and R. Iosif. Iterating octagons. In S. Kowaleski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 337–351. Springer, 2009.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, pages 491–504. Springer, 2005.

[9] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In M. Abadi and L. de Alfaro, editors, *Concurrency Theory, CONCUR 2005*, volume 3653 of *LNCS*, pages 488–502. Springer, 2005.

[10] M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999. Preliminary (conference) version in LNCS 1298 (1997).

[11] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.

[12] M. Colón and H. Sipma. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.

[13] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS'10*, volume 6015 of *LNCS*, pages 236–250. Springer, 2010.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *Programming Language Design and Implementation, PLDI'06*, pages 415–426. ACM, 2006.

[15] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1–2):117–156, 2001.

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem. II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[18] F. Mesnard and A. Serebrenik. Recurrence with affine level mappings is p-time decidable for clp(r). *TPLP*, 8(1):111–119, 2008.

[19] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, March 2006.

[20] R. Peña and A. D. Delgado-Muñoz. Size invariant and ranking function synthesis in a functional language. In *Functional and Constraint Logic Programming*, volume 6816 of *LNCS*, pages 52–67. Springer, 2011.

[21] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI'04*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[22] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.

[23] K. Sohn and A. V. Gelder. Termination detection in logic programs using argument sizes. In D. J. Rosenkrantz, editor, *Symposium on Principles of Database Systems*, pages 216–226. ACM Press, 1991.

[24] F. Spoto, F. Mesnard, and E. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):1–70, 2010.

[25] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1948. reprinted in: The early British computer conferences, vol. 14 of Charles Babbage Institute Reprint Series For The History Of Computing, MIT Press, 1989.

Transforming Undecidable Synthesis Problems into Decidable Problems

Bernd Finkbeiner

Universität des Saarlandes
finkbeiner@cs.uni-saarland.de

Synthesis holds the promise to revolutionize the development of complex systems by automating the translation from specifications to implementations. Synthesis algorithms are based on the same level of mathematical rigor as verification algorithms but can be applied at earlier development stages, when only parts of the design are available. Given a formal specification of the desired system properties, for example in a temporal logic, we determine if the partial design can be completed into a full design that satisfies the properties.

For general distributed systems, the synthesis problem is undecidable. However, there has been a sequence of discoveries where the decidability was established for specific system architectures, such as pipelines and rings, or other restrictions on the problem, such as local specifications. Encouraged by these findings, new specification languages like Coordination Logic aim for a uniform treatment of the synthesis problem.

In this talk, I will review several techniques that transform undecidable synthesis problems into decidable problems. Compositional synthesis uses a proof rule to reduce an undecidable synthesis problem into several decidable synthesis problems. Bounded synthesis transforms the synthesis problem into a decidable search problem by introducing a bound on the size of the implementation. Lazy synthesis transforms the synthesis problem into a sequence of constraint solving problems, each decidable but increasingly complex, until an implementation is found.

On the Termination of Higher-Order Positive Supercompilation

G.W. Hamilton

School of Computing and Lero
Dublin City University
Ireland
hamilton@computing.dcu.ie

Abstract

The verification of program transformation systems requires that we prove their termination. For positive supercompilation, ensuring termination requires the memoisation of expressions which are subsequently used to determine when to perform generalization and folding. For a first-order language, it is sufficient to memoise only those expressions immediately prior to a function unfolding step. However, for a higher-order language, this is not sufficient to ensure termination, so more expressions need to be memoised. Determining which additional expressions to memoise can greatly affect the results obtained. Memoising too many expressions requires a lot more expensive checking for the possibility of generalization or folding; more new functions will also be created and generalization will be performed more often, resulting in less improved residual programs. We would therefore like to memoise as few expressions as possible while still ensuring termination. In this paper, we describe a simple pre-processing step which can be applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there must be function unfolding. We prove, for programs that have been pre-processed in this way, that it is only necessary to memoise expressions immediately before function unfolding to ensure termination, and we demonstrate this on a number of tricky examples.

1 Introduction

Supercompilation is a program transformation technique for functional languages which can be used for program specialization and for the removal of intermediate data structures. Supercompilation was originally devised by Turchin in what was then the USSR in the early 1970s, but did not become widely known to the outside world until over a decade later. One reason for this delay was that the work was originally published in Russian in journals which were not accessible to the outside world; it was eventually published in mainstream journals

much later [23, 24]. Another possible reason why supercompilation did not become more widely known much earlier is that it was originally formulated in the language Refal, which is rather unconventional in its use of a complex pattern matching algorithm. This meant that Refal programs were hard to understand, and describing transformations making use of this complex pattern matching algorithm made the descriptions quite inaccessible. This problem was overcome by the development of *positive supercompilation* [19, 22], which is defined over a more familiar functional language.

Ensuring the termination of positive supercompilation requires the memoisation of expressions, and then using these memoised expressions to determine when to perform generalization and folding. Positive supercompilation was originally formulated for a first-order language, so it was sufficient to memoise only the expressions immediately prior to function unfolding to ensure termination since in any potentially infinite sequence of transformation steps there must be an unfolding. However, this is not sufficient to ensure termination when transforming a higher-order language. For example, consider the following program:

Example 1. $(\lambda x \rightarrow x\ x)\ (\lambda x \rightarrow x\ x)$

When this program is transformed there will be a potentially infinite sequence of transformation steps without any unfolding. Although this expression would not be accepted by most type checkers, there are also many examples of expressions which would be accepted by a type checker and which will have a potentially infinite sequence of transformation steps without any unfolding. For example, consider the following program:

Example 2. $\mathbf{data}\ D = F\ (D \rightarrow D)$

$$(\lambda f \rightarrow f\ (F\ (\lambda x \rightarrow f\ x\ x))\ (F\ (\lambda x \rightarrow f\ x\ x)))$$
$$(\lambda y \rightarrow \mathbf{case}\ y\ \mathbf{of}\ F\ g \rightarrow g)$$

This program will also produce a potentially infinite sequence of transformation steps without any unfolding when transformed.

To avoid this potential non-termination, some formulations of positive supercompilation for a higher-order language memoise *all* expressions [17, 2], or at least a substantial subset of them [9, 10, 11]. Memoising too many expressions requires a lot more expensive checking for the possibility of generalization or folding. Also, more new functions will be created and generalization will be performed more often, resulting in less improved residual programs.

In this paper, we describe a simple pre-processing step which can be applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there

must be an unfolding. This involves introducing names for some anonymous functions (and possibly also performing λ-lifting [7]) to ensure that only memoising expressions immediately preceding an unfold step is sufficient to ensure termination of the transformation. This pre-processing step would transform the program in Example 1 to the following:

$$f\ f\ \textbf{where}\ f = \lambda x \rightarrow x\ x$$

Thus, any potentially infinite sequence of transformation steps would have to include the unfolding of f. Applying the pre-processing transformation to the program in Example 2 would give the following:

$$\begin{array}{lll} f_1\ f_2 & & \\ \textbf{where} & & \\ f_1 & = & \lambda f \rightarrow f\ (F\ (f_3\ f))\ (F\ (f_3\ f)) \\ f_2 & = & \lambda y \rightarrow \textbf{case}\ y\ \textbf{of}\ F\ g \rightarrow g \\ f_3 & = & \lambda f \rightarrow \lambda x \rightarrow f\ x\ x \end{array}$$

Thus, any potentially infinite sequence of transformation steps would have to include the unfolding of f_2 and f_3. The new functions are introduced sparingly, so we argue that there will not be a large overhead required for the additional memoisation and comparison of expressions prior to the unfolding of these functions, and that better residual programs will be produced as a result.

The remainder of this paper is structured as follows. In Section 2, we describe the higher-order language over which the transformations are defined. In Section 3, we give our own formulation of the positive supercompilation algorithm on this language. In Section 4, we consider the different situations in which this transformation may not terminate and give examples. In Section 5, we present our pre-processing step to transform higher-order programs into a form for which positive supercompilation will be guaranteed to terminate and prove that this is the case. Section 6 concludes and considers related work.

2 Language

In this section, we describe the higher-order functional language which will be used throughout this paper. The syntax of this language is given in Fig. 1.

The intended operational semantics of the language is normal order reduction. Programs in the language consist of an expression to evaluate and a set of function definitions. An expression can be a variable, constructor application, λ-abstraction, function call, application, **case** or **let**. Variables introduced by λ-abstraction, **let** or **case** patterns are *bound*; all other variables are *free*. We use $fv(e)$ and $bv(e)$ to denote the free and bound variables respectively of expression e. We write $e_1 \equiv e_2$ if e_1 and e_2 differ only in the names of bound variables.

prog	::=	e_0 **where** $f_1 = e_1 \ldots f_n = e_n$	Program
e	::=	x	Variable
	|	$c\ e_1 \ldots e_n$	Constructor
	|	$\lambda x \ \rightarrow e$	λ-Abstraction
	|	f	Function Call
	|	$e_0\ e_1$	Application
	|	**case** e_0 **of** $p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$	Case Expression
	|	**let** $x = e_0$ **in** e_1	Let Expression
p	::=	$c\ x_1 \ldots x_n$	Pattern

Figure 1: Language Syntax

It is assumed that the input program contains no **let** expressions; these are only introduced during transformation. Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. It is assumed that the language is typed using the Hindley-Milner polymorphic typing system [16, 3] so erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x \ \rightarrow e)$ **of** $p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$ cannot occur.

Definition 2.1 (Substitution). $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ denotes a *substitution.* If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 2.2 (Renaming). $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$, where σ is a bijective mapping, denotes a *renaming.* If e is an expression, then $e\sigma = e\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ is the result of simultaneously replacing the variables $x_1, \ldots, x_n$ with the corresponding variables $x'_1, \ldots, x'_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 2.3 (Shallow Reduction Context). A shallow reduction context $\mathcal{R}$ is an expression containing a single hole $\bullet$ in the place of the redex, which can have one of the two following possible forms:

$$\mathcal{R} ::= \bullet\ e \mid (\textbf{case}\ \bullet\ \textbf{of}\ p_1 \rightarrow e_1 \mid \ \ldots \ \mid p_n \rightarrow e_n)$$

Definition 2.4 (Evaluation Context)**.** An evaluation context $\mathcal{E}$ is represented as a sequence of shallow reduction contexts (known as a *zipper* [6]), representing the nesting of these contexts from innermost to outermost within which the expression redex is contained. An evaluation context can therefore have one of the two following possible forms:

$$\mathcal{E} ::= \langle\rangle \mid \langle\mathcal{R} : \mathcal{E}\rangle$$

Definition 2.5 (Insertion into Redex)**.** The insertion of an expression e into the redex of an evaluation context κ, denoted by $\kappa{\bullet}e$, is defined as follows:

$$\begin{array}{lll}
\langle\rangle{\bullet}e & = & e \\
\langle(\bullet\ e') : \kappa\rangle{\bullet}e & = & \kappa{\bullet}(e\ e') \\
\multicolumn{3}{l}{\langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n) : \kappa\rangle{\bullet}e} \\
 & = & \kappa{\bullet}(\mathbf{case}\ e\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n)
\end{array}$$

Free variables within the expression e may become bound within $\kappa{\bullet}e$; if $\kappa{\bullet}e$ is closed then we call κ a *closing context* for e.

3 Positive Supercompilation

In this section, we give our own higher-order formulation of the positive supercompilation algorithm [22]. At the heart of the positive supercompilation algorithm are a number of *driving* rules which reduce a term (possibly containing free variables) using normal-order reduction. Function unfolding is performed as a part of this reduction process, and folding is performed upon encountering a renaming of a memoised expression. To ensure the termination of the transformation, generalization is performed when an expression is encountered which is a *homeomorphic embedding* of a memoised expression, denoted by $\lesssim$.

The homeomorphic embedding relation was derived from results by Higman [5] and Kruskal [12] and was defined within term rewriting systems [4] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [21, 20], partial evaluation [14] and partial deduction [1, 13].

Definition 3.1 (Well-Quasi Order)**.** A well-quasi order on a set S is a reflexive, transitive relation $\lesssim$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \lesssim s_j$.

This ensures that in any infinite sequence of expressions $e_0, e_1, \ldots$ there definitely exists some $i < j$ where $e_i \lesssim e_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely.

Definition 3.2 (Embedding of Expressions)**.** To define our homeomorphic embedding relation on expressions $\lesssim$, we first define a relation $\trianglelefteq$ as shown in Figure 2, where $e_1 \trianglelefteq e_2$ if e_1 is embedded in e_2 and all of the free variables within e_1 and e_2 also match up.

$$\frac{e_1 \trianglelefteq_c e_2}{e_1 \trianglelefteq e_2} \qquad \frac{e_1 \trianglelefteq_d e_2}{e_1 \trianglelefteq e_2}$$

$$x \trianglelefteq_c x \qquad f \trianglelefteq_c f$$

$$\frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq e'_i}{(c\ e_1 \ldots e_n) \trianglelefteq_c (c\ e'_1 \ldots e'_n)} \qquad \frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq_d (c\ e_1 \ldots e_n)}$$

$$\frac{e \trianglelefteq (e'\{x' \mapsto x\})}{\lambda x.e \trianglelefteq_c \lambda x'.e'} \qquad \frac{e \trianglelefteq e'}{e \trianglelefteq_d \lambda x.e'}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad e_1 \trianglelefteq e'_1}{(e_0\ e_1) \trianglelefteq_c (e'_0\ e'_1)} \qquad \frac{\exists i \in \{0,1\}.e \trianglelefteq e_i}{e \trianglelefteq_d (e_0\ e_1)}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad \forall i \in \{1 \ldots n\}.\exists \sigma.p_i \equiv (p'_i\ \sigma) \wedge e_i \trianglelefteq (e'_i\ \sigma)}{(\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n) \trianglelefteq_c (\mathbf{case}\ e'_0\ \mathbf{of}\ p'_1 \rightarrow e'_1 | \ldots | p'_n \rightarrow e'_n)}$$

$$\frac{\exists i \in \{0 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq_d (\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n)}$$

Figure 2: Homeomorphic Embedding Relation

An expression is embedded within another by this relation if either *diving* (denoted by $\trianglelefteq_d$) or *coupling* (denoted by $\trianglelefteq_c$) can be performed. Diving occurs when an expression is embedded in a sub-expression of another expression, and coupling occurs when two expressions have the same top-level construct and all the corresponding sub-expressions of the two constructs are embedded. Our version of this embedding relation extends previous versions to handle λ-abstractions and **case** expressions that contain bound variables. In these instances, the bound variables within the two expressions must also match up.

The homeomorphic embedding relation $\lesssim$ can now be defined as follows:

$$e_1 \lesssim e_2 \text{ iff } \exists \sigma . e_1 \sigma \trianglelefteq_c e_2$$

Within this relation the two expressions must be coupled but, since σ is a renaming, there is no longer a requirement that all of the free variables within the two expressions match up. Generalizing only when two expressions are coupled ensures that the result is not a variable, and there is no need for a *split* operation as used in [21].

Example 3. Some examples of homeomorphic embedding are as follows:

1.	$f\ (g\ x) \lesssim f\ (g\ y)$	6.	$f\ (g\ x) \not\lesssim g\ (f\ y)$
2.	$f\ (h\ x) \lesssim f\ (g\ (h\ y))$	7.	$g\ (h\ x) \not\lesssim f\ (g\ (h\ y))$
3.	$f\ x\ y \lesssim f\ z\ z$	8.	$f\ z\ z \not\lesssim f\ x\ y$
4.	$f\ x\ x \lesssim f\ (g\ y)\ (h\ y)$	9.	$f\ (g\ y)\ (h\ y) \not\lesssim f\ x\ x$
5.	$\lambda x.x \lesssim \lambda y.y$	10.	$\lambda x.x \not\lesssim \lambda y.x$

Theorem 3.3. *The homeomorphic embedding relation $\lesssim$ is a well-quasi-order.*

Proof. The proof is identical to that in [10]. It involves showing that there are a finite number of functors (function names and constructors) in the language. Applications of different arities are replaced with separate constructors; we prove that arities are bounded so there are a finite number of these. We also replace case expressions with constructors. Since our homeomorphic embedding relation requires that the bound variables in expressions match up, bound variables are defined using de Bruijn indices, and each of these are replaced with separate constructors; we also prove that de Bruijn indices are bounded. The overall number of functors is therefore finite, so Kruskal's tree theorem can then be applied to show that $\lesssim$ is a well-quasi-order. □

Definition 3.4 (Generalization)**.** The generalization of two expressions e_1 and e_2 is a triple $(e_g, \theta_1, \theta_2)$ where θ_1 and θ_2 are substitutions such that $e_g\theta_1 \equiv e_1$ and $e_g\theta_2 \equiv e_2$.

The generalization we define for expressions e_1 and e_2 is the *most specific generalization*, denoted by $e_1 \sqcap e_2$, as defined in term algebra [4]. When an expression is generalized, sub-expressions within it are replaced with variables, which implies a loss of knowledge about the expression. The most specific generalization therefore entails the least possible loss of knowledge.

Definition 3.5 (Most Specific Generalization)**.** A most specific generalization of expressions e_1 and e_2 is a generalization $(e_g, \theta_1, \theta_2)$ such that for every other generalization $(e'_g, \theta'_1, \theta'_2)$ of e_1 and e_2, e_g is an instance of e'_g.

Definition 3.6 (The Generalization Operator $\sqcap$)**.** The most specific generalization of two expressions e_1 and e_2, denoted by $e_1 \sqcap e_2$, is defined as shown in Figure 3.

$$x \sqcap x = x$$

$$f \sqcap f = f$$

$$\begin{array}{ll} (c\ e_1 \ldots e_n) \sqcap (c\ e'_1 \ldots e'_n) = & (c\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i) \\ & \text{where} \\ & \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \end{array}$$

$$\begin{array}{ll} (\lambda x.e_0) \sqcap (\lambda x'.e'_0) = & (\lambda x.e^g_0, \theta_0, \theta'_0) \\ & \text{where} \\ & (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap (e'_0\{x' \mapsto x\}) \end{array}$$

$$\begin{array}{ll} (e_0\ e_1) \sqcap (e'_0\ e'_1) = & (e^g_0\ e^g_1, \theta_0 \cup \theta_1, \theta'_0 \cup \theta'_1) \\ & \text{where} \\ & (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \\ & (e^g_1, \theta_1, \theta'_1) = e_1 \sqcap e'_1 \end{array}$$

$$\begin{array}{l} (\textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n) \sqcap (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 \rightarrow e'_1 | \ldots | p'_n \rightarrow e'_n) = \\ \quad (\textbf{case}\ e^g_0\ \textbf{of}\ p_1 \rightarrow e^g_1 | \ldots | p_n \rightarrow e^g_n, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta'_i) \\ \quad \text{where} \\ \quad (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \\ \quad \forall i \in \{1 \ldots n\}.\exists \sigma.p_i \equiv (p'_i\ \sigma) \wedge (e^g_i, \theta_i, \theta'_i) = e_i \sqcap (e'_i\ \sigma) \end{array}$$

$$e \sqcap e' = (x, \{x \mapsto e\}, \{x \mapsto e'\}) \qquad \text{in all other cases } (x \text{ is fresh})$$

Figure 3: Generalization Rules

Within these rules, if both expressions have the same top-level construct, this is made the top-level construct of the resulting generalized expression, and the corresponding sub-expressions within the construct are then generalized. Otherwise, both expressions are replaced by the same fresh variable. It is assumed that the new variables introduced are all different and distinct from the original program variables. The following rewrite rule is exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions that were previously given different names:

$$(e, \theta \cup \{x \mapsto e', x' \mapsto e'\}, \theta' \cup \{x \mapsto e'', x' \mapsto e''\}) \Rightarrow$$

$$(e\{x \mapsto x'\}, \theta \cup \{x' \mapsto e'\}, \theta \cup \{x' \mapsto e''\})$$

The results of applying this most specific generalization to items 1-5 in Example 3 are as follows:

1. $(f\ g\ v, \{v \mapsto x\}, \{v \mapsto y\})$
2. $(f\ v, \{v \mapsto h\ x\}, \{v \mapsto g\ (h\ y)\})$
3. $(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto y\}, \{v_1 \mapsto z, v_2 \mapsto z\})$
4. $(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto x\}, \{v_1 \mapsto g\ y, v_2 \mapsto h\ y\})$
5. $(\lambda x.x, \{\}, \{\})$

During transformation, **let** expressions are introduced to represent the results of generalization; note that there were no **let** expressions in the original program and these are only introduced as a result of generalization. We define an abstraction operation on expressions that extracts the sub-terms resulting from generalization.

Definition 3.7 (Abstraction Operation)**.**

$$\begin{array}{l} \mathit{abstract}(e, e') = \mathbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathbf{in}\ e_0 \\ \text{where} \quad e \sqcap e' = (e_0, \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}, \theta) \end{array}$$

Positive supercompilation effectively performs a normal-order reduction of the input program. Previously encountered terms are memoised and if the current term is a renaming of a memoised one, then folding is performed, and the transformation is complete. If the current term has a memoised term embedded, then generalization is performed, and the sub-terms of the generalization are further transformed. Generalization ensures that a renaming of a memoised term is always eventually encountered, and that the transformation therefore terminates. The rules for our formulation of positive supercompilation are as shown in Figure 4.

The rules $\mathcal{T}$ are defined on an expression and its surrounding context, denoted by κ. Only those expressions that have a function in the redex position (immediately prior to unfolding) are memoised in rule (6). These expressions are replaced by a new function call, and this new function call is associated with the expression it replaced in the set ρ. On encountering a renaming of a memoised expression contained in ρ, it is also replaced by a corresponding call of its associated new function. The parameter Δ contains the set of function definitions in the original program.

The rules $\mathcal{T}'$ are defined on an expression and its surrounding context, also denoted by κ. These rules are applied when the normal-order reduction of the

$$
\begin{array}{ll}
(1) & \mathcal{T}[\![x]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}'[\![x]\!]\ \kappa\ \rho\ \Delta \\
(2) & \mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \Delta = c\ (\mathcal{T}[\![e_1]\!]\ \langle\rangle\ \rho\ \Delta) \ldots (\mathcal{T}[\![e_n]\!]\ \langle\rangle\ \rho\ \Delta) \\
(3) & \mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e'_1 \mid \cdots \mid p_k \rightarrow e'_k) : \kappa\rangle\ \rho\ \Delta = \\
& \qquad \mathcal{T}[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa\ \rho\ \Delta\ (p_i = c\ x_1 \ldots x_n) \\
(4) & \mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle\rangle\ \rho\ \Delta = \lambda x \rightarrow (\mathcal{T}[\![e]\!]\ \langle\rangle\ \rho\ \Delta) \\
(5) & \mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta = \mathcal{T}[\![e\{x \mapsto e'\}]\!]\ \kappa\ \rho\ \Delta \\
(6) & \mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta = \begin{cases} e\sigma & \text{if } \exists (e = e') \in \rho, \sigma . e'\sigma \equiv \kappa \bullet f \\ \mathcal{T}[\![abstract(\kappa \bullet f, e')]\!]\ \langle\rangle\ \rho\ \Delta & \text{if } \exists (e = e') \in \rho . e' \lesssim \kappa \bullet f \\ f'\ x_1 \ldots x_n & \text{otherwise} \\ \mathbf{where} & (f' \text{ is fresh}, \{x_1 \ldots x_n\} = fv(\kappa \bullet f)) \\ f' = \lambda x_1 \ldots x_n \rightarrow & \\ \quad (\mathcal{T}[\![\Delta(f)]\!]\ \kappa\ (\rho \cup \{f'\ x_1 \ldots x_n = \kappa \bullet f\})\ \Delta) & \end{cases} \\
(7) & \mathcal{T}[\![e\ e']\!]\ \kappa\ \rho\ \Delta = \mathcal{T}[\![e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta \\
(8) & \mathcal{T}[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n]\!]\ \kappa\ \rho\ \Delta = \\
& \qquad \mathcal{T}[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta \\
(9) & \mathcal{T}[\![\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta = \mathbf{let}\ x = (\mathcal{T}[\![e_0]\!]\ \langle\rangle\ \rho\ \Delta)\ \mathbf{in}\ (\mathcal{T}[\![e_1]\!]\ \kappa\ \rho\ \Delta)
\end{array}
$$

$$
\begin{array}{ll}
(10) & \mathcal{T}'[\![e]\!]\ \langle\rangle\ \rho\ \Delta = e \\
(11) & \mathcal{T}'[\![e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta = \mathcal{T}'[\![e\ (\mathcal{T}[\![e']\!]\ \langle\rangle\ \rho\ \Delta)]\!]\ \kappa\ \rho\ \Delta \\
(12) & \mathcal{T}'[\![x]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta = \\
& \mathbf{case}\ x\ \mathbf{of} \\
& p_1 \rightarrow (\mathcal{T}[\![(\kappa \bullet e_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \Delta) \mid \cdots \mid p_n \rightarrow\ (\mathcal{T}[\![(\kappa \bullet e_n)\{x \mapsto p_n\}]\!]\ \langle\rangle\ \rho\ \Delta) \\
(13) & \mathcal{T}'[\![e]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta = \\
& \qquad \mathbf{case}\ e\ \mathbf{of}\ p_1 \rightarrow (\mathcal{T}[\![e_1]\!]\ \kappa\ \rho\ \Delta) \mid \cdots \mid p_n \rightarrow (\mathcal{T}[\![e_n]\!]\ \kappa\ \rho\ \Delta)
\end{array}
$$

Figure 4: Positive Supercompilation Transformation Rules

input program becomes 'stuck' as a result of encountering a variable in the redex position. The expression will already have been transformed and so is not transformed any further, but the surrounding context is further transformed. In rule (12), if the context surrounding a variable redex is a **case**, then information is propagated to each branch of the **case** to indicate that this variable has the value of the corresponding branch pattern.

We can see that there are no trivial loops in the rules $\mathcal{T}$ and $\mathcal{T}'$. In the rules $\mathcal{T}$, a reduction step is performed in rules (3), (5) and (6), and sub-expressions of the redex are transformed in rules (2), (4), (7), (8) and (9). In rule (1), the rules $\mathcal{T}'$ are invoked, and in each of these rules, sub-expressions of the context are further transformed using the rules $\mathcal{T}$. Non-termination can therefore only occur if the terms encountered in the transformation rules grow uncontrollably.

4 Termination

In [19], three different possible causes of non-termination of positive supercompilation when applied to a first-order functional language were identified: *obstructing function calls*, *accumulating parameters* and *accumulating narrowing*[1]. A further possible cause of non-termination has also been identified in [18] for the deforestation of higher-order functional languages, which also applies to positive supercompilation: *accumulating spines*. We now give examples of each of these causes of non-termination.

$$
\begin{array}{l}
nrev\ xs \\
\mathbf{where} \\
\begin{array}{lcl}
nrev & = & \lambda xs \rightarrow \mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad\quad\quad\quad Nil \rightarrow Nil \\
 & & \quad\quad\quad\quad \mid Cons\ x'\ xs' \rightarrow app\ (nrev\ xs')\ (Cons\ x'\ Nil) \\
app & = & \lambda xs \rightarrow \lambda ys \rightarrow \mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad\quad\quad\quad\quad\quad Nil \rightarrow ys \\
 & & \quad\quad\quad\quad\quad\quad \mid Cons\ x'\ xs' \rightarrow Cons\ x'\ (app\ xs'\ ys)
\end{array}
\end{array}
$$

Figure 5: Example Obstructing Function Call

Example 4 (Obstructing Function Call)**.** Consider the program shown in Figure 5.

During the transformation of this program, we encounter the progressively larger terms: $nrev\ xs$, $\mathbf{case}\ (nrev\ xs)\ \mathbf{of} \cdots$, $\mathbf{case}\ (\mathbf{case}\ (nrev\ xs)\ \mathbf{of} \cdots)\ \mathbf{of} \cdots$, etc. The call to *nrev* thus prevents the surrounding context from being reduced, so this context continues to grow. This call to *nrev* is therefore an obstructing function call.

Example 5 (Accumulating Parameter)**.** Consider the program shown in Figure 6.

During transformation of this program, we encounter the progressively larger terms: $arev'\ xs\ Nil$, $arev'\ xs'\ (Cons\ x'\ Nil)$, $arev'\ xs''\ (Cons\ x''\ (Cons\ x'\ Nil))$, etc. The second parameter in each recursive call to $arev'$ therefore accumulates a progressively larger term.

$$
\begin{array}{l}
arev\ xs \\
\textbf{where} \\
arev \quad = \quad \lambda xs \rightarrow arev'\ xs\ Nil \\
arev' \quad = \quad \lambda xs \rightarrow \lambda ys \rightarrow \ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Nil \rightarrow ys \\
\qquad\qquad\qquad\qquad\qquad\quad \mid Cons\ x'\ xs' \rightarrow arev'\ xs'\ (Cons\ x'\ ys)
\end{array}
$$

Figure 6: Example Accumulating Parameter

$$
\begin{array}{l}
app\ xs\ xs \\
\textbf{where} \\
app \quad = \quad \lambda xs \rightarrow \lambda ys \rightarrow \ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Nil \rightarrow ys \\
\qquad\qquad\qquad\qquad\qquad\quad \mid Cons\ x\ xs \rightarrow Cons\ x\ (app\ xs\ ys)
\end{array}
$$

Figure 7: Example Accumulating Narrowing

Example 6 (Accumulating Narrowing)**.** Consider the program shown in Figure 7.
During transformation of this program, we encounter the progressively larger terms: $app\ xs\ xs$, $app\ xs'\ (Cons\ x'\ xs')$, $app\ xs''\ (Cons\ x'\ (Cons\ x''\ xs''))$, etc. The second parameter in each recursive call to *app* therefore also accumulates a progressively larger term, but in this case the accumulation is caused by unification-based information propagation (narrowing).

Example 7 (Accumulating Spine)**.** Consider the program shown in Figure 8.

$$
\begin{array}{l}
f\ x \\
\textbf{where} \\
f \quad = \quad \lambda x \rightarrow f\ x\ x
\end{array}
$$

Figure 8: Example Accumulating Spine

During transformation of this program, we encounter the progressively larger terms: $f\ x$, $f\ x\ x$, $f\ x\ x\ x$, etc. Each recursive call to f therefore accumulates an additional parameter. We should also note that this type of function definition is prohibited in most typing schemes.

[1] This was originally called *accumulating side-effects*, but since functional languages do not admit side-effects, we prefer to call this possible cause of non-termination *accumulating narrowing*.

In all of the above examples, a previously encountered term becomes embedded within a subsequent one. Since the sequence of transformation steps in each example must always include function unfolding, this embedding will be detected by our positive supercompilation algorithm and generalization will be performed, thus ensuring termination. However, not all recursion has to take place through named functions; this will also occur if there is a λ-term which is not strongly normalizing, as is the case for the programs given in examples 1 and 2. To ensure termination, we therefore need to make sure that all recursive functions are named.

5 Ensuring Termination

In this section, we show how to ensure the termination of our formulation of positive supercompilation. As shown in the previous section, non-termination can occur even in the absence of named functions if we have a λ-term which is not strongly normalizing. The sequence of terms obtained will require no function unfolding, and therefore will not be checked for possible folding or generalization. To avoid this possibility, we require that programs are in λ-*prefix* form, in which the only λ-abstractions occur in the prefix of the program expression or the prefix of function bodies. This λ-prefix form is defined as shown in Figure 9.

$$
\begin{array}{lll}
prog & ::= pf_0 \ \textbf{where} \ f_1 = pf_1 \ldots f_n = pf_n & \text{Program} \\
\\
pf & ::= \lambda x \ \rightarrow pf & \lambda\text{-Abstraction} \\
 & \ \ \ \mid pf' & \lambda\text{-Free Expression} \\
\\
pf' & ::= x & \text{Variable} \\
 & \ \ \ \mid c \ pf'_1 \ldots pf'_n & \text{Constructor} \\
 & \ \ \ \mid f & \text{Function Call} \\
 & \ \ \ \mid pf'_0 \ pf'_1 & \text{Application} \\
 & \ \ \ \mid \textbf{let} \ x = pf'_0 \ \textbf{in} \ pf'_1 & \text{Let Expression} \\
 & \ \ \ \mid \textbf{case} \ pf'_0 \ \textbf{of} \ p_1 \rightarrow pf'_1 \mid \cdots \mid p_n \rightarrow pf'_n & \text{Case Expression} \\
\\
p & ::= c \ x_1 \ldots x_n & \text{Pattern}
\end{array}
$$

Figure 9: λ-Prefix Form

It is quite straightforward to convert any program into this form; simply replace any λ-abstractions which are not in the prefix of the program expression or the prefix of a function body with a freshly named function. λ-lifting [7] is

also performed to abstract over any of the free variables in the λ-abstraction, as named functions in our language cannot contain free variables. If the λ-abstraction matches one which has already been replaced by a function call, then it is replaced with a call to the same function as previously, thus minimizing the number of new function definitions which are introduced.

Example 8. Consider the program given in Example 2. This contains four λ-abstractions which are not in its prefix. The abstraction over f is made the body of the freshly named function f_1 and the abstraction over y is made the body of the freshly named function f_2. The other two abstractions over x are identical, so this abstraction is made the body of the freshly named function f_3; however, since the variable f appears free in this expression, λ-lifting is performed to abstract over f, and this extra parameter is added to the two calls of f_3.

We now prove that our simple pre-processing step is sufficient to ensure the termination of our formulation of the positive supercompilation algorithm. We do this by showing that if the original input to our positive supercompilation algorithm is in λ-prefix form, then all of the terms subsequently encountered must be in a particular form. We then show that any potentially infinite sequence of transformation steps in which the expressions are in this form must include function unfolding, so the transformation is guaranteed to terminate.

Lemma 5.1 (On The Form of Terms Encountered by Positive Supercompilation)**.** If the input to our positive supercompilation algorithm is in λ-prefix form, then all of the terms subsequently encountered must have the following form in which the only λ-abstractions are in the prefix of the redex:

$$\begin{aligned} sf ::= &\ \textbf{case}\ sf\ \textbf{of}\ p_1 \rightarrow pf'_1 \mid \cdots \mid p_n \rightarrow pf'_n \\ &\mid sf\ pf' \\ &\mid pf \end{aligned}$$

where pf and pf' are as defined in figure 9.

Proof. The interesting cases are where substitution is performed in rules (3) and (5), and where generalisation is performed in rule (6). Since the only λ-abstractions can be in the redex, the terms which are substituted in rules (3) and (5) cannot contain any λ-abstractions, so the resulting term must also be in the above form. Generalization is performed in rule (6) when the redex is a function name, so the generalized term cannot contain any λ-abstractions and neither can the extracted sub-expressions. Details of the proof are given in Appendix A. □

Lemma 5.2. If the input to our positive supercompilation algorithm is in λ-prefix form, then every infinite sequence of transformation steps must include function unfolding.

Proof. Every infinite sequence of transformation steps must include either function unfolding or λ-application. If the input term is in λ-prefix form, then by Lemma 5.1, the only λ-abstractions in the terms subsequently encountered will be in the prefix of the redex, so transformation rule (4) or (5) will be continually applied until there are no λ-abstractions remaining in the current term. Thus, the only way in which new λ-abstractions can be introduced is by function unfolding. Thus every infinite sequence of transformation steps must include function unfolding. □

We are now able to prove our desired result.

Theorem 5.3. If the input to our positive supercompilation algorithm is in λ-prefix form, then it is guaranteed to terminate.

Proof. The proof is by contradiction. If our positive supercompilation algorithm did not terminate then the set of memoised expressions ρ must be infinite, since by Lemma 5.2 every infinite sequence of transformation steps must include function unfolding. Every new expression which is added to ρ cannot have any of the previous expressions in ρ embedded within it by the homeomorphic embedding relation $\lesssim$, since folding or generalization would have been performed instead. However, this contradicts the fact that $\lesssim$ is a well-quasi-order (Theorem 3.3). □

6 Conclusion and Related Work

In this paper, we have described a simple pre-processing step which can be applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there must be an unfolding. This involves introducing names for some anonymous functions to ensure that only memoising expressions immediately preceding an unfold step is sufficient to ensure termination of the transformation. The original positive supercompilation algorithm [19, 22] was only formulated for a first-order language, so it was sufficient to only memoise the expressions immediately prior to an unfolding step. In the higher-order formulations of positive supercompilation given by Mitchell [17] and Bolingbroke [2], *all* expressions are memoised. We argue that the extra work required for the additional checking for generalization and folding is too computationally expensive, particularly since the homeomorphic embedding check is very time consuming; this has been borne out by the experimental results obtained using this approach. It should also be

pointed out that the implementation of positive supercompilation in [17] will not terminate on programs such as that given in Example 2. This is because the simplification rules that are applied to terms prior to transformation by positive supercompilation will not terminate for such programs which use *contravariant* (*negative*) data types. It is argued in [17] that this problem only occurs for contrived programs, and it is also a problem for GHC, which will not terminate when compiling this example program. However, this seems unsatisfactory. It is noted in [17] that this non-termination problem could be avoided by not performing simplification on negative data types. A similar approach was also adopted by Jonsson [8] and Mendel-Gleason [15] by requiring that all types in the input program are *positive*. This also seems unsatisfactory since such typing schemes are not used in mainstream functional languages.

Rather than memoising all expressions, the approach taken in the higher-order supercompiler HOSC [9, 10, 11] is to restrict this to only those expressions which are considered to be *non-trivial*. In HOSC 1.0 [9], an expression is considered to be non-trivial if it either has a function in the redex or an irreducible expression in the selector of a **case** expression (corresponding to the left hand side of our rules (6), (12) and (13)). However, it was subsequently discovered [10] that this was not sufficient to ensure the termination of the supercompiler, because it will not terminate for programs which encode recursion using a data type such as that given in Example 2. In HOSC 1.1 [10], an expression for which the next transformation step involves a substitution (corresponding to the left hand side of our rules (3) and (5)) is considered to be non-trivial if it satisfies a size constraint in which the expression resulting from the substitution is no smaller than the expression before substitution. However, it was subsequently discovered [11] that memoising every expression for which the next transformation step involves a β-reduction produces poor residual programs. In HOSC 1.5 [11], expressions for which the next transformation step involves a β-reduction are not memoised, but all applications and **case** expressions are (corresponding to the left hand side of our rules (7) and (8)), thus ensuring that in any potentially infinite sequence of transformation steps expressions will still be memoised. However, we argue that this approach still requires a lot of additional work when checking for generalization and folding, and can still produce poor residual programs. Using our approach, after the pre-processing step, only those expressions encountered immediately prior to an unfolding step (rule (6)) need to be memoised and checked for generalization and folding.

Using our approach, we are not able to prove that the programs resulting from transformation are an improvement over the original programs, since not all new functions are introduced in conjunction with the unfolding of an old function. However, this is a problem for all of the described algorithms for higher-order positive supercompilation. However, using our approach, less new

functions will be created and generalization will be performed less often, resulting in more improved residual programs. We have implemented the techniques described in this paper and preliminary experiments show that they make the resulting supercompiler more efficient and that they produce more improved residual programs in some cases.

Acknowledgements

Many thanks to the anonymous referees who provided very useful comments and feedback on an earlier version of this paper. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

References

[1] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1–2):25–46, 1993.

[2] Max Bolingbroke and Simon Peyton Jones. Supercompilation by Evaluation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, pages 135–146 , 2010.

[3] L. Damas and R. Milner. Principal Type Schemes for Functional Programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.

[5] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

[6] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[7] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the Workshop on Implementation of Functional Languages*, pages 165–180, February 1985.

[8] Peter Jonsson. *Time- and Size-Efficient Supercompilation.* PhD thesis, Dept. of Computer Science and Electrical Engineering, Lulea University of Technology, 2011.

[9] Ilya Klyuchnikov. Supercompiler HOSC 1.0: Under the Hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.

[10] Ilya Klyuchnikov. Supercompiler HOSC 1.1: Proof of Termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[11] Ilya Klyuchnikov. Supercompiler HOSC 1.5: Homeomorphic Embedding and Generalization in a Higher-Order Setting. Preprint 62, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[12] J.B. Kruskal. Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[13] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of the International Static Analysis Symposium, Pisa, Italy*, pages 230–245, 1998.

[14] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle.* PhD thesis, Université de Nice - Sophia Antipolis, 1994.

[15] Gavin Mendel-Gleason. *Types and Verification for Infinite State Systems.* PhD thesis, School of Computing, Dublin City University, 2012.

[16] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[17] Neil Mitchell. Rethinking Supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 309–320 , 2010.

[18] H. Seidl and M. H. Sørensen. Constraints to Stop Higher-Order Deforestation. *Proceedings of the Twelfth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 400–413, 1997.

[19] M. H. Sørensen. Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[20] M. H. Sørensen. Convergence of Program Transformers in the Metric Space of Trees. *Lecture Notes in Computer Science*, 1422:315–337, 1998.

[21] M. H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *Lecture Notes in Computer Science*, 787:335–351, 1994.

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

[23] V.F. Turchin. Program Transformation by Supercompilation. *Lecture Notes in Computer Science*, 217:257–281, 1985.

[24] V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.

A Proof of Lemma 5.1

We need to prove that for each of the transformation rules given in Figure 4, if $\mathcal{T}[\![e]\!]\ \kappa\ \rho\ \Delta = \ldots \mathcal{T}[\![e_1]\!]\ \kappa_1\ \rho_1\ \Delta \ldots \mathcal{T}[\![e_n]\!]\ \kappa_n\ \rho_n\ \Delta \ldots$ and $\kappa \bullet e \in sf$, then $\forall i \in \{1 \ldots n\}. \kappa_i \bullet e_i \in sf$.

Case (1): $\mathcal{T}[\![x]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}'[\![x]\!]\ \kappa\ \rho\ \Delta$

All further applications of $\mathcal{T}$ arising from this application of $\mathcal{T}'$ are applied to sub-expressions from the context κ. Since $\kappa \bullet x \in sf$, then for all sub-expressions e_i in κ, $e_i \in pf'$, so $e_i \in sf$

Case (2): $\mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \Delta = c\ (\mathcal{T}[\![e_1]\!]\ \langle\rangle\ \rho\ \Delta) \ldots (\mathcal{T}[\![e_n]\!]\ \langle\rangle\ \rho\ \Delta)$

Since $c\ e_1 \ldots e_n \in sf$, then $\forall i \in \{1 \ldots n\}.e_i \in pf'$, so $\forall i \in \{1 \ldots n\}.e_i \in sf$
Case (3): $\mathcal{T}[\![c\ e_1 \ldots e_n]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e'_1 \mid \cdots \mid p_k \rightarrow e'_k) : \kappa\rangle\ \rho\ \Delta =$
$\mathcal{T}[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa\ \rho\ \Delta\ (p_i = c\ x_1 \ldots x_n)$

Since $\langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e'_1 \mid \cdots \mid p_k \rightarrow e'_k) : \kappa\rangle \bullet (c\ e_1 \ldots e_n) \in sf$,
then $\forall i \in \{1 \ldots n\}.e_i \in pf' \wedge \forall i \in \{1 \ldots k\}.\kappa \bullet e_i{}' \in pf'$,
so $\kappa \bullet e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\} \in sf$

Case (4): $\mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle\rangle\ \rho\ \Delta = \lambda x \rightarrow (\mathcal{T}[\![e]\!]\ \langle\rangle\ \rho\ \Delta)$

Since $\lambda x\ \rightarrow e \in sf$, then $e \in pf$, so $e \in sf$

Case (5): $\mathcal{T}[\![\lambda x\ \rightarrow e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta = \mathcal{T}[\![e\{x \mapsto e'\}]\!]\ \kappa\ \rho\ \Delta$

Since $\langle(\bullet\ e') : \kappa\rangle \bullet (\lambda x\ \rightarrow e) \in sf$, then $\kappa \bullet e \in pf \wedge e' \in pf'$,
so $\kappa \bullet e\{x \mapsto e'\} \in sf$

Case (6a): $\mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta = e\sigma$ if $\exists (e = e') \in \rho, \sigma.e'\sigma \equiv \kappa \bullet f$

No further transformation is performed.

Case (6b):
$\mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}[\![abstract(\kappa \bullet f, e')]\!]\ \langle\rangle\ \rho\ \Delta$ if $\exists (e = e') \in \rho.e' \lesssim \kappa \bullet f$

Since $\kappa \bullet f \in sf$, then $\kappa \bullet f \in pf'$, so $abstract(\kappa \bullet f, e') \in sf$

Case (6c):
$\mathcal{T}[\![f]\!]\ \kappa\ \rho\ \Delta =\ \ f'\ x_1 \ldots x_n$ otherwise
where
$f' = \lambda x_1 \ldots x_n \rightarrow (\mathcal{T}[\![\Delta(f)]\!]\ \kappa\ (\rho \cup \{f'\ x_1 \ldots x_n = \kappa \bullet f\})\ \Delta)$
$(f'$ is fresh, $\{x_1 \ldots x_n\} = fv(\kappa \bullet f))$

Since $\kappa \bullet f \in sf \wedge \Delta(f) \in pf$, then $\kappa \bullet \Delta(f) \in sf$

Case (7): $\mathcal{T}[\![e\ e']\!]\ \kappa\ \rho\ \Delta = \mathcal{T}[\![e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \rho\ \Delta$

Since $\kappa\bullet(e\ e') \in sf$, then $\langle(\bullet\ e') : \kappa\rangle\bullet e \in sf$

Case (8): $\mathcal{T}[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 \mid\cdots\mid p_n \rightarrow e_n]\!]\ \kappa\ \rho\ \Delta =$
$\mathcal{T}[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e_1 \mid\cdots\mid p_n \rightarrow e_n) : \kappa\rangle\ \rho\ \Delta$

Since $\kappa\bullet(\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \rightarrow e_1 \mid\cdots\mid p_n \rightarrow e_n) \in sf$,
then $\langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \rightarrow e_1 \mid\cdots\mid p_n \rightarrow e_n) : \kappa\rangle\bullet e_0 \in sf$

Case (9): $\mathcal{T}[\![\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta = \mathbf{let}\ x = (\mathcal{T}[\![e_0]\!]\ \langle\rangle\ \rho\ \Delta)\ \mathbf{in}\ (\mathcal{T}[\![e_1]\!]\ \kappa\ \rho\ \Delta)$

Since $\kappa\bullet(\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1) \in sf$,
then $e_0 \in pf' \wedge \kappa\bullet e_1 \in pf'$, so $e_0 \in sf \wedge \kappa\bullet e_1 \in sf$

Acceleration For Presburger Petri Nets

Jérôme Leroux*

LaBRI, Université de Bordeaux, CNRS

Abstract

The reachability problem for Petri nets is a central problem of net theory. The problem is known to be decidable by inductive invariants definable in the Presburger arithmetic. When the reachability set is definable in the Presburger arithmetic, the existence of such an inductive invariant is immediate. However, in this case, the computation of a Presburger formula denoting the reachability set is an open problem. Recently this problem got closed by proving that if the reachability set of a Petri net is definable in the Presburger arithmetic, then the Petri net is flatable, i.e. its reachability set can be obtained by runs labeled by words in a bounded language. As a direct consequence, classical algorithms based on acceleration techniques effectively compute a formula in the Presburger arithmetic denoting the reachability set.

1 Introduction

Petri Nets are one of the most popular formal methods for the representation and the analysis of parallel processes [1]. The reachability problem is central since many computational problems (even outside the realm of parallel processes) reduce to this problem. Sacerdote and Tenney provided in [14] a partial proof of decidability of this problem. The proof was completed in 1981 by Mayr [13] and simplified by Kosaraju [8] from [13,14]. Ten years later [9], Lambert provided a further simplified version based on [8]. This last proof still remains difficult and the upper-bound complexity of the corresponding algorithm is just known to be non-primitive recursive. Nowadays, the exact complexity of the reachability problem for Petri nets is still an open-question. Even an Ackermannian upper bound is open (this bound holds for Petri nets with finite reachability sets [2]).

Basically, a Petri net is a pair $(T, \mathbf{c}_{\text{init}})$ where $T \subseteq \mathbb{N}^d \times \mathbb{N}^d$ is a finite set of *transitions*, and $\mathbf{c}_{\text{init}} \in \mathbb{N}^d$ is the *initial configuration*. A vector $\mathbf{c} \in \mathbb{N}^d$ is called a *configuration*. The semantics of Petri nets is defined as follows. A transition $t = (\mathbf{a}, \mathbf{a}')$ is said to be *fireable* from a configuration $\mathbf{x}$ if $\mathbf{x} \geq \mathbf{a}$. We introduce the binary relation $\xrightarrow{t}$ over the configurations in $\mathbb{N}^d$ defined by $\mathbf{x} \xrightarrow{t} \mathbf{x}'$ if t is fireable from $\mathbf{x}$ and $\mathbf{x}' = \mathbf{x} - \mathbf{a} + \mathbf{a}'$. A *run* from a configuration

*Work funded by ANR grant REACHARD-ANR-11-BS02-001.

$\mathbf{x}$ to a configuration $\mathbf{x}'$ labelled by a word $\sigma = t_1 \dots t_k$ of transitions $t_j \in T$ is a sequence $(\mathbf{c}_0, t_1, \mathbf{c}_1, \dots, t_k, \mathbf{c}_k)$ where $\mathbf{c}_0, \dots, \mathbf{c}_k$ are configurations such that $\mathbf{c}_0 = \mathbf{x}$, $\mathbf{c}_k = \mathbf{x}'$, and such that $\mathbf{c}_{j-1} \xrightarrow{t_j} \mathbf{c}_j$ for every $1 \leq j \leq k$. When $\mathbf{x}$ is the initial configuration, the configuration $\mathbf{x}'$ is said to be *reachable*. The *reachability set* is the set of reachable configurations.

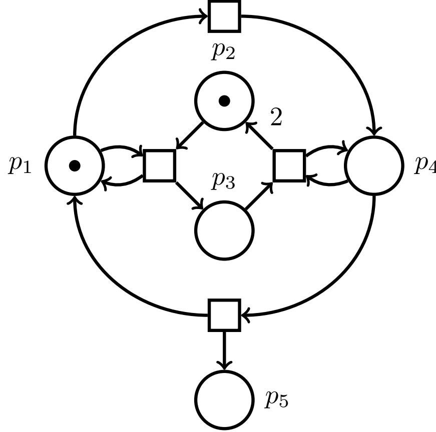


Figure 1: The Hopcroft and Pansiot net.

Example 1.1. *The Petri net depicted in Figure 1 was introduced in [7] as an example of Petri net having a reachability set which cannot be defined by a formula in the logic* $\mathrm{FO}(\mathbb{N}, +)$*, called the* Presburger arithmetic*. In fact, the set of reachable configurations is equal to:*

$$\left\{ (p_1, p_2, p_3, p_4, p_5) \in \mathbb{N}^5 \mid \begin{array}{l} (\ p_1 = 1 \wedge p_4 = 0 \wedge 1 \leq p_2 + p_3 \leq 2^{p_5}\ ) \vee \\ (\ p_1 = 0 \wedge p_4 = 1 \wedge 1 \leq p_2 + 2p_3 \leq 2^{p_5+1}\ ) \end{array} \right\}$$

Recently, in [10], the reachability sets of Petri nets are proved to be *almost semilinear*, a class of sets that extends the class of Presburger sets (the sets definable in $\mathrm{FO}(\mathbb{N}, +)$) inspired by the *semilinear sets* [5]. Note that in general reachability sets are not definable in the Presburger arithmetic [7] (see Example 1.1). An application of the almost semilinear sets was provided; a final configuration is not reachable from an initial one if and only if there exists a forward inductive invariant definable in the Presburger arithmetic that contains the initial configuration but not the final one. Since we can decide if a Presburger formula denotes a forward inductive invariant, we deduce that there exist checkable certificates of non-reachability in the Presburger arithmetic. In particular,

there exists a simple algorithm for deciding the general Petri net reachability problem based on two semi-algorithms. A first one that tries to prove the reachability by enumerating finite sequences of actions and a second one that tries to prove the non-reachability by enumerating Presburger formulas. Such an algorithm always terminates in theory but in practice an enumeration does not provide an efficient way for deciding the reachability problem. In particular the problem of deciding *efficiently* the reachability problem is still an *open question*.

When the reachability set is definable in the Presburger arithmetic, the existence of checkable certificates of non-reachability in the Presburger arithmetic is immediate since the reachability set is a forward inductive invariant (in fact the most precise one). The problem of deciding if the reachability set of a Petri is definable in the Presburger arithmetic was studied twenty years ago independently by Dirk Hauschildt during his PhD [6] and Jean-Luc Lambert. Unfortunately, these two works were never published. Moreover, from these works, it is difficult to deduce a simple algorithm for computing a Presburger formula denoting the reachability set when such a formula exists.

For the class of *flatable* Petri nets [3, 12], such a computation can be performed with *accelerations techniques*. Let us recall that a Petri net is said to be *flatable* if there exist some words $\sigma_1, \dots, \sigma_k \in T^*$ such that every reachable configuration is the target of a run labeled by a word in $\sigma_1^* \dots \sigma_k^*$ from the initial configuration (A language included in $\sigma_1^* \dots \sigma_k^*$ is said to be *bounded* [4]). *Acceleration techniques* provide a framework for deciding reachability properties that works well in practice but without termination guaranty in theory. Intuitively, acceleration techniques consist in computing with some symbolic representations transitive closures of sequences of actions. For Petri nets, the Presburger arithmetic is known to be expressive enough for this computation. As a direct consequence, when the reachability set of a Petri net is computable with acceleration techniques, this set is necessarily definable in the Presburger arithmetic. In [12], we proved that a Petri net is flatable if, and only if, its reachability set is computable by acceleration.

Recently, we proved that many classes of Petri nets with known Presburger reachability sets are flatable [12] and we conjectured that Petri nets with reachability sets definable in the Presburger arithmetic are flatable. In [11] the conjecture get closed positively. As a direct consequence, classical acceleration techniques always terminate on the computation of Presburger formulas denoting reachability sets of Petri nets when such a formula exists.

Acknowledgment

This work was supported by the ANR project ReacHard (ANR-11-BS02-001), and the "Réseau de formation et de recherche franco-russe en Sciences et Technologies de l'Information et de la Communication".

References

[1] Javier Esparza and Marcus Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[2] Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. Ackermannian and primitive-recursive bounds with dickson's lemma. In *Proc. of LICS 2011*, pages 269–278. IEEE Computer Society, 2011.

[3] Laurent Fribourg. Petri nets, flat languages and linear arithmetic. In María Alpuente, editor, *Proc. of WFLP'2000*, pages 344–365, 2000.

[4] S. Ginsburg and E. H. Spanier. Bounded regular sets. *Proceedings of the American Mathematical Society*, 17(5):1043–1049, 1966.

[5] Seymour Ginsburg and Edwin H. Spanier. Semigroups, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

[6] Dirk Hauschildt. *Semilinearity of the Reachability Set is Decidable for Petri Nets.* PhD thesis, University of Hamburg, 1990.

[7] John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoritical Computer Science*, 8:135–159, 1979.

[8] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proc. of STOC'82*, pages 267–281. ACM, 1982.

[9] Jean Luc Lambert. A structure to decide reachability in petri nets. *Theoretical Computer Science*, 99(1):79–104, 1992.

[10] Jérôme Leroux. The general vector addition system reachability problem by Presburger inductive invariants. In *Proc. of LICS 2009*, pages 4–13. IEEE Computer Society, 2009.

[11] Jérôme Leroux. Presburger vector addition systems. In *Proc. LICS 2013*. IEEE Computer Society, 2013. To appear.

[12] Jérôme Leroux and Grégoire Sutre. Flat counter automata almost everywhere! In *Proc. of ATVA'05*, volume 3707 of *LNCS*, pages 489–503. Springer, 2005.

[13] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *Proc. of STOC'81*, pages 238–246. ACM, 1981.

[14] George S. Sacerdote and Richard L. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *Proc. of STOC'77*, pages 61–76. ACM, 1977.

Transforming EVENT B Models into Verified C# Implementations

Dominique Méry[1] and Rosemary Monahan[2]

[1] Université de Lorraine
LORIA, BP 239, 54506 Vandœuvre lès Nancy, France
mery@loria.fr
[2] Department of Computer Science,
National University of Ireland, Maynooth, Co. Kildare, Ireland
rosemary.monahan@nuim.ie

Abstract

The refinement-based approach to developing software is based on the *correct-by-construction* paradigm where software systems are constructed via the step-by-step refinement of an initial high-level specification into a final concrete specification. Proof obligations, generated during this process are discharged to ensure the consistency between refinement levels and hence the system's overall correctness.

Here, we are concerned with the refinement of specifications using the EVENT B modelling language and its associated toolset, the RODIN platform. In particular, we focus on the final steps of the process where the final concrete specification is transformed into an executable algorithm. The transformations involved are (a) the transformation from an EVENT B specification into a concrete recursive algorithm and (b) the transformation from the recursive algorithm into its equivalent iterative version. We prove both transformations correct and verify the correctness of the final code in a static program verification environment for C# programs, namely the Spec# programming system.

1 Introduction

EVENT B is a formal modelling language developed by Abrial [1]. Key features of EVENT B are the use of set theory as a modelling notation, the use of refinement to represent software systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels. This mathematical proof is typically achieved in a semi-automated way, with the user interacting with theorem proving tools using the RODIN platform. The final concrete representation of the system results from discharging accumulated proof obligations, which are recorded as invariants of the system under development.

In this paper, we focus on the transformation of the final concrete specification into an executable algorithm. We present the transformations for (a) transforming an

EVENT B specification into a recursive algorithm and (b) transforming from that recursive program to an iterative version of the same program. We prove both transformations correct and verify the correctness of the final code in a static program verification environment for C# programs, namely the Spec# programming system. This work is a component of our general framework for integrating two popular approaches to formal software development. In this framework we combine the efforts of program refinement as supported by EVENT B and program verification as supported by the Spec# programming system. The architecture induces a methodology [12], which improves the usability of formal verification tools for the specification, the construction and the verification of correct sequential algorithms.

In the sections that follow, we provide an overview of EVENT B, our integrated development framework and the transformations that form an essential component of the transformation of concrete specifications into an executable recursive program. Finally, we present the iterative program verified in the automatic program verification environment of the Spec# programming system.This verification ensures that the generated program is correct with respect to the initial EVENT B abstract model.

2 The EVENT B Modelling Framework

EVENT B [1] is a formal method for system-level modelling and analysis. An EVENT B model is defined via *contexts* and *machines*. As shown in Figure 1, *machines* express dynamic information about the model via *events*, which modify state variables that are defined in the *contexts*. Machines may also express other properties, such as invariant and safety properties of the model.

An event is equivalent to a *reactive action* waiting for a condition (called a guard) to hold in order to trigger an action. It has three main parts: a list of local parameters, a guard G and a relation R over values of state variables denoted *pre*-values (x) and *post*-values (x'). When the guard holds the actions in the event body modify the state variables according to the relation R. The *before–after* predicate $BA(e)(x,x')$ associated with each event describes the event as a logical predicate for expressing the relationship linking values of the state variables just before, and just after, the *execution* of event e. We indicate the ith action in each event using the prefix $acti$. The most common event representation has the form

$$\text{ANY } t \text{ WHERE } G(t,x) \text{ THEN } x :|(R(x,x',t)) \text{ END}$$

where t is a local parameter and the event actions establish $x :|(R(x,x',t))$. The form is semantically equivalent to $\exists t \cdot (G(t,x) \ \wedge \ R(x,x',t))$.

These basic structures are extended by the refinement process, which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows the gradual development of EVENT B

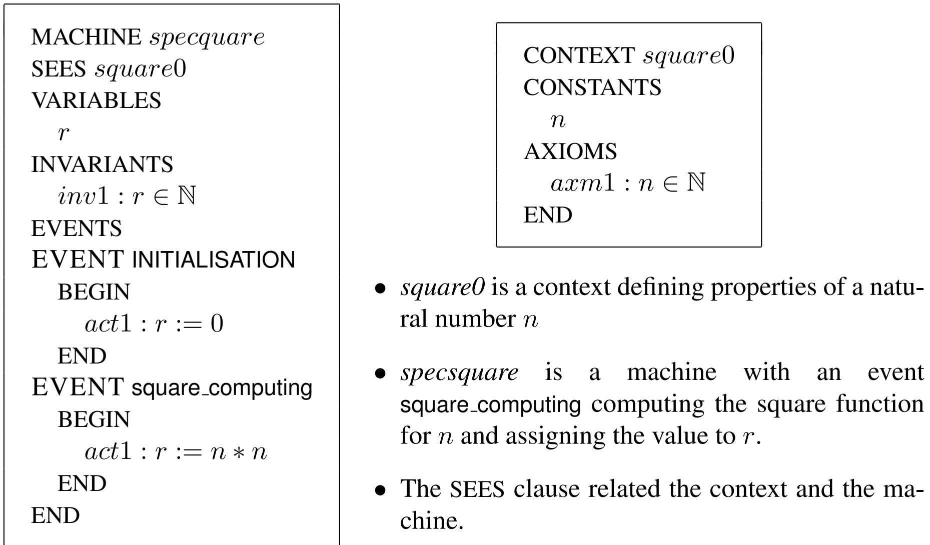


Figure 1: EVENT B structure: context and machine

models and the validation of each decision step. The refinement of a formal model allows us to enrich our formal reactive models via a *step-by-step* approach and is the foundation of our correct-by-construction approach [7]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description.

Refinement is achieved by extending the list of state variables (and possibly suppressing some of them), by refining each abstract event to a set of possible concrete versions, and by adding new events. The abstract (x) and concrete (y) state variables are linked by means of a *gluing invariant* $J(x, y)$, which must be maintained throughout the system modelling. A number of proof obligations ensure that each abstract event is correctly refined by its corresponding concrete version, each new event refines $skip$, no new event takes control forever and relative deadlock freedom is preserved. The refinement relationship is expressed as follows: a model M is refined by a model P, when P simulates M. The final concrete model is close to the behaviour of the final software system that executes events using real source code. In this paper we present the translation of these concrete models to recursive and iterative algorithms that can be directly mapped to code.

The EVENT B modelling language is supported by the Atelier B [3] environment and by the RODIN platform [14] . Atelier B and the RODIN platform both provide facilities for editing contents and machines, refinements, contexts and projects, for generating proof obligations corresponding to a given property, for proving proof obligations

in an automatic or/and interactive process and for animating models.

3 Implementing EVENT B models

Our integrated development framework for implementing abstract EVENT B models brings together the strengths of the refinement based approaches and verification based approaches to software development. In particular, our framework supports:

1. Splitting the abstract specification to be solved into its component specifications.
2. Refining these specifications into a concrete model using EVENT B and the RODIN platform.
3. Transforming the concrete model into recursive and iterative algorithms that can be directly implemented as real source code.
4. Verifying the iterative algorithm in the automatic program verification environment of Spec#.

In this paper we focus on the transformations involved in item number three above. First we provide an overview of our integrated development framework to help set the context of our work.

3.1 An overview of our integrated development framework

Figure 2 provides an overview of our framework for refinement based program verification. The problem to be solved is stated as a collection of method contracts, in the form of a Spec# program. Spec# is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# through a rich assertion language that allows the specification of objects through class invariants, field annotations, and method specifications [8, 2]. Method preconditions, annotated with the keyword *requires*, express the constraints under which the method will execute correctly. Method postconditions, annotated with the keyword *ensures*, express what should happen as a result of the methods proper execution. The post-condition of methods may refer to the return value of a method using the keyword *result*. The type of the value stored in result must be a subtype of the method's return type. Note also that variables in post-conditions can be prefixed with the keyword *old* e.g., $x = old(x) + 1$ indicates that the new value of x is the old value incremented by 1.

Spec# comes with a sound programming methodology that permits the extended static verification of specifications and their implementations. This process is represented by the arrow labelled *checking* in Figure 2. Dynamic analysis allows the compiler to emit run-time checks at compile time, recording the assertions in the specification as meta-data for consumption by downstream tools. This allows the analysis

of program correctness before allowing the program to be run. Internally, it uses an automatic SMT solver (such as Simplify [6] or Z3 [5]) that analyses the verification conditions to prove the correctness of the program or find errors in it.

Note that in the traditional verification approach, the programmer provides both the specification and its implementation. In our integrated development framework we use model refinement in Event B to construct the Spec# implementation from its specification. This refinement also generates the proof obligations that must be discharged as part of the verification. We add these as invariants and assertions in the program so that its verification is completely automatic with the Spec# programming system. The result is a program, from which we can obtain a *cross-proof*, which verifies that the refinement process generates a program, which correctly implements its contract.

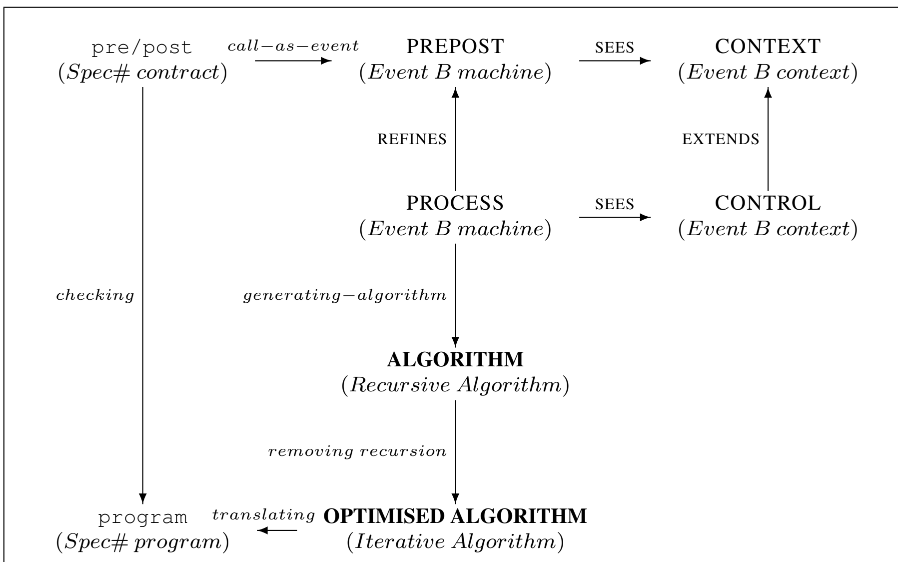


Figure 2: The specification and implementation of an algorithm containing a loop.

The refinement square (with nodes PREPOST, CONTEXT, PROCESS and CONTROL) in Figure 2, provides the mechanism for deriving annotations via refinement. It can be explained briefly as follows:

- The EVENT B machine PREPOST contains events, which have the same contract as that expressed in the original pre/post contract. This machine SEES the EVENT B CONTEXT, which expresses static information about the machine.

- The EVENT B machine PROCESS refines PREPOST generating a concrete specification that satisfies the contract. This machine SEES the EVENT B context CONTROL, which adds control information for the new machine.
- The labelled actions REFINES, SEES and EXTENDS, are supported by the RODIN platform and are checked *completely* using the proof assistant provided by RODIN.

The result of the refinement is the EVENT B machine PROCESS, which contains the refined events and the proof obligations that must be discharged in order to prove that the refinement is correct. The transformation of this EVENT B machine PROCESS into a concrete iterative OPTIMISED ALGORITHM is achieved via two transformations which we present in the sections that follow:

1. Transformation of an EVENT B machine into a concrete recursive algorithm (represented by the arrow labelled *generating-algorithm*).
2. Transformation of this recursive algorithm into its equivalent partially annotated and iterative algorithm (represented by the arrow labelled *removing recursion*).

4 Generating a recursive algorithm from the EVENT B machine

As seen in Figure 2 the result of the refinement is a concrete machine which contains events and their associated proof obligations. Our approach to generating the recursive algorithm depends on the format of these events and is determined by using the pre/post contract of the calling procedure. The event's format may be deterministic, may contain a recursive call of the *procedure under development*, or may contain a call to another procedure which may (or may not) be already developed. The translation of each event into a computable structure is based on a systematic transformation using control labels. Each event is characterised by a current label and a next label. These control labels are added as *annotations* in the event, their purpose being to *simulate* the different steps of the computation. The computation of the recursive algorithm is described by the acyclic graph of labels describing the set of events used in the computation.

4.1 Generating the machines computation graph

Each event e annotates one link in the computation graph, joining nodes that represent the event's pre and post labels. If e annotates the link $\ell_1 \xrightarrow{e} \ell_2$, then the guard of e contains a predicate $\ell = \ell_1$ and the action of e contains $\ell := \ell_2$. From a label ℓ_1, the set of possible *events* that can be observed is denoted by $\mathcal{E}(\ell_1)$. The set of *target labels*,

$\mathcal{L}(\ell_1)$, are those labels that can be directly reached from ℓ_1 by an event of $\mathcal{E}(\ell_1)$. The graph of labels annotated by events, denoted $(\mathcal{L}, \mathcal{E}, \longrightarrow)$, is built in a way such that the label $start$ (representing the initial event) has no incoming labels and the label end has no outgoing labels. A further property of the graph is that, for any label $\ell \in \mathcal{L}$, there is a path from label $start$ to end via label ℓ. Moreover, the graph has no cycles since we use recursive calls. This acyclic nature of the graph leads to a recursive version of the algorithm that implements the specification.

For every label in the graph there exists, by construction, a bottom label. The bottom label satisfies the following property: If there is a path from ℓ_1 to ℓ_2 via each $\ell_3 \in \mathcal{L}(\ell_1)$, and a path that leads directly from ℓ_1 to ℓ_2 then the bottom label ℓ_2 is unique. This bottom label is denoted by $\perp(\ell_1)$.

4.2 Deriving the Recursive Program

The next step is to derive a programming structure from the graph. As the graph is acyclic, we derive a program that initially consists only of if statements as illustrated. If we consider the label ℓ_1 we have the following general pattern:

- The set of labels in $\mathcal{L}(\ell_1)$ is $\{\ell_{31}, \ldots, \ell_{3n}\}$.
- The guard of the event labelling the link from ℓ_1 to ℓ_{3i} is denoted by $\ell = \ell_1 \wedge g_{\ell_1,\ell_{3i}}(x)$ where x is a variable parameter of the guard.
- The sentence $\ell = \ell_1$ is removed in the translation.
- $\mathsf{comp}_{\ell_{3i}}$ denotes the result of the translation from ℓ_{3i}.

$/ * \ell = \ell_1$
IF $\mathsf{act}_{\ell_{31}}$
  $/ * \ell = \ell_{31}$
  $\mathsf{comp}_{\ell_{31}}$
ELSIF $\mathsf{act}_{\ell_{32}}$
  $/ * \ell = \ell_{32}$
  $\mathsf{comp}_{\ell_{32}}$
ELSIF $\mathsf{act}_{\ell_{3i}}$
  $/ * \ell = \ell_{3i}$
  $\mathsf{comp}_{\ell_{3i}}$
  ...
ELSE $\mathsf{act}_{\ell_{3n}}$
  $/ * \ell = \ell_{3n}$
  $\mathsf{comp}_{\ell_{3n}}$
FI
$/ * \ell = \perp(\ell_1)$

The translation process uses the labelled graph of events for translating events into *programming* structures (hence defining what the statements $\mathsf{act}_{\ell_{3i}}$ are). It achieves this by applying a rule for each label ℓ, in each of the three following scenarios: basic events, recursive calls and non recursive calls. We discuss these three scenarios below.

4.2.1 Case 1:Basic Events

If the event e is a basic event controlling the state of the variable x, guarded by $g_{\ell_1,\ell_2}(x)$ and modified by the assignment $x := f_{\ell_1,\ell_2}$ where f is a function, the event e takes the form below.

```
EVENT e
  WHEN
    ℓ = ℓ1
    g_{ℓ1,ℓ2}(x)
  THEN
    ℓ := ℓ2
    x := f_{ℓ1,ℓ2}(x)
  END1
```

The translation omits the control variable ℓ, introduces an IF statement using the guard as the condition and translates the assignment to the target programming language as long as the function f_{ℓ_1,ℓ_2} is implementable. If the event e labels the link $\ell_1 \xrightarrow{e} \ell_2$ then the statement act_{ℓ_2} is defined as WHEN $g_{\ell_1,\ell_2}(x)$ THEN $x := f_{\ell_1,\ell_2}(x)$.
The function f_{ℓ_1,ℓ_2} must be deterministic and translated by an expression definable in some programming language. The EVENT B models are designed to satisfy this hypothesis. If the writer of the models can not remove the non-determinism, the event falls into the two possible next categories.

4.2.2 Case 2: Recursive Call of the Procedure

```
EVENT rec%PROC(h(x),y)%P(y)
ANY y
WHEN
  ℓ = ℓ1
  g_{ℓ1,ℓ2}(x, y)
THEN
  ℓ := ℓ2
  x := f_{ℓ1,ℓ2}(x, y)
END1
```

The definition of the event e is not executable and the translation is driven by instances of the control variable ℓ in the guard (as $\ell = \ell_1$) and in the assignment ($\ell := \ell_2$). The statement act_{ℓ_2} is therefore defined as: $\mathsf{PROC}(h(x), y)$. The choice of the event name is the responsibility of the writer of the EVENT B models, who must identify the case corresponding to a recursive call. RODIN authorizes any string and we choose to indicate as much as possible the category of the event (using the keyword rec) to facilitate the translation into the programming language. The name is meaningful and annotates the EVENT B models. Note that it is possible that other occurrences of $rec\%\mathsf{PROC}(h(x), y)\%$ start from the same label and lead to the same post-label. For instance, if the postcondition $P(y)$ holds in one possible event, then another event, with the same pre-label and post-label, may occur with $\neg P(y)$. In this case, the two events are translated into one call.

4.2.3 Case 3: Non Recursive Call

In the third and final case, the event e can be transformed into a call of another procedure. The call is expressed by an event e, which we name $call\%\mathsf{APROC}(h(x), y)\%P(y)$ and the statement act_{ℓ_2} is defined as $\mathsf{APROC}(h(x), y)$.

```
EVENT call%APROC(h(x),y)%P(y)
ANY y
WHEN
  ℓ = ℓ1
  g_{ℓ1,ℓ2}(x, y)
THEN
  ℓ := ℓ2
  x := f_{ℓ1,ℓ2}(x, y)
END1
```

However, the procedure APROC should already be defined (or at least specified) by an EVENT B machine PREPOST. We consider that there is a tree-like structure of subprocedures under development and we develop the identified procedure APROC in the same way. This last case provides a way to define a hierarchical structure of procedures, which are developed using the same methodology.

In summary, the annotated, and possibly recursive algorithm ALGORITHM is derived from the PROCESS machine by a systematic transformation using the control labels to *simulate* the different steps of the computation. The next step is the transformation of ALGORITHM, into a partially annotated and non recursive OPTIMISED ALGORITHM. This transformation will be presented in the section that follows.

5 Transforming the recursive algorithm into an iterative one

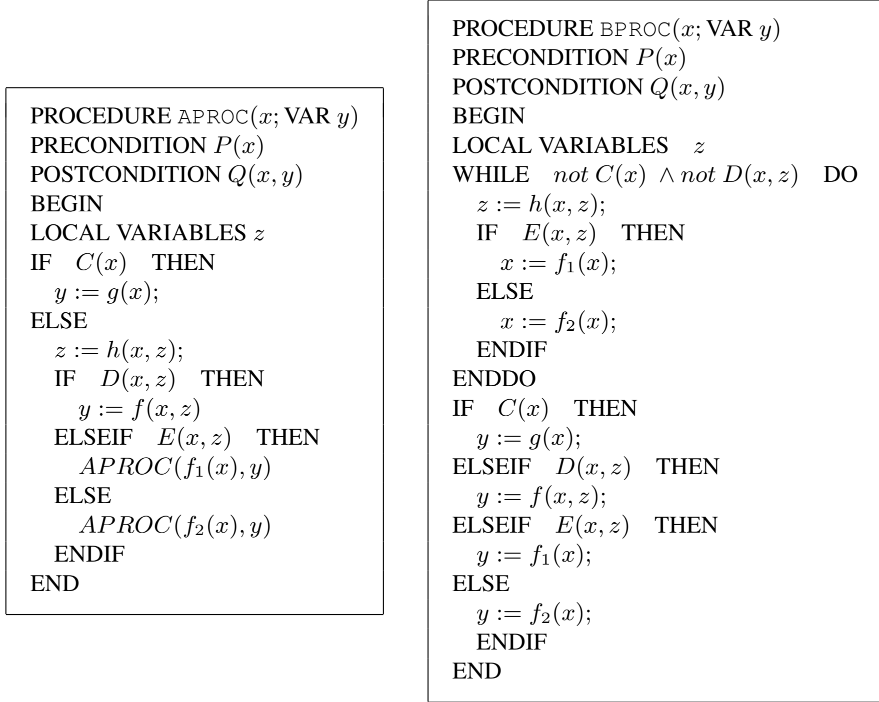
A recursive procedure named APROC can be transformed into a non-recursive procedure named BPROC via a transformation $\mathcal{T}$ as follows $\mathcal{T}(\text{APROC}) = \text{BPROC}$. The source and target of the transformation are stated below in Figure 3. The transformation $\mathcal{T}$ produces a new procedure without *recursive calls* and preserves the partial correctness with respect to the pre and post specification. It must preserve the *operational* semantics of the algorithms. This is, if $[\![A]\!]$ is the function denoting the procedure A then $[\![\mathcal{T}(A)]\!] = [\![A]\!]$. The permitted states are expressed as the set $\Sigma = V \rightarrow \text{VALUES}$, where V is the set of variables of the procedure and VALUES are their values.

Theorem 1. *The transformation is sound with respect to the pre and post specification.*

We consider the macro-expansion corresponding to the call $APROC(x, y)$ leading to the set of variables as $V = x \cup z \cup y$ where the initial values of x, y, and z are x_0, y_0, and z_0. We prove that $[\![APROC]\!] = [\![BPROC]\!]$ by considering two cases.

CASE 1 No iteration: Consider that $C(x_0)$ is true. Then $[\![BPROC]\!](x_0, y_0, z_0) = g(x_0)$, since the WHILE loop is not possible and the post processing leads to $y = g(x_0)$.

CASE 2 Iteration: Consider that a sequence of values of x, namely $x_0 \ldots x_n$, lead to a value such that either $C(x_n)$ is true or $D(x_n)$ is true, causing the loop to stop iterating. The relation between these values are defined by two sub-cases as follows:



```
PROCEDURE APROC(x; VAR y)
PRECONDITION P(x)
POSTCONDITION Q(x, y)
BEGIN
LOCAL VARIABLES z
IF  C(x)  THEN
  y := g(x);
ELSE
  z := h(x, z);
  IF  D(x, z)  THEN
    y := f(x, z)
  ELSEIF  E(x, z)  THEN
    APROC(f1(x), y)
  ELSE
    APROC(f2(x), y)
  ENDIF
END
```

```
PROCEDURE BPROC(x; VAR y)
PRECONDITION P(x)
POSTCONDITION Q(x, y)
BEGIN
LOCAL VARIABLES  z
WHILE  not C(x) ∧ not D(x, z)  DO
  z := h(x, z);
  IF  E(x, z)  THEN
    x := f1(x);
  ELSE
    x := f2(x);
  ENDIF
ENDDO
IF  C(x)  THEN
  y := g(x);
ELSEIF  D(x, z)  THEN
  y := f(x, z);
ELSEIF  E(x, z)  THEN
  y := f1(x);
ELSE
  y := f2(x);
  ENDIF
END
```

Figure 3: Transformation

Sub-case 2.1: The sequence is terminated by $C(x_n)$ and no value of (x_i, z_i) satisfies $D(x_i, z_i)$. Hence the following properties hold:

1. $\forall i \in 0..n-1.z_{i+1} = h(x_i, z_i)$
2. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 1$ if $E(x_{i-1}, z_{i-1})$
3. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 2$ if $\neg E(x_{i-1}, z_{i-1})$
4. $\forall i \in 0..n.\neg D(x_i, z_i)$

The result y is therefore set by the statement $y = g(x_n)$. Hence $[\![APROC]\!](x_0) = g(x_n)$.

Next, we consider the procedure BPROC executed under the same conditions: We build the same sequence of values for x and z, which ensures that the loop terminates when $C(x_n) \wedge \neg D(x_n, z_n)$. Since the value x_n satisfies $C(x_n)$, the next statement

executed is $y := g(x)$ giving y the final value $g(x_n)$. Therefore, $[\![APROC]\!](x_0) = [\![BPROC]\!](x_0)$.

Sub-case 2.2: The sequence is terminated by $D(x_n, z_n)$ and no value of x_i satisfies $C(x_i)$. Hence the following properties hold:

1. $\forall i \in 0..n-1.z_{i+1} = h(x_i, z_i)$
2. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 1$ if $E(x_{i-1}, z_{i-1})$
3. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 2$ if $\neg E(x_{i-1}, z_{i-1})$
4. $\forall i \in 0..n.\neg C(x_i)$
5. $\forall i \in 0..n-1.\neg D(x_i, z_i)$

The value x_n satisfies $D(x_n, z_n)$ and the result y is therefore set by the statement $y = f(x_n, z_{n+1})$. Hence, $[\![APROC]\!](x_0) = f(x_n, z_{n+1})$. Similar reasoning on BPROC leads to termination of the loop when $D(x_n, z_n)$ and $\neg C(x_n)$. The next statement executed is the assignment $y = f(x_n, z_{n+1})$. Hence, $[\![APROC]\!](x_0) = [\![BPROC]\!](x_0)$.

By the reasoning applied to both cases above, the overall transformation of APROC into BPROC is sound. We illustrate our approach by using a transformation for removing recursion in a given case. In the next section, we illustrate the approach on the binary search problem.

6 Case Study: Binary Search Problem

The binary search problem is a classic algorithmic problem. We reformulate the development of a solution and illustrate the use of the transformation rules for removing recursive calls from the algorithm generated from the EVENT B machine.

6.1 Specifying the binary search problem

The input parameters of the *binsearch* procedure are: a sorted array t; the bounds of the array within which the algorithm should search (lo and hi); and the value for which the algorithm should search (val). Output parameters are $result$ and a boolean flag ok that indicates if $t(result) = val$. The procedure pre and post conditions are presented below in Algorithm 1.

The array t is sorted with respect to the ordering over integers and a simple inductive analysis is applied leading to a binary search strategy. The specification is first expressed by two events corresponding to the two possible cases: either a key exists in the array t containing the value val, or there is no such key. These two events correspond to the two possible resulting *calls* to the procedure $binsearch(t, val, lo, hi; ok, result)$:

Algorithm 1: $binsearch(t, val, lo, hi, ok, result)$

$$\textbf{precondition} : \begin{pmatrix} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi-1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \\ l, h \in 0..t.Length \\ lo \leq hi \end{pmatrix}$$

$$\textbf{postcondition} : \begin{pmatrix} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

- EVENT find is $binsearch(t, val, lo, hi; ok, result)$ with $ok = TRUE$
- EVENT fail is $binsearch(t, val, lo, hi; ok, result)$: with $ok = FALSE$

```
EVENT find
  ANY  j
  WHERE
    grd1 : j ∈ lo .. hi
    grd2 : t(j) = val
  THEN
    act1 : ok := TRUE
    act2 : i := j
  END
```

```
EVENT fail
  WHEN
    grd1 : ∀k·k ∈ lo .. hi ⇒ t(k) ≠ val
  THEN
    act1 : ok := FALSE
END
```

These two events form the machine called *binsearch1* (which corresponds to the PREPOST machine of Figure 2). This machine is refined to obtain *binsearch2* (which corresponds to PROCESS of Figure 2). This refined machine contains a new control variable, l, which *simulates* how the binary search is achieved.

6.2 Refinement for Computation

The two events EVENT find and EVENT fail are refined according to the following diagram. Note that computations are controlled by the new control variable l, which takes on the values $start$, $middle$ and end to define the possible computation paths of the algorithm. We consider eight possible scenarios within this refinement diagram:

1. $$\begin{pmatrix} l = start \\ lo = hi \\ t(lo) = val \end{pmatrix} \xrightarrow{m_1} \begin{pmatrix} l = end \\ lo = hi \\ ok = TRUE \ \wedge \ result = lo \end{pmatrix}$$

2. $$\begin{pmatrix} l = start \\ lo = hi \\ t(lo) \neq val \end{pmatrix} \xrightarrow{m_2} \begin{pmatrix} l = end \\ lo = hi \\ ok = FALSE \end{pmatrix}$$

3. $$\begin{pmatrix} l = start \\ lo < hi \end{pmatrix} \xrightarrow{split} \begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \end{pmatrix}$$

4. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val < t(mi) \end{pmatrix} \xrightarrow{rec(lo,mi-1,val,ok,result)} \begin{pmatrix} l = end \\ ok = TRUE \wedge t(result) = val \end{pmatrix}$$

5. $$\begin{pmatrix} l = middle \\ lo < hi \\ val < t(mi) \end{pmatrix} \xrightarrow{rec(lo,mi-1,val,ok,result)} \begin{pmatrix} l = end \wedge ok = FALSE \\ \wedge(\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

6. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val = t(mi) \end{pmatrix} \xrightarrow{m_3} \begin{pmatrix} l = end \\ ok = TRUE \ \wedge \ result = mi \end{pmatrix}$$

7. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val > t(mi) \end{pmatrix} \xrightarrow{rec(mi+1,hi,val,ok,result)} \begin{pmatrix} l = end \\ ok = TRUE \wedge \ t(result) = val \end{pmatrix}$$

8. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val > t(mi) \end{pmatrix} \xrightarrow{rec(mi+1,hi,val,ok,result)} \begin{pmatrix} l = end \wedge ok = FALSE \\ \wedge(\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

Each of these scenarios are used to generate the refined events in the concrete machine *binsearch2* with event names corresponding to the labels on the arrows in each scenario.

6.3 Generating the algorithm from events

The events of the machine called *binsearch2* are listed below. Note that events $m1$, $m2$, $split$ and $m3$ correspond directly with scenarios 1, 2, 3 and 6.

EVENT m1 REFINES find
WHEN
$grd1 : l = start$
$grd2 : lo = hi$
$grd3 : t(lo) = val$
WITNESSES
$j : j = lo$
THEN
$act1 : l := end$
$act2 : ok := TRUE$
$act3 : i := lo$
END

EVENT m3 REFINES find
WHEN
$grd1 : l = middle$
$grd3 : t(mi) = val$
WITNESSES
$j : j = mi$
THEN
$act1 : l := end$
$act2 : ok := TRUE$
$act3 : i := mi$
END

EVENT m2 REFINES fail
WHEN
$grd1 : l = start$
$grd2 : lo = hi$
$grd3 : t(lo) \neq val$
THEN
$act1 : l := end$
$act2 : ok := FALSE$
END

EVENT split
WHEN
$grd1 : l = start$
$grd2 : lo < hi$
THEN
$act1 : l := middle$
$act2 : mi := (lo + hi)/2$
END

Scenarios 4 and 5 correspond to the case where $val < t(mi)$ and hence the search continues on the left part of the array. Two events will be generated for this case: one where the value is found $(OK = true)$, and one where the value is not found $(OK = false)$. Similarly scenarios 7 and 8 correspond to the case where $val > t(mi)$ and the search continues on the right part of the array. Again, two events will be generated for this case: one where the value is found $(OK = true)$, and one where the value is not found $(OK = false)$. We illustrate two of the four events below.

EVENT rightsearchOK
  REFINES find
  ANY j
  WHERE
    $grd1 : l = middle$
    $grd2 : val > t(mi)$
    $grd3 : j \in mi + 1 \,..\, hi$
    $grd4 : t(j) = val$
    $grd5 : mi + 1 \leq hi$
  THEN
    $act1 : i := j$
    $act2 : ok := TRUE$
  END

EVENT rightsearchKO REFINES fail
  WHEN
    $grd1 : l = middle$
    $grd2 : val > t(mi)$
    $grd4 : \forall j \cdot j \in mi + 1 \,..\, hi \Rightarrow t(j) \neq val$
    $grd5 : mi + 1 \leq hi$
  THEN
    $act2 : ok := FALSE$
  END

We identify that the translation from EVENT B into an algorithmic notation introduces new proof obligations. These proof obligations state that the call is correct [15] i.e. the current state implies that the precondition is true. In the case of our example, we prove that each guard of rightsearchOK and rightsearchKO implies the precondition of the algorithm: the theorems labelled th_{call1} and th_{call2}, in the invariant below, express the discharged conditions.

Using the control variable l, we can apply our **generating-algorithm** transformation to produce the recursive algorithm. The result is Algorithm 2 below which is derived from the refined EVENT B model *binsearch2*. The control variable l is removed by introducing control via if statements.

Finally, using RODIN we prove that the following assertion is an invariant for our model and hence it can be used for inferring the invariant of the generated code.

$inv1 : i \in 1 \,..\, n$
$inv2 : l \in LOC$
$inv3 : dom(t) = 1 \,..\, n$
$inv4 : mi \in 1 \,..\, n$
$inv5 : l = middle \Rightarrow lo < hi \wedge mi \in lo \,..\, hi$
$inv6 : l = middle \wedge val < t(mi) \Rightarrow (\forall k \cdot k \in mi \,..\, hi \Rightarrow t(k) \neq val)$
$inv7 : l = middle \wedge val > t(mi) \Rightarrow (\forall k \cdot k \in lo \,..\, mi \Rightarrow t(k) \neq val)$
$inv8 : l = end \wedge ok = TRUE \Rightarrow i \in lo \,..\, hi \wedge t(i) = val$
$inv9 : l = end \wedge ok = FALSE \Rightarrow (\forall k \cdot k \in lo \,..\, hi \Rightarrow t(k) \neq val)$
$inv10 : lo \,..\, hi \subseteq 1 \,..\, n$
$th_{call1} : (\exists j \cdot l = middle \wedge j \in mi + 1 \,..\, hi$
$\quad \wedge key > t(mi) \wedge t(j) = key \wedge mi + 1 \leq hi) \Rightarrow mi + 1 \leq hi$
$th_{call2} : (\exists j \cdot l = middle \wedge j \in lo \,..\, mi - 1$
$\quad \wedge key < t(mi) \wedge t(j) = key \wedge lo \leq mi - 1) \Rightarrow lo \leq mi - 1$

Algorithm 2: Recursive Algorithm binsearch(t,val,lo,hi,ok,result).

precondition : $\left(\begin{array}{l} n \in \mathbb{N}1 \wedge lo, hi \in dom(t) \wedge lo \leq hi \\ t \in 0 \,..\, n-1 \rightarrow \mathbb{N} \wedge \forall i.i \in 0..n-2 \Rightarrow t(i) \leq t(i+1) \end{array}\right)$

postcondition : $\left(\begin{array}{l} ok = true \Rightarrow t(result) = val \\ ok = false \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{array}\right)$

local variables: $mi \in \mathbb{Z}$

$start : \left(\begin{array}{l} n \in \mathbb{N}1 \wedge lo, hi \in dom(t) \wedge lo \leq hi \\ t \in 0 \,..\, n-1 \rightarrow \mathbb{N} \wedge \forall i.i \in 0..n-2 \Rightarrow t(i) \leq t(i+1) \end{array}\right)$

if $lo = hi \wedge t(lo) = val$ **then**
- $result := lo; ok := true;$

else
- **if** $lo = hi \wedge t(lo) \neq val$ **then**
  - $ok := false;$
- **else**
  - $mi := (lo + hi)/2;$
  - $middle : \left\{\left(\begin{array}{l} mi = (lo + hi)/2 \\ val < t(mi) \Rightarrow \forall k.k \in mi..hi \Rightarrow t(k) \neq val \\ val > t(mi) \Rightarrow \forall k.k \in lo..mi \Rightarrow t(k) \neq val \end{array}\right)\right\}$
  - **if** $t(mi) = val$ **then**
    - $result := mi; ok := true;$
  - **else**
    - **if** $mi + 1 \leq hi \wedge t(mi) < val$ **then**
      - binsearch(t,val,mi+1,hi,ok,result);
    - **else**
      - **if** $lo \leq mi - 1 \wedge t(mi) < val$ **then**
        - binsearch(t,val,lo,mi-1,ok,result);
      - **else**
        - ok:=FALSE

;

$end : \left\{\left(\begin{array}{l} ok = true \Rightarrow t(result) = val \\ ok = false \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{array}\right)\right\};$

6.4 Transforming the Binary Search Recursive Procedure

In order to generate the iterative version of our algorithm, we apply the **removing recursion** transformation. We identify the rules condition C as

$$\left(\begin{array}{l} lo = hi \wedge t(lo) = val \\ \vee\ lo = hi \wedge t(lo) \neq val \\ \vee\ lo < hi \wedge mi = (lo+hi)/2 \wedge t(mi) = val \end{array}\right)$$

and obtain PROCEDURE $binsearch(t, val, lo, hi, ok, result)$ as presented in Figure 4.

6.5 Interpreting the algorithms within Spec#

In order to fully utilize our integrated development framework for refinement based program verification, we have translated the resulting iterative algorithm into Spec#. As shown in Figure 5 this is almost a one-to-one mapping. The main difference in the algorithms is that we simply return a value of -1 when our iterative algorithm sets OK to $false$ and return the index where the value is found when our iterative algorithm sets OK to $true$. The algorithm verified as correct, in less than 2 seconds using the Spec# programming system (version 2011-10-03). No user interaction is required in the verification as all assertions required (preconditions, postconditions and loop invariants) have been generated as part of the refinement and transformation of the initial abstract specification into the final iterative algorithm. It is interesting to note that, prior to formalising our transformation rules, our initial attempt at writing this iterative C# program contained an error. This error in the loop body, was due to our omission to check that the values of $mi + 1$ and $mi - 1$ were within the array bounds before narrowing the search space. This error was immediately detected by the Spec# programming system. The automatic verification of the final program is available online at `http://www.rise4fun.com/SpecSharp/psP4`.

This verification step acts as an insurance check for the *correct-by-construction* approach in two ways. Firstly, while the Event B framework provides for the automatic verification of some proof obligations, many proofs require the user to manually interact with the tools to provide guidance.This often leads to error, typically introduced by incorrect assumptions made by the user while proving a proof obligation. Having an alternative verification tool that automatically verifies that the final implementation is correct with respect to its specification re-assures the developer that their interactions were correct at each stage of the development. Secondly, Event B is a modelling language where data types, event guards and actions are *logical* structures based on set theory. While the *programming* structures of Spec# have logical features, they also have programming constraints that must be taken into account when translation from the resulting iterative algorithm to a programming langauge. The *cross* verification increases trust in the final product ensuring that the semantics of the original specification is maintained.

PROCEDURE $binsearch(t, val, lo, hi, ok, result)$

PRECONDITION $\left(\begin{array}{l} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi-1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \wedge lo, hi \in 0..t.Length \wedge lo \leq hi \end{array}\right)$

POSTCONDITION $\left(\begin{array}{l} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{array}\right)$

BEGIN

WHILE $not \left(\begin{array}{l} lo = hi \wedge t(lo) = val \\ \vee\, lo = hi \wedge t(lo) \neq val \\ \vee\, lo < hi \wedge mi = (lo+hi)/2 \wedge t(mi) = val \end{array}\right)$ DO

$mi := (lo+hi)/2;$

$middle : \left\{\left(\begin{array}{l} mi = (lo+hi)/2 \\ val < t(mi) \Rightarrow \forall k.k \in mi..hi \Rightarrow t(k) \neq val \\ val > t(mi) \Rightarrow \forall k.k \in lo..mi \Rightarrow t(k) \neq val \end{array}\right)\right\}$

IF $mi+1 \leq hi \wedge val > t(mi)$ THEN

$lo := mi+1$

ELSEIF $lo \leq mi-1 \wedge val < t(mi)$ THEN

$hi := mi-1$

ENDDO

IF $lo = hi \wedge t(lo) = val$ THEN

$result := lo; ok := true$

ELSEIF $lo = hi \wedge t(lo) \neq val$ THEN

$ok := false$

ELSEIF $lo < hi \wedge t(mi) = val$ THEN

$result := mi; ok := true$

ELSE $ok := false$

ENDIF

END

Figure 4: PROCEDURE $binsearch(t, val, lo, hi, ok, result)$

7 Related Work

The topic of program transformation, and in particular, the transformation of recursive programs to iterative ones is not new. In 1965, Gordon [16] investigated the tranformation of *recursive relations to recurrence or iterative relations*. He also addressed the transformation of non primitive recursive functions (namely Ackerman's function) into iterative functions, opening a new domain of research on transformations and promoting the use of recursive definitions of algorithms. Later Strong [9] specified the problem of transforming *recursive equations* into iterative equations expressed by flowcharts while

```
class BS {
 int BinarySearch(int[] t, int val, int lo, int hi, bool ok)
  requires 0 <= lo && lo < t.Length && 0 <= hi
                                   && hi < t.Length;
  requires lo <= hi && 0 < t.Length;
  requires forall {int i in (0:t.Length),
                  int j in (i:t.Length); t[i] <= t[j]};
  ensures -1 <= result && result < t.Length;
  ensures (0 <= result && result < t.Length)==>
                                          t[result] == val;
  ensures result == -1 ==>
                   forall {int i in (lo..hi); t[i] != val};
 {
      int mi = (lo + hi) / 2;
      while (!(lo == hi && t[lo] == val)
                    || ( lo == hi && t[lo] != val)
              || (lo < hi && (mi == (lo + hi) /2)
                          && t[mi] == val))
       invariant 0 <= lo && lo < t.Length && 0 <= hi
                                         && hi < t.Length;
       invariant 0 <= mi && mi < t.Length;
       invariant (val < t[mi]) ==>
                   forall {int i in (mi..hi); t[i] != val};
       invariant (val > t[mi]) ==>
                   forall {int i in (lo..mi); t[i] != val};
      {
          mi = (lo + hi) /2;
          if  ((mi+1 <= hi) && (val > t[mi])) lo = mi +1;
          else if ((lo <= mi-1) && (val < t[mi]))
                                             hi = mi - 1;
      }
      if ((lo == hi) && (t[lo] == val))
                                    {ok = true; return lo;}
      else{
          if ((lo == hi) && (t[lo] != val))
                                  {ok = false; return -1;}
          else if ((lo < hi) && (t[mi] == val))
                                   {ok = true; return mi;}
               else {ok = false; return -1;}
     }
 }
}
```

Figure 5: Binary Search C# program corresponding to the generated iterative procedure.

Pettorossi [13], aimed to improve a program's memory usage. Research by Darlington and Burstall [4] proposed a list of transformations, which can be automatically applied for removing recursive calls from a program.

Our work does not claim to discover new transformations. Instead we have extended these transformations within EVENT-B models to integrate a *correctness* phase in the transformation. Our work relates two complementary frameworks: the programming framework of C# and the modelling framework of EVENT-B. We promote the development of annotated programs and present this as the main contribution over previous work. We consider that the recursive algorithms are easier to generate using our approach and they can be easily transformed to get an efficient iterative solution.

In our seminal work on code generation [12] we have developed a shortest path algorithm based on the *dynamic programming* paradigm where the discovery of program invariants utilises the underlying inductive properties of the algorithm. The operational aspect of the iterative solution is useful for improving the quality and efficiency of the resulting code. This is our motivation for transforming our recursive algorithms into iterative ones, obtaining their correctness for free using our integrated development framework, that brings together the world of system modelling and the world of program verification.

8 Conclusion

We have presented and verified the correctness of two transformation rules, which transform EVENT B models into iterative algorithms. The resulting algorithms are correct-by-construction and can be directly mapped into an executable programming language. We provide a cross-proof by verifying the correctness of our final program using the Spec# programming system. Our integrated development framework (Figure 2) indicates where our transformations are used for producing a program that is *correct-by-construction*. The translation of the PROCESS machine into a recursive algorithm is straightforward and removes the control variable used to relate events when generating the code.

This work builds on a method for code generation that is detailed by one of the authors in [11, 12] and provides the foundation for an integrated development framework that brings together the world of system modelling and the world of program verification. The EB2ALL code generation tool [10] can also produce a program from the PROCESS machine. However, the control variable is not removed and the resulting code is not structured. The advantage of our approach is the production of a structured iterative program, which can be automatically verified using the Spec# programming system.

Our experience shows that our approach assists students in developing and understanding the tasks of software specification and verification. It also makes different

forms of formal software development more accessible to the Software Engineers, helping them to build correct and reliable software systems. Future work will include the development of adequate plugins, which will integrate and facilitate the co-operation between Spec# tools and RODIN tools.

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[3] ClearSy, Aix-en-Provence (F). *Atelier B*, 2002. Version 3.6.

[4] J. Darlington and R.M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[5] Leonardo de Moura and Nikolaj Björner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, 2008.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[7] Gary T. Leavens et al. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.

[8] Mike Barnett et al. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[9] H.R. Strong Jr. Translating recursion equations into flow charts. *Journal of Computer and System Sciences*, 5(3):254 – 285, 1971.

[10] D. Méry and N. Singh. eb2all.loria.fr, 2011.

[11] Dominique Méry. A simple refinement-based method for constructing algorithms. *ACM SIGCSE Bulletin*, 41(2):51–59, 2009-06.

[12] Dominique Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3):197–239, 2009-09.

[13] Alberto Pettorossi. Improving memory utilization in transforming recursive programs (extended abstract). In Józef Winkowski, editor, *MFCS*, volume 64 of *LNCS*, pages 416–425. Springer, 1978.

[14] Project RODIN. Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/, 2004.

[15] John C. Reynolds. *The Craft of Programming*. Prentice-Hall International series in computer science. Prentice-Hall International, 1982.

[16] H. Gordon Rice. Recursion and iteration. *Commun. ACM*, 8(2):114–115, 1965.

Ping-Pong Protocols as Prefix Grammars and Turchin Relation

Antonina Nepeivoda

Program Systems Institute, Pereslavl-Zalessky, Russia
a_nevod@mail.ru

Abstract

This paper describes how to verify cryptographic protocols by a general-purpose program transformation technique with unfolding. The questions of representation and analysis of the protocols as prefix rewriting grammars are discussed. In these aspects Higman and Turchin embeddings on computational paths are considered, and a refinement of Turchin's relation is presented that allows to algorithmically decide the empty word problem for prefix rewriting grammars.

1 Introduction

It is known that even in the case when algorithms of message encryption themselves are considered as completely secure some existing cryptographic protocols that use them may be insecure. Many vulnerabilities in the protocols appear due to the use of common communication channel that can be listened and analyzed by someone other than the legal participants of the interaction. Although the problem of automatic verification of such interactions is undecidable in general [1], there were described some classes of protocols for which the verification task can have a decision procedure. In particular, Dolev and Yao presented the ping-pong model of the cryptographic protocols [2]. In a ping-pong protocol a message is a single data item encrypted by a sequence of keys. Principals can apply a finite number of encryption and decryption operations to the message. Dolev, Yao and Karp showed that this protocol model can be verified in a polynomial time if an intruder can listen the communication channel between agents and participate in the interaction on every its stage [3]. Then the question appears whether these verification results can be used while modelling the protocols not only by special tools but by general-purpose program transformation tools.

A ping-pong protocol can be naturally presented as a prefix rewriting grammar. The prefix-rewriting grammars are also used as function stack abstractions in construction of loop approximations in program analysis (for example, in V. Turchin's works on supercompilation [10]). The main disctinction is that the stack operations in supercompilation are modelled by a smaller class of the prefix

grammars than the ping-pong protocols. But since Turchin's statements remain provable for the wider class of prefix grammars [7], the distinction becomes insignificant.

In this paper we show how to solve the verification problem by the general purpose program transformation technique without constructing any additional tools. The only two actions to be performed by a technique are:

1. Unfolding a computational tree of a program.

2. Terminating too long computational paths in the tree to avoid infinite unfolding [1].

This paper is organized as follows. First, we introduce the classical approach to ping-pong protocol verification that uses finite automata. Then we discuss different types of protocol representations as prefix rewriting grammars. After that we show what difficulties appear in the verification of protocols as prefix grammars by program transformation techniques that use unfolding together with the scattered subword relation as a path termination criterion and discuss how to avoid these difficulties by a small refinement of the criterion. Last, we describe a way to construct prefix grammars with very long minimal tracks that end with the empty word (these tracks represent attacks on the corresponding protocol).

Our contribution is the following:

1. We show that Higman condition of path termination is too weak for successful verification of ping-pong protocols (from the class introduced in [3]) in the form of the prefix grammars.

2. Basing on the Turchin relation we define a simple refinement for this condition. We prove that this refinement allows to verify any ping-pong protocol from the class being considered.

2 Ping-Pong Protocols

Consider an information exchange between several participants that is controlled by some interaction rules. Let Σ_X be a set of actions that are available to a participant X. Some actions from Σ_X, such as encryption, decryption, letter appending, etc, are elementary; these actions do not allow their decomposition and are denoted by single letters. Other actions from Σ_X are compositions of elementary actions that are unavailable to X separately. This can happen, for example, if a participant of interaction is a user of some specific cryptographic

[1] A description of these two techniques can be found in [9].

program that does not allow him/her to use its encryption algorithm without adding his/her personal information to the message to be encrypted. The composite actions from Σ_X are represented as words consisting of letters that denote corresponding elementary actions.

Consider two participants S, R of an interaction. Let us denote an initial single data item as M. Suppose S starts the interaction corresponding to a protocol by sending R the message $\alpha_1(M)$ where $\alpha_1 \in \Sigma_S^*$. R responds by $\alpha_2(\alpha_1(M))$ ($\alpha_2 \in \Sigma_R^*$) and so on until the last action α_n is reached; the message is sent there and back as a ping-pong ball. The tuple $\langle \alpha_1, \ldots, \alpha_n \rangle$ is called a ping-pong protocol.

Definition 1. *A ping-pong protocol $P(S, R)$ (where S and R denote legal participants of the protocol) is a sequence $\Gamma = \langle \alpha_1, \ldots, \alpha_n \rangle$ of operator words, where $\alpha_i \in \Sigma_S^*$ if i is odd and $\alpha_i \in \Sigma_R^*$ if i is even.*

Some elementary actions in a sequence $\alpha_{i+1}\alpha_i$ may partially cancel each other. For example, consider the situation when both of the principals R, S can encrypt a message by two different keys, but each principal knows only one decryption key of the corresponding two. Then $\Sigma_S = \{E_R, E_S, D_S\}$ and $\Sigma_R = \{E_R, D_R, E_S\}$, where E_X means encryption operation and D_X means the corresponding decryption action. Let the protocol be $\langle E_R, E_S D_R \rangle$. If the initial message is M then it is first transformed by S to $E_R(M)$ and then is transformed by R to $E_S(M)$. Thus the sequence $D_R E_R$ collapses to the empty word (denoted by Λ).

The cancellations satisfy the Church–Rosser property and thus can be done in an arbitrary order. So we denote them as rules $R_l \to R_r$, where R_l is a sequence of operations to be transformed and R_r is the result of the transformation. E.g. the previous cancellation rule can be denoted as $D_R E_R \to \Lambda$. Note that the cancellations must not necessarily have the form $xy \to \Lambda$; an action may be cancelled not only by a single other action but also by a sequence of several other actions.

Assume that an intruder Z can read any message x sent from S to R and vice versa and can replace any such message x by a message $\beta(x)$, where $\beta \in \Sigma_Z^*$. A protocol is insecure iff the intruder can get the initial message M.

Definition 2. *A ping-pong protocol $P(S, R) = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ is insecure iff there is a sequence $\langle \beta_1, \beta_2, \ldots, \beta_m \rangle$, $\beta_1 \beta_2 \ldots \beta_m \to \Lambda$ such that $\beta_1 = \alpha_1$, and either $\exists j (\beta_i = \alpha_j)$ and $i \equiv j (mod 2)$ or $\beta_i \in \Sigma_Z^*$. The sequence $\langle \beta_1, \beta_2, \ldots, \beta_m \rangle$ is called an attack on the protocol.*

Note that we allow α_i to appear more than once in an attack since an intruder can initiate multiple interactions with principals.

Example 1. *Consider the following protocol $P_a = \langle ba, b^{-1}\rangle$, where $bb^{-1} \to \Lambda$ and $b^{-1}b \to \Lambda$. Let $\Sigma_Z = \{c, d\}$ such that $c = a^{-1}a^1$ and a^{-1} is the left inverse of a ($a^{-1}a \to \Lambda$ but not $aa^{-1} \to \Lambda$), and $d = ba$. Then the protocol is insecure: Z can get a when it is sent back from R to S, then send baa to R and receive aa which collapses with c.*

In the original paper [3] the following algorithm of protocol verification is introduced. First, we build a nondeterministic finite state automaton that corresponds to the protocol in the following sense.

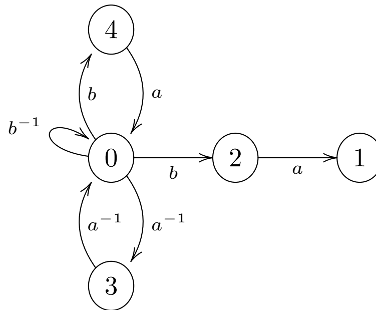
1. State 0 is the unique initial state and state 1 is the unique final state. The input alphabet is $\Sigma = \Sigma_R \cup \Sigma_S \cup \Sigma_Z$.

2. There is a directed path from 0 to 1 whose labels correspond to $\alpha_1(S, R)$.

3. For every input letter $\sigma \in \Sigma_Z$ there is a self-loop from 0 to 0, labelled σ. While we allow not only elementary actions in Σ_Z, the self-loop can contain several edges (in the original work only one-edge loops are considered).

4. For every $\alpha_i \in P$, there is a loop from 0 to 0 whose edges are labelled by the letters of α_i.

Example 2. *Let us build such an automaton for the protocol P_a.*



*Note that the action **ba** is repeated twice in the automaton.*

Let us say that a path p collapses iff its corresponding word collapses to Λ. For example, there is a collapsing path from the state (0 to the state 4) since $b^{-1}b \to \Lambda$. The set of all collapsing paths is denoted by C. To verify the protocol we must investigate whether $(0, 1) \in C$. The following algorithm solves this problem [3].

1. Place in C all the pairs (i, i). Construct a queue Q which also contains all these pairs in an arbitrary order.

2. While $Q \neq \emptyset$ do
   (a) Delete the first pair (i, j) from Q.
   (b) If $(j, k) \in C$, $(i, k) \notin C$, then place (i, k) in C and in Q.
   (c) If $(k, i) \in C$, $(k, j) \notin C$, then place (k, j) in C and in Q.
   (d) If there is an edge $k \to i$ labelled τ and there is an edge $j \to l$ labelled σ, and $\tau\sigma \to \Lambda$ and $(k, l) \notin C$, then place (k, l) in C and in Q.

This algorithm terminates. The final C contains $(0, 1)$ if and only if there is a collapsing path from 0 to 1. But it can be noticed that the described type of automata actually performs stack operations, since a finite number of first symbols in the corresponding word is changed in every loop from 0 to 0. So the automata can be rewritten as grammars of the special kind and it becomes unnecessary to use special algorithms of analysis since there is a wide range of transformation and analysis tools for this kind of the grammars.

Moreover, this automata algorithm has one restriction that naturally disappears when the transition to grammars is made. The restriction is implied from the action `2(d)` of the algorithm. Since all the edges are marked with elementary actions, only cancellations of the form $xy \to \Lambda$ are processed correctly. Thus if we rewrite the conditions $c = a^{-1}a^{-1}$, $a^{-1}a \to \Lambda$ from the Example 1 as $caa \to \Lambda$ (and replace the loop from the state `0` to the state `3` and back to `0` by a self-loop marked c in the corresponding automaton) the algorithm cannot find the attack on P_a. Thus, by the transition to grammar form we become able not only to test a general-purpose program transformation technique on the verification task but also to expand the class of protocols to be verified.

Let us denote letters of an alphabet Σ by the small Latin letters $a, b, c, \ldots, p, q, r$ and the capital Latin letters A, B, C, D, E, F (maybe with subscripts or superscripts), variables that can take some value from Σ as x, y, z, w, and let us denote words from Σ^* by the Greek capitals $\Gamma, \Delta, \Phi, \Psi, \Theta$.

Definition 3. *Consider a tuple $\langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$, where Σ is an alphabet, $\Gamma_0 \in \Sigma^+$ is an initial word and $\mathbf{R} \subset \Sigma^* \to \Sigma^*$ is a set of rewrite rules. If the rewrite rules are applied only to word prefixes $\dfrac{R : \Phi \longrightarrow \Psi}{\Phi\Theta \stackrel{R}{\longrightarrow} \Psi\Theta}$ then the tuple $\langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$ is a prefix rewriting grammar.*

We call a trace of a prefix rewriting grammar $\mathbf{G} = \langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$ a sequence $\{\Phi_i\}$ (finite or infinite), s.t. $\Phi_1 = \Gamma_0$ and $\forall i (i < n \Rightarrow \exists R (R : R_l \to R_r \;\&\; R \in \mathbf{R} \;\&\; \Phi_i = R_l\Theta \;\&\; \Phi_{i+1} = R_r\Theta)$.

Example 3. *Consider the "double protection" protocol P_{RR} from the paper [3] with the following modification. Let $a = E_R$, $b = E_S$, $c = E_Z$, $A = i_R$, $B = i_S$, $C = i_Z$, and i_X be the operation appending the name of X to the message. The*

protocol is $P_{RR} = \langle aBa, b\rangle$. Let $\Sigma_Z = \{a, c, C, c^{-1}, B^{-1}, C^{-1}\}$ and suppose that $x^{-1}x \to \Lambda$ but not $xx^{-1} \to \Lambda$ for every encryption or appending operation x. Thus, only left inverse elements are available.

The actions $aBa \to b$ (the legal interaction between principals), $B^{-1}B \to \Lambda$ (removing the name of the agent S), $\Lambda \to c$ (encryption by the intruder's key) can be considered as rules of a prefix rewriting grammar.

There exists a hierarchy of prefix rewriting grammars (called Caucal hierarchy) that classifies the grammars by the types of their rewriting rules [5]. The Caucal hierarchy of the prefix rewriting grammars is presented in the following table [2].

Type of a grammar	Form of rewriting rules
Type 0	$\Phi \to \Psi$
Type $1\frac{1}{2}$	$pa \to q\Psi$
Type 2	$a \to \Psi$
Type 3	$a \to b \vee a \to \Lambda$

The grammars of the type 2 (and 3) are called alphabetic prefix rewriting grammars in the original Caucal work since their rules can transform only the first letter of a word.

The class $1\frac{1}{2}$ is equivalent to the class 0 (so that if some set of words can be generated by a 0-class grammar then there exists a $1\frac{1}{2}$-class grammar that generates exactly the same set of words) and wider than the class 2. The class 2 is wider than the class 3.

The ping-pong protocols are to be represented as 0- or $1\frac{1}{2}$- prefix grammars. But if we try to use straightforward representation of a ping-pong protocol as a 0-type grammar, some uncertainties can appear.

Example 4. *On the first look, the automaton for the protocol P_a (from the Example 1) can be rewritten as a 0-type prefix grammar as follows.*

$\mathbf{G_{PA}}$:

$R_1 : \Lambda \to a^{-1}a^{-1}$	$R_3 : bb^{-1} \to \Lambda$	$R_5 : \Lambda \to ba$
$R_2 : a^{-1}a \to \Lambda$	$R_4 : b^{-1}b \to \Lambda$	$R_6 : \Lambda \to b^{-1}$

Λ in the left-hand side of a rule means that the rule can be applied to any word.

In the terms of G_{PA} the question of verification is to find out whether a trace starting from the initial word ba and ending by Λ exists. These traces represent attacks on the corresponding protocol; it is unnecessary to construct them all since the existence of even one means that the protocol is insecure.

[2] The "negative" part of the hierarchy (types −2 and −1) is dropped since it is unnecessary for our protocol model.

Despite the fact that the grammar G_{PA} looks rather simple, it is incorrect and generates some non-collapsing words that belong to collapsing paths in the automaton. For example the word $a^{-1}a^{-1}aa$ is never transformed to Λ because this demands to transform the infix $a^{-1}a$ before the first letter a^{-1}, and such actions are forbidden in prefix rewriting grammars.

The well-known way to avoid these difficulties is to use $1\frac{1}{2}$-type prefix rewriting grammars [4]. For every rule $R : pa \to q\Psi$ the letters p and q can appear nowhere but on the first position of a word. The set of these letters represents a set of control states in the corresponding automaton. Then it is possible to force all erasings to be done immediately when they can be done.

Example 5. *The $1\frac{1}{2}$-prefix grammar for the protocol P_A can look as follows ($[N]$ is a single symbol representing a state; the square brackets are introduced for better readability).*

$\mathbf{G_{PA'}}$:

$R_1 : [0]x \to [0]bax$	$R_4 : [0]b \to [0]$	$R_7 : [1]a \to [0]$
$R_2 : [0]x \to [0]a^{-1}a^{-1}x$	$R_5 : [0]a \to [1]$	
$R_3 : [0]x \to [0]b^{-1}x$	$R_6 : [1]x \to [0]a^{-1}x$	

x denotes an arbitrary letter from Σ. This grammar is non-deterministic and also can generate non-collapsing words that correspond to the empty word (for example, after applying R_2 to the word a)[3]. But if there is an attack on the protocol P_A, then the grammar generates some trace ending by Λ that represents this attack.

The representation as $1\frac{1}{2}$-type grammars is especially helpful when used together with the automaton model. But when it is used together with tree unfolding techniques it can demand some additional work, such as constructing a control state alphabet and transforming long rewriting rules in the $1\frac{1}{2}$-form. So in our investigations we use the equivalent 0-form of the grammars.

3 Ping-Pong Protocols as Prefix Grammars

Consider a ping-pong protocol as a prefix grammar in the following sense. Let every elementary action be a letter in the initial alphabet of the prefix grammar. Every action $a_1a_2a_3\ldots a_n$ corresponding to an iteration from the state 0 to itself produces the following rewriting rules

[3]To avoid such cases we must introduce restrictions on x in the rules R_2, R_3, R_6.

$$\begin{array}{lcl} \Lambda & \to & a_1 a_2 \dots a_n \\ a_n^{-1} & \to & a_1 a_2 \dots a_{n-1} \\ a_n^{-1} a_{n-1}^{-1} \dots a_1^{-1} & \to & \Lambda \end{array}$$

In this interpretation all collapsing rules (of the form $a_i a_i^{-1} \to \Lambda$ or $a_i^{-1} a_i \to \Lambda$) can be applied immediately. This does not change properties of the protocol. The prefix grammar is finite (every loop from the state 0 to itself of the length k produces not more than $k+1$ rules). In fact, this grammar repeats the corresponding $1\frac{1}{2}$-type prefix grammar presented in the previous section but does not introduce an auxiliary state alphabet.

Example 6. *The automaton for the protocol P_a is represented by the following 0-type grammar of this sort.*

$\mathbf{G_{PA0}}$:

$R_1 : \Lambda \to ba$	$R_4 : b \to \Lambda$
$R_2 : \Lambda \to a^{-1}a^{-1}$	$R_5 : a \to a^{-1}$
$R_3 : \Lambda \to b^{-1}$	$R_6 : aa \to \Lambda$

This grammar allows doing all cancellations as early as possible. It is non-deterministic as the grammar $\mathbf{G_{PA'}}$ from 5 but contains one rule less because arbitrary left-hand sides are allowed and the rules R_5 and R_6 do not need to be decomposed to the combinations of two rules.

Consider an algorithm unfolding all possible traces of a given prefix grammar $\mathbf{G}$. This algorithm finds all the attacks on the corresponding protocol if they exist but almost all traces are infinite so the algorithm does not terminate. In general-purpose program transformation techniques (for example, supercompilation), that perform such unfoldings, some conditions of when to terminate are introduced to force termination. These conditions are not perfect because they are not specified to prefix rewriting systems and can force early terminations of finite branches (considering them as infinite). In the context of our problem this means that some collapsing path in an automaton may not be found, and the task of verification may not be solved soundly by the technique. So the question raises if the existing termination conditions in, e.g. supercompilation, fit the verification task of ping-pong protocols in the form of prefix rewriting grammars, or they need some refinement.

One of the popular conditions of path termination is the Higman–Kruskal embedding on terms [8]. Now we only define the Higman relation since it operates with words, as the ping-pong protocols do.

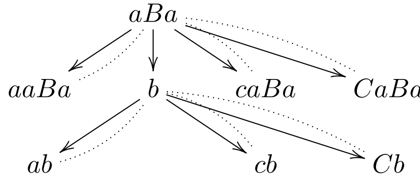
Definition 4. *Given two words in an alphabet* Σ, $\Phi = a_1 a_2 \dots a_m$, $\Psi = b_1 b_2 \dots b_n$, Φ *is embedded in* Ψ *in the sense of Higman relation* ($\Phi \trianglelefteq \Psi$) *iff* Φ *is a subsequence of* Ψ. *This relation is also called a scattered subword relation.*

Example 7. *Consider the "double protection" protocol* P_{RR} *from Example 3. Then the prefix grammar for* P_{RR} *can look as follows.*

$\mathbf{G_{RR}}$:

$R_1 : \Lambda \to a$	$R_4 : aBa \to b$	$R_7 : B \to \Lambda$
$R_2 : \Lambda \to c$	$R_5 : aCa \to c$	$R_8 : C \to \Lambda$
$R_3 : \Lambda \to C$	$R_6 : c \to \Lambda$	

Let the initial word Γ_0 *be* aBa. *If we start unfolding until finding a first such* Γ *and* Δ *that* $\Gamma \trianglelefteq \Delta$, *then the graph for the grammar can look as follows:*



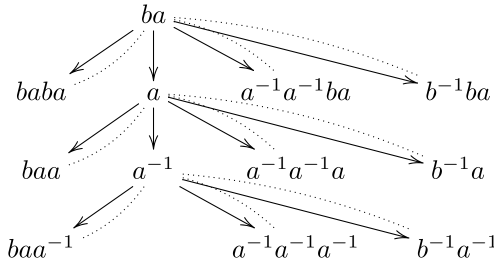
The dot arcs denote the embeddings over $\trianglelefteq$.

No path in this graph reaches Λ. *But such path in the whole tree exists: trace that corresponds to it is*

$$aBa \to CaBa \to aCaBa \to cBa \to Ba \to a \to Ca \to aCa \to c \to \Lambda.$$

This path corresponds to the attack on the protocol P_{RR}.

Example 8. *Let us unfold all traces of* $\mathbf{G_{PA0}}$ *(Example 6) until a pair* Γ, Δ, *such that* Δ *is a descendant of* Γ *and* $\Gamma \trianglelefteq \Delta$, *emerges on a branch.*



All the paths end with Higman pairs and the branch which ends with Λ *is lost. Note that if the rule* $aa^{-1} \to \Lambda$ *is allowed then such branch (but not all possible such branches) is found even by the unfolding with the Higman condition.*

Therefore Higman relation itself is not enough as a termination condition while working with prefix rewriting grammars that are generated from protocols. In the next section we consider some other classical termination condition that reveals some interesting properties of the protocol verification task.

4 Time Indexing and Turchin Relation

Consider a trace $\{\Phi_i\}_{i=1}^n$ of a prefix rewriting grammar $\mathbf{G} = \langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$. Let us write it down letter by letter, from the rightmost letter of a word to its leftmost letter. Let us mark every letter by a natural number denoting the moment when the letter appeared first (starting from the moment 0). We call this notation *time-indexing* and a trace generated by $\mathbf{G}$ with the time-indexing notation is called *a computation*.

This procedure can be described more formally as follows. The i-th letter of Φ_1 is marked by $|\Gamma_0| - i$, where $|\Gamma_0|$ is the length of Γ_0; if the largest number that is used as a time index in the track segment $\{\Phi_i\}_{i=1}^k$ $(k < n)$ is M and Φ_{k+1} is generated from Φ_k by the rule $R : R_l \to R_r$ then the i-th letter of Φ_{k+1} $(i \leq |R_r|$ where $|R_r|$ is the length of $R_r)$ is marked by the time index $M + |R_r| - i + 1$. Time indexes of all other letters in Φ_{k+1} remain the same as in Φ_k, since these letters are unchanged by R.

The length of a word Δ is denoted as $|\Delta|$. $\Delta[k]$ denotes the k-th letter of Δ; $\Phi \approx \Psi$ iff the words Φ and Ψ coincide up to time indices. The time-indexing notation allows take into account not only structure of a word but also the history. The time indices are useful to describe Turchin's relation on words in a trace — this relation is more powerful than Higman relation (it permits later termination of branches).

Example 9. *Consider the protocol* $\mathbf{G_{RR}}$ *from the example 7. The first word* Γ_0 *is* aBa *and should be annotated by the time indices as* $a_{(2)}B_{(1)}a_{(0)}$ *(we write the indices in the subscripts, enclosed in brackets). When it is transformed by the rule* R_4*, the generated word with the time indices is* $b_{(3)}$*. When it is transformed then by* R_1*, the generated word is time-indexed as* $a_{(4)}b_{(3)}$*.*

Definition 5. *Two words* Γ *and* Δ *form a Turchin pair iff* $\Gamma = \Phi\Theta_0$*,* $\Delta = \Phi'\Psi\Theta_0$ *and* $\Phi' \approx \Phi$*. This fact is denoted as* $\Gamma \preceq \Delta$*.*

Now the first word must not only be a subsequence of the second but also this subsequence must contain only one gap and have the special properties of the time indices. For example if the word $\Gamma = a_{(i)}$ is transformed to $\Delta = b_{(i+2)}a_{(i+1)}$ (after an application of the rule $a \to ba$), then the pair (Γ, Δ) does not satisfy Turchin relation, but it satisfies Higman one.

In 1987 V.F. Turchin proved that the Turchin pairs necessarily appear in every infinite trace generated by an arbitrary finite type 2 prefix rewriting grammar[4]. Thus the Turchin relation can be used (and is successfully used [6]) to terminate unfolding of computations in a computational tree. Our next task is to investigate for what types of prefix grammars the relation $\preceq$ can help to find at least one trace that ends with Λ and how to refine the relation to make it applicable to solve this task for any 0-type prefix grammar.

5 Verification of Ping-Pong Protocols as Prefix Grammars

Now we introduce the notion of annotated prefix rewriting grammars, which was described in [7] as a way of removing all occasional Turchin pairs in traces generated by the 2-type prefix rewriting grammars.

Example 10. *Consider the following modification of the grammar* $\mathbf{G_{PA0}}$*.*

$\mathbf{G_{PAC}}$*:*

$R_1 : \Lambda \to ba$	$R_4 : b \to \Lambda$	$R_7 : \Lambda \to bac$
$R_2 : \Lambda \to a^{-1}a^{-1}$	$R_5 : a \to a^{-1}$	
$R_3 : \Lambda \to b^{-1}$	$R_6 : aa \to \Lambda$	

The two rules R_1 *and* R_7 *are different in essence: an application of the second leads to impossibility of* Λ *in the further trace. But if two words* Φ *and its descendant* Ψ *form a Turchin pair, and* Ψ *is generated directly by* R_7 *from* Ψ' *then the word generated from* Ψ' *by* R_1 *also forms a Turchin pair with* Φ*. Thus* $\preceq$ *does not separate applications of these two rules because of the same prefix* ba*.*

To avoid these useless Turchin pairs we can mark the letters of the right-hand side of a rule by the number of the rule in the whole list of rules and then forbid unification of the letters with different rule numbers. Such marked grammars are called annotated.

Definition 6. *A prefix rewriting grammar* $\mathbf{G}$ *is annotated if every two rules either have the same right-hand side or have no letters shared by their right-hand sides.*

Example 11. *Let us annotate the grammar* $\mathbf{G_{RR}}$ *(considering* Γ_0 *as* R_0*).*

[4] A formal representation of the theorem in the terms of prefix grammars and a short proof of it can be found in [7].

$\mathbf{G_{ARR}}$:

$R_1 : \Lambda \to a^{(2)}$	$R_4 : aBa \to b^{(1)}$	$R_7 : B \to \Lambda$
$R_2 : \Lambda \to c^{(1)}$	$R_5 : aCa \to c^{(2)}$	$R_8 : C \to \Lambda$
$R_3 : \Lambda \to C^{(1)}$	$R_6 : c \to \Lambda$	$R_0 : \Lambda \to a^{(1)}B^{(0)}a^{(0)}$

The attack now looks as

$$a^{(1)}B^{(0)}a^{(0)} \to C^{(1)}a^{(1)}B^{(0)}a^{(0)} \to a^{(2)}C^{(1)}a^{(1)}B^{(0)}a^{(0)} \to c^{(2)}B^{(0)}a^{(0)} \to$$
$$\to B^{(0)}a^{(0)} \to a^{(0)} \to C^{(1)}a^{(0)} \to a^{(2)}C^{(1)}a^{(0)} \to c^{(2)} \to \Lambda.$$

We drop time indices for the sake of brevity. No pairs in this trace are comparable by $\preceq$ so the corresponding path is to be found during the unfolding.

The transition to annotated grammars leads to a refinement of $\preceq$ that allows to solve the empty word problem for the type 2 prefix rewriting grammars.

Proposition 1. *Let us consider a relation $\preceq^T$ such that $\Gamma \preceq^T \Delta$ iff $\Gamma = \Phi\Theta_0$, $\Delta = \Phi'\Psi\Theta_0$, and there exists such rule $R : R_l \to R_r$ with a non-empty right-hand side R_r that $\Phi \approx R_r$ and $\Phi' \approx R_r$. Every infinite computation generated by a 0-type prefix grammar contains an infinite subsequence $\Gamma_1, \dots, \Gamma_n, \dots$ such that for all n and i $\Gamma_n \preceq^T \Gamma_{n+i}$) and Γ_{n+i} is a descendant of Γ_n.*

This proposition can be reformulated in the following way. Let us say that an application of a rule $R : R_l \to R_r$ to $R_l\Theta_0$ is cancelled in a trace iff $\Theta_0[1]$ is modified or erased in the trace. Then every infinite computation by a 0-type prefix grammar contains infinite subsequence $\Gamma_1, \dots, \Gamma_n, \dots$, such that for all n and i Γ_n and Γ_{n+i} are generated by the same rule $R : R_l \to R_r$ with the non-empty right-hand side R_r and the application of the R in Γ_n is not cancelled in Γ_{n+i}.

Note that the usage of $\preceq^T$ with the non-annotated grammar $\mathbf{G_{RR}}$ allows to find the trace ending by Λ as well as the usage of $\preceq$ with the annotated grammar $\mathbf{G_{ARR}}$. Moreover, the transition to the annotated grammars (or to $\preceq^T$ instead of $\preceq$) solves the empty word problem for traces generated by 2-type prefix rewriting grammars.

Proposition 2. *Let* $\mathbf{G}$ *be an annotated the 2-type prefix rewriting grammar. If there is a computation by* $\mathbf{G}$ *ending with Λ then there is a computation by* $\mathbf{G}$ *ending with Λ and containing no Turchin pairs.*

Proof. The proof follows from the fact that every first Turchin pair in a computation by $\mathbf{G}$ satisfies also $\preceq^T$ condition. Consider the shortest computation to Λ that is generated by an annotated alphabetic prefix rewriting grammar. Suppose it contains a Turchin pair $\Gamma = R_r\Theta_0$, $\Delta = R'_r\Psi\Theta_0$. If Δ collapses to Λ then the trace from Δ to Λ must contain words $\Psi\Theta_0$ and Θ_0. But this means

that there exists a way to collapse R_r to Λ and Θ_0 to Λ separately, and there is a shorter computation to Λ from Γ without reaching Δ.

□

But the strong relation $\preceq^T$ does not still solve the whole problem of finding a finite trace when verifying protocols as prefix rewriting grammars.

Example 12. *Let us try the time-indexing notation on the trace representing the attack on the protocol P_a.*

$$\Gamma_0 : b_{(1)}a_{(0)} \xrightarrow{R_4} \Gamma_1 : a_{(0)} \xrightarrow{R_1} \Gamma_2 : b_{(3)}a_{(2)}a_{(0)} \xrightarrow{R_4} \Gamma_3 : a_{(2)}a_{(0)} \xrightarrow{R_6} \Gamma_4 : \Lambda$$

The new a in Γ_3 is marked by the index (2) because (1) is used with the first b which collapses in Γ_1.

Even using the most powerful relation $\preceq^T$ in the example we cannot avoid the embedding $b_{(1)}a_{(0)} \preceq^T b_{(3)}a_{(2)}a_{(0)}$ which leads to the loss of attack. The bottleneck is that $\mathbf{G_{PA0}}$ is the 0-type prefix rewriting grammar. And the annotation of a grammar helps to express (and eliminate) dependencies only between the right-hand sides of rules. If a grammar is the 2-type then all the rules with the empty right-hand sides look as $x \to \Lambda$ and thus every letter can be erased independently. But if we have any dependence between the erasings, we cannot express or eliminate it by the simple annotating. Exactly this happens in $\mathbf{G_{PA0}}$: the rule $aa \to \Lambda$ assumes that we erase the first and the second a only together, but both of them can be generated only by the same rule $\Lambda \to ba$, which produces the essential Turchin pair.

Definition 7. *Let x be a letter that appears in the right-hand side of a rule of a given grammar. Its erasing counter is the number of the occurrences of x in the different left-hand sides.*

Thus, for a in $\mathbf{G_{PA0}}$ the erasing counter is 3, and for b is 1.

Example 13. *Consider the grammar $\mathbf{G_{ARR}}$ from Example 11. $c^{(i)}$ is to be erased by the single rule R_6 and has the erasing counter 1, $a^{(i)}$ can be erased either by R_4 or R_5 and has the erasing counter 4, since in the both rules there are two occurrences of a in the left-hand sides.*

Every letter in a 2-type prefix grammar has the erasing counter either 0 (if the letter cannot be erased, as c in $\mathbf{G_{PAC}}$ from Example 10) or 1.

Definition 8. *Let us say that* $\Gamma \preceq^! \Delta$ *(or forms with* Δ *a Turchin pair with erasing distinction) iff* $\Gamma \preceq^T \Delta$ *and for some* $x \in \Sigma$ *the number of the occurrences of* x *in* Δ *is greater than the erasing counter of* x*.*

Proposition 3. *Let* $\mathbf{G}$ *be a finite annotated prefix rewriting grammar. Every infinite trace generated by* $\mathbf{G}$ *either contains some* Γ *and* Δ*, such that* $\Gamma \approx \Delta$*, or contains* Γ *and* Δ*, such that* $\Gamma \preceq^! \Delta$*.*

Proof. Since there is an infinite sequence $\Gamma_1, \dots, \Gamma_n, \dots$, such that for all i, n $\Gamma_n \preceq^T \Gamma_{i+n}$, there exists a rule $R : R_l \to R_r$ with $R_r \neq \Lambda$, that is applied infinite many times to generate all of the Γ_i. Then the two situations can appear. Either there is an infinite number of Γ_i such that they all coincide up to the time indices, or there is a strictly growing (by the length) infinite subsequence $\{\Gamma_{n_1}\}$. Since the alphabet is finite, there exists such element Γ_{j_1} from this subsequence. that some letter from Σ repeats itself in Γ_{j_1} more times than its erasing counter is. The pair Γ_1 and Γ_{j_1} satisfies the $\preceq^!$ relation.

□

Proposition 4. *Let* $\mathbf{G}$ *be an arbitrary finite annotated prefix rewriting grammar of the type* 0*. If there is a computation by* $\mathbf{G}$ *that ends with* Λ *then there is a computation by* $\mathbf{G}$ *that ends with* Λ *and contains no Turchin pairs with erasing distinction.*

Proof. Let the shortest computation of Λ contain a Turchin pair with erasing distinction, so there are such Γ and Δ that $\Gamma \preceq^! \Delta$. Let a be a letter that is to be erased twice by the same rule. Consider the immediate moments before the erasings. They must look like $\hat{R}_l a \Psi a \Theta_0$ and $\hat{R}_l{}' a \Theta_0$, where $\hat{R}_l$ denotes a prefix of the left-hand side of some rule. Now consider the moments in which a were generated. They must look like $\hat{R}'_r a \Theta_0$ and $\hat{R}'_r a \Psi a \Theta_0$ respectively and form the pair in respect of $\preceq^!$.

All the considered moments together form the following sequence.

...
$\hat{R}'_r a \Theta_0$
...
$\hat{R}'_r{}' a \Psi a \Theta_0$
...
$\hat{R}_l a \Psi a \Theta_0$
...
$\hat{R}_l{}' a \Theta_0$
...
Λ

Now we can apply to $\hat{R}'_r a \Theta_0$ the same transformations as in the segment from $\hat{R}'_r{}' a \Psi a \Theta_0$ to $\hat{R}_l a \Psi \Theta_0$ and get a shorter computation to Λ. □

Proposition 4 together with the proposition 3 gives a sound condition of when to terminate a computation if we want to find the shortest finite computation path. If some two words that are comparable over $\preceq^!$ or over $\approx$ appear, then we can stop unfolding the computation that contains them. All infinite computations contain such pairs, so we cannot diverge when unfolding until them.

The proposition 4 implies an obvious observation that if there a rule with a letter in the right-hand side that have the erasing counter 0 is applied in a computation then the computation contains no Λ. Moreover, the proposition 2 is a straight consequence of the lemma 4 if none letters in rules with non-empty right-hand sides have erasing counters greater than 1.

6 Modelling Long Attacks via Refined Turchin Relation

Now when we know the test of branch termination for finding attacks, the question of maximal attack length arises. The work [7] describes how to build maximal trace with no Turchin pairs in it. But this trace is not necessarily a minimal trace ending with Λ.

Example 14. *Consider the annotated grammar from [7]. Remind that R_0 coincides with the initial word Γ_0. x denotes an arbitrary letter from $\Sigma = \{a, b, c, d, e, f, g\}$.*

$\mathbf{G'_F}$:

$R_0 : x \to ab$	$R_2 : x \to ef$
$R_1 : x \to cd$	$R_3 : x \to g$
$R_4 : x \to \Lambda$	

The longest trace of this grammar without the Turchin pairs has the length 22. However, the shortest attack has the length 3: it is enough to apply rule R_4 twice to the initial $a_{(1)}b_{(0)}$ to receive Λ.

Now our task is not only to build the longest possible trace with no Turchin pairs but to build a grammar with a long shortest possible trace ending by Λ (in particular, this means it contains no pairs in respect of $\preceq^!$ but very likely contains some pairs in respect of $\preceq^T$). We use the same idea of the "ladder" construction as in [7] to show that it is possible to construct a grammar with a minimal trace ending by Λ such that it repeats the "ladder" constructions as well as contains pairs over $\preceq^T$.

Example 15. *In the following grammar* $\mathbf{G_{EXP}}$ *the shortest trace ending by* Λ *has the length* 80 *(see Appendix).*

$\mathbf{G_{EXP}}$*:*

$R_1 : \Lambda \to aA$	$R_4 : Ba \to bB$	$R_7 : BB \to cC$
$R_2 : \Lambda \to bB$	$R_5 : AA \to bB$	$R_8 : c \to \Lambda$
$R_3 : \Lambda \to cC$	$R_6 : Cb \to cC$	$R_9 : CC \to \Lambda$

7 Conclusion

We described a way to verify some class of ping-pong protocols via unfolding techniques and prefix rewriting grammars. Now we considered not only the 2-type rewriting grammars but grammars of the type 0 of Caucal hierarchy. We showed that the condition of path termination with respect of $\preceq^!$ cannot be replaced by $\preceq^T$ or weaker conditions (such as Higman relation) and how a grammar with a small number of rules can generate a long minimal finite computation. Thus the question of how to verify systems described by the 0-type prefix rewriting grammars via computational tree unfolding is answered in general.

This shows that the field of applications of the Turchin relation is much wider than function stack embeddings, as in the original work. Annotation and erasing distinction can be a reliable method of avoiding too early termination of unfolding, but for many special classes of systems it would be more efficient to use not $\preceq^!$ itself but its generalizations or restrictions that are more relevant to the class.

References

[1] M. Abadi and A.D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998.

[2] A.C. Yao D. Dolev. On the security of public key protocols. *Transactions on Information Theory*, 29:198–208, 1983.

[3] S. Even D. Dolev and R.M. Karp. On the security of ping-pong protocols. *Information and Control*, 55:57–68, 1982.

[4] G. Delzanno, J. Esparza, and J. Srba. *Monotonic Set-Extended Prefix Rewriting and Verification of Recursive Ping-Pong Protocols*, volume 4218 of *Lecture Notes in Computer Science*, pages 415–429. IEEE Computer Society Press, 2006.

[5] P. Jancar and J. Srba. *Undecidability Results for Bisimilarity on Prefix Rewrite Systems*, volume 3921 of *Lecture Notes in Computer Science*, pages 277–291. IEEE Computer Society Press, 2006.

[6] A. P. Nemytykh. *The Supercompiler Scp4: General Structure.* URSS, Moscow, 2004.

[7] A. Nepeivoda. A refinement of higman embedding for loop approximation. [unpublished], 2013. `http://refal.botik.ru/preprints/Antonina_Nepeivoda-On_Turchin_Theorem-06042013v1.pdf`.

[8] M. H. Sørensen and R. Gluck. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.

[9] M.H. Sørensen, R. Gluck, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:465–479, 1993.

[10] V.F. Turchin. The algorithm of generalization in the supercompiler. *Partial Evaluation and Mixed Computation*, pages 341–353, 1988.

A The Long Attack on $\mathbf{G_{EXP}}$

The long trace ending by Λ generated by $\mathbf{G_{EXP}}$ with the initial word aA.

1	aA	$cCBBA$	$cCbBAA$	bB
2	$bBaA$	$CBBA$	$CbBAA$	$cCbB$
3	$cCbBaA$	$cCCBBA$	$cCBAA$	CbB
4	$CbBaA$	$CCBBA$	$CBAA$	cCB
5	$cCBaA$	BBA	$cCCBAA$	CB
6	$CBaA$	cCA	$CCBAA$	$cCCB$
7	$cCCBaA$	CA	BAA	CCB
8	$CCBaA$	$cCCA$	$bBBAA$	B
9	BaA	CCA	$cCbBBAA$	bBB
10	bBA	A	$CbBBAA$	$cCbBB$
11	$cCbBA$	aAA	$cCBBAA$	$CbBB$
12	$CbBA$	$bBaAA$	$CBBAA$	$cCBB$
13	$cCBA$	$cCbBaAA$	$cCCBBAA$	CBB
14	CBA	$CbBaAA$	$CCBBAA$	$cCCBB$
15	$cCCBA$	$cCBaAA$	$BBAA$	$CCBB$
16	$CCBA$	$CBaAA$	$cCAA$	BB
17	BA	$cCCBaAA$	CAA	cC
18	$bBBA$	$CCBaAA$	$cCCAA$	C
19	$cCbBBA$	$BaAA$	$CCAA$	cCC
20	$CbBBA$	$bBAA$	AA	CC

Program Transformation for Program Verification

Alberto Pettorossi[1] and Maurizio Proietti[2]

[1] DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[2] IASI-CNR, Rome, Italy maurizio.proietti@iasi.cnr.it

We present a transformational approach to program verification and software model checking that uses three main ingredients: (i) Constraint Logic Programming (CLP), (ii) metaprogramming and program specialization, and (iii) proof by transformation. (i) *Constraints* are used for representing in a compact way (finite or infinite) sets of values or memory states, and *logic* is used for expressing properties of program executions [2, 4, 5]. The least fixpoint semantics and negation allow us to denote both the least models and the greatest models of programs, and thus to reason about the (finite or infinite) behaviour of programs. (ii) *Metaprogramming* is used for getting a verification technique which is parametric with respect to the programming language in use. In particular, we introduce a CLP program I which defines the (meta)interpreter of the programming language in which the program P to be verified is written. Then, in order to gain efficiency, we remove this interpretation layer by *specializing* the interpreter I with respect to the given program P [1, 6, 7]. The property φ that should be proved (or disproved) about program P, is expressed through the CLP clauses that characterize the set of states in which φ holds (or does not hold, respectively). (iii) Having derived a CLP program $\widetilde{P}$ whose semantics represents the behaviour of the given program P and the property φ to be verified, we start a third phase which consists in the *proof by CLP program transformation.* This transformation is performed by using *unfold/fold rules* and also some generalization and goal replacement rules which all preserve the semantics [8]. By the generalization rule [3] one can derive the invariants which hold during program execution and are needed to verify the given property. Rules are applied according to some *strategies* with the objective of deriving from program $\widetilde{P}$ a new CLP program $\widetilde{P1}$ so that a selected atom, say *prop*, either belongs to $\widetilde{P1}$ (in which case φ holds) or no clause for *prop* belongs to $\widetilde{P1}$ (in which case φ does not hold). We have designed a few (semi)automatic strategies which make the transformation process to terminate. Obviously, they are all incomplete due to undecidability limitations, but they work well on many non-trivial examples.

References

[1] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *Proc. ACM PEPM'13*, 43–52, New York, USA, 2013.

[2] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, ed., *TACAS'99*, LNCS 1579, 223–239. Springer-Verlag, 1999.

[3] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming.* 13(2):175–199, 2013.

[4] L. Fribourg. Constraint logic programming applied to model checking. In A. Bossi, ed., *Proc. LOPSTR'99*, LNCS 1817, 31–42. Springer-Verlag, 2000.

[5] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In C. Flanagan and B. König, eds., *Proc. TACAS'12*, LNCS 7214, 549–551. Springer, 2012.

[6] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. *Proc. LOPSTR'99*, LNCS 1817, 63–82. Springer, 2000.

[7] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, ed., *Proc. SAS'98*, LNCS 1503, 246–261. Springer, 1998.

[8] A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, ed., *Proc. CL 2000*, Lecture Notes in Artificial Intelligence 1861, 613–628. Springer, 2000.

Building Trustworthy Refactoring Tools

Simon Thompson

School of Computing, University of Kent,
Canterbury, Kent, CT2 7NF, UK
s.j.thompson@kent.ac.uk

The bar for adoption of refactoring tools is high: not only does a refactoring extract information from your source code, it also transforms it, often in a radical way.

After discussing what users require from their tools, we will examine ways in which tool builders can try to increase their users' confidence in the tools. These mechanisms include visualisation, unit testing, property-based testing and verification, and are based on the Kent functional programming group's experience of building the HaRe and Wrangler refactoring systems for Haskell and Erlang.

Научное издание

Труды конференции

Сборник трудов Первого международного семинара по верификации и преобразованиям программ, г. Санкт-Петербург, 13-14 июля 2013 г.

Под редакцией А. П. Лисицы и А. П. Немытых.

Для научных работников, аспирантов и студентов.

Издательство «**Университет города Переславля**»,
152020 г. Переславль-Залесский, ул. Советская 2.

Гарнитура **URW Helvetica**. Формат **60×84/16**.
Дизайн обложки: *Н. А. Федотова*. Уч. изд. л. **11.12**.
Усл. печ. л. **11.53**. Подписано к печати **17.06.2013**.
Ответственная за выпуск: *Н. А. Федотова*.

Отпечатано в **ООО "Регион"**.
Печать **офсетная**. Бумага **офсетная**. Тираж **150 экз**. Заказ 12.
152025 Ярославская область г. Переславль-Залесский ул. Строителей, д. 41