Extending Non-Termination Proof Techniques to Asynchronously Communicating Concurrent Programs

Matthias Kuntz
University of Konstanz
Konstanz, Germany
matthias.kuntz@uni-konstanz.de
, Stefan Leue
University of Konstanz
Konstanz, Germany
stefan.leue@uni-konstanz.de
and
Christoph Scheben
University of Konstanz
Konstanz, Germany
christoph.scheben@uni-konstanz.de

Abstract

Currently, no approaches are known that allow for non-termination proofs of concurrent programs which account for asynchronous communication via FIFO message queues. Those programs may be written in high-level languages such as Java or Promela. We present a first approach to prove non-termination for such programs. In addition to integers, the programs that we consider may contain queues as data structures. We present a representation of queues and the operations on them in the domain of integers, and generate invariants that help us prove non-termination of selected control flow loops using a theorem proving approach. We illustrate this approach by applying a prototype tool implementation to a number of case studies.

1 Introduction

The non-termination of program loops is an interesting property of a program. For instance, tests for buffer-unboundedness of concurrent code, such as the one suggested in [10], rely on the presence of non-terminating program loops. Non termination is a necessary but not sufficient condition for buffer-unboundedness: a system contains an unbounded buffer b iff there exists *no* constant c such that, for each reachable state S, the length of the content of the buffer b in state S is less than c. This implies in particular that the state space of a program p with an unbounded buffer has to be infinite. The application of König's lemma, which asserts that every infinite tree of finite degree has an infinite branch, on the reachability tree of p implies that p has a non-terminating execution. Hence we are confronted with the problem of proving non-termination for high level programming and modeling languages.

It is our ultimate goal to adopt the test described in [10] to more expressive concurrent code. As a sub-goal, we describe in this paper a non-termination verification methodology for concurrent programs that contain message passing via unbounded FIFO queues. In addition to queues, the programs that we consider may contain integers as data structures. It is our assumption that the methodology we present in this paper can easily be adopted to other high-level data structures.

Asynchronous concurrent programming is possible via various programming languages, such as for instance Java. Java does not have syntactically built-in communication buffers, but the language comes along with libraries that allow for programming FIFO queue based communication. While it will ultimately be interesting to apply our verification methodology to a programming language such as Java, we currently apply our analysis only to concurrent code specified in Promela, the input language of the SPIN model checker [9]. This choice is motivated by convenience – the SPIN environment offers easy access to the syntactic structure of the language, SPIN can easily be modified to allow for the type of concolic execution [7, 12] that we use, and there are lots of concurrent Promela models available in the public domain. We maintain that Promela encompasses the central communication features offered by any higher-level concurrent programming language, and hence assume that our conceptual results are easily transferable to those languages. We also wish to point out that we do not intend to compete in any way with the finite state verification engine of SPIN, which would be entirely unsuitable to verify the type of properties we aim at. Finally, we do not assume the message buffers to have finite capacity, even though we need to fill in dummy capacity values to satisfy SPIN's Promela parsing requirements.

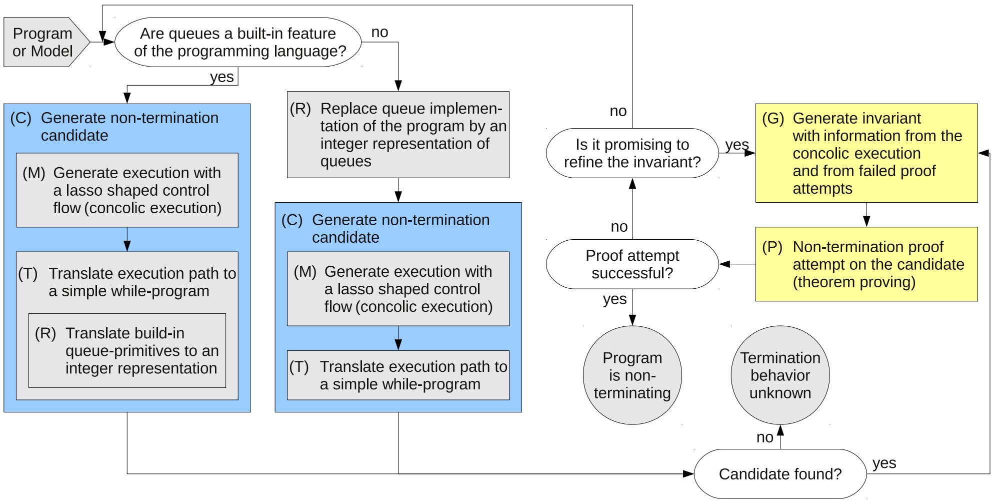
Program or Model
Are queues a built-in feature of the programming language?
no
yes
(C) Generate non-termination candidate
(M) Generate execution with a lasso shaped control flow (concolic execution)
(T) Translate execution path to a simple while-program
(R) Translate build-in queue-primitives to an integer representation
(R) Replace queue implementation of the program by an integer representation of queues
(C) Generate non-termination candidate
(M) Generate execution with a lasso shaped control flow (concolic execution)
(T) Translate execution path to a simple while-program
no
Is it promising to refine the invariant?
yes
(G) Generate invariant with information from the concolic execution and from failed proof attempts
no
Proof attempt successful?
(P) Non-termination proof attempt on the candidate (theorem proving)
yes
Program is non-terminating
Termination behavior unknown
no
Candidate found?
yes

Figure 1: The non-termination proof approach.

The Approach. The general structure of our approach, as illustrated in Figure 1, comprises the following main stages:

- During the first stage (marked C) we need to find looping executions of the program that can at all lead to non-terminating program behavior. We do so by determining lasso shaped executions in the global control flow graph with the help of concolic execution.
- During the second stage (marked G) we generate an invariant for any non-termination candidate that we determined during stage C using information collected during the concolic execution.
- In the course of the third stage (marked P) we use a theorem prover in order to attempt to show non-termination for the candidate that we determined before. In this proof we will use the invariant generated during stage G.

These three stages are embedded in two cycles: If a proof attempt fails, then the invariant is refined using information from the failed proof attempt. Afterwards, a new proof attempt is started. If the proof still fails after several refinement steps we assume that the candidate is spurious. In this case the concolic execution is resumed to generate another non-termination candidate. Now, the same procedure is applied to this candidate. The approach comes to an end if either a proof attempt succeeds, or if the concolic execution does not deliver any more candidates. In the latter case the termination behavior of the program remains unknown.

In order to use state of the art non-termination proof techniques as a basis of our approach we replace the queues of the program by an integer representation of queues (part R). The advantage of this representation is that state of the art theorem provers and constraint solvers provide powerful and sophisticated techniques to deal with integers. A translation to an integer representation promises the possibility to take advantage of the reasoning power of these tools. Depending on whether the language under consideration provides queues as first class entities or not, the replacement of the queues has to be done before or as part of the non-termination candidate generation. Notice that in our particular setting, queues are first class entities within the Promela language. The non-termination candidate generation itself consists of two parts: In the first part (marked M) we use concolic execution in order to generate executions with a lasso shaped control flow. Those executions could potentially lead to a non-terminating execution. In part two of the candidate generation the executions are rewritten to a simple while-program. This while-program has the property that its non-termination implies the non-termination of the original program.

We have implemented our approach by modifying the model checker SPIN [9] to deliver lasso-shaped non-termination candidates. This modification allows us to obtain lassos from SPIN without having to specify a never claim or some form of liveness property to be checked. Furthermore, we use the all purpose verification system KeY [1] to perform the non-termination proof attempts on the candidates.

Related Work. Though there has been a lot of research on non-termination proof techniques for logic programs and term rewriting systems [6], to the best of our knowledge there exist only three approaches which focus on proving non-termination *for high level languages*.

In [8] an approach to prove non-termination of `C`-programs by calculating recurrent sets with the help of concolic execution is presented. The approach uses a non-termination candidate generation similar to the one in part M of our approach. However, instead of translating the execution to a while-program as we do in part T, the executions are transformed to a set of linear inequations which is solved by a constraint solver. A drawback of [8] is that their approach can generate recurrent sets only for variables of integer-like types. Moreover, this approach is able to handle only loops with linear updates to variables.

Another approach [13] for non-termination proofs for Java programs is based on the KeY system. KeY is able to execute arbitrary Java Card programs (as well as a subset of Java SE programs) symbolically with the help of a multi modal logic, called Java Dynamic Logic (Java-DL). The approach in [13] is restricted to a subset of Java-DL, called While-DL. Non-termination is expressed in terms of a While-DL formula. Proofs of the correctness of this formula are attempted with the help of loop-invariants that aim to ensure non-termination. The loop invariants are generated and refined by an invariant generator which uses information from failed proofs to refine previously generated invariants. A drawback of this approach is that it is currently not applicable to concurrent programs, and that it has not yet been applied to other data types than integers. Nevertheless, we are using this approach as a starting point for the parts G and P of our approach.

The last approach [6] proposes a non-termination proof technique for Java Bytecode. This approach approximates the Bytecode and compiles it into a constraint logic program with linear constraints. In case the approximation is exact, non-termination of the constraint logic program implies non-termination of the Bytecode. In case exact approximations of data structures are available, those may be handled as well. However, this approach suffers from the same shortcoming as [8]: Since the approximation of non-linear constraints with the help of a linear constraint system is not exact, both approaches can handle only programs with linear constraints.

2 Foundations

2.1 Promela / SPIN

Promela is a modeling language for concurrent systems and as such the input language of the explicit state model checker *SPIN* [9]. It has been successfully used for the modeling and analysis of many concurrent systems [11, 4]. Promela is based on a subset of the programming language `C`, and adds guarded commands as well as concurrent processes, variables, arrays, structs and different types of communication to the language. The communication primitives used in Promela include asynchronous communication, synchronous rendez-vous communication, and synchronization via shared variables.

```
mtype = {a,b,c,d};

chan f12 = [100] of {mtype};
chan f21 = [100] of {mtype};

active proctype P2(){
  do
  :: f12?d -> skip;
  :: f21!a; f12?c -> f21!b;
  :: f21!c; f12?a -> f21!a;
  :: f21!b; f12?b -> f21!c;
  od;
}
```

```
active proctype P1(){
  int i = 0;
  int j = -3;
  do
  :: i >= 0 -> i++; f12!a; f21?b -> f12
      !d;
  :: j < -2 -> j--; f12!b; f21?c -> f12
      !d;
  :: i - j >= 1 -> f12!c; f21?a -> f12!
      d;
  od;
}
```

Figure 2: An example model in Promela.

In the following we will go through parts of the language with the help of a simple example model, given in Figure 2. This example will be used as a running example in the paper in order to explain our approach. The model consists of two concurrent processes, one of the process type `P1` and one of process type `P2`. Both types are instantiated once, which is expressed by the keyword `active`. `P1` and `P2` exchange messages via communication buffers: `P1` sends messages to `P2` via buffer `f12` while `P2` sends messages to `P1` via buffer `f21`. The types of exchanged messages are defined as elements of the special enumeration type `mtype`. Each client performs a loop where it chooses non-deterministically between three respectively four possible execution blocks. Each possibility can be executed only if the first statement of the block is executable. A statement is executable if it evaluates to true. In contrast to Boolean expressions like `i >= 0` and receive statements like `f21?c` assignments like `i = 0;` are always executable.

The operational semantics of Promela can be defined with the help of transition systems with FIFO channels (TSFCs) [10] extended by data types ranging over finite domains, and depth-bounded process recursion. TSFCs are a generalization of Communicating finite state machines (CFSMs) [14, 2]. CFSMs are often used for the specification of asynchronous concurrent systems like communication protocols [14, 2]. Informally, a system of CFSMs is a set of concurrent

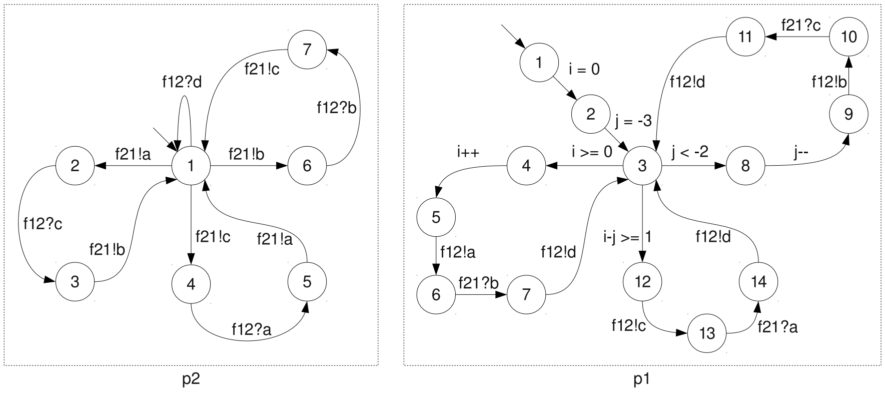

Figure 3: The system of extended CFSMs for the model of Figure 2.

processes. Each of the processes in a CFSM system is a finite state systems that communicates via the exchange of messages through First-In-First-Out (FIFO) buffers with other processes in the system. The FIFO buffers have unbounded capacity. A CFSM system is hence an infinite state system, and many interesting properties for this class of systems are undecidable [2]. Figure 3 shows an example of a system of CFSMs extended by data types ranging over finite domains. In a system of CFSMs, there is a buffer $f_{i\rightarrow j}$ for each pair of processes (i, j). $f_{i\rightarrow j}$ can only be written by process i, whereas $f_{i\rightarrow j}$ can only be read by process j. In contrast to CFSMs, in a TSFC, every process is allowed to access every buffer within the system. The semantics of TSFCs can be given as reachability trees. Every finite or infinite path in the reachability tree is an execution of the system.

As stated in full detail in the introduction, the choice of the Promela in the context of this paper is motivated by convenience – the SPIN environment offers easy access to the syntactic structure of the language, SPIN can easily be modified to allow for the type of concolic execution that we use, and there are lots of concurrent Promela models available in the public domain. It is important to note that we are ignoring the bounds on buffers in Promela and assume that buffers can hold an arbitrary number of messages. In other words, we do not take advantage of the fact that Promela models, when complying with runtime limitations imposed by the SPIN model checker, are finite state systems, but consider Promela models to be infinite state instead.

2.2 The ADT queue

We are using the axiomatic definition of the abstract data type (ADT) queue of [5] with the abstract algebra restricted to the operations *enqueue*, *dequeue*, *head* and *isEmpty*. However, we abbreviate the terms

$$\mathit{enqueue}(x_0, \mathit{enqueue}(x_1, \ldots \mathit{enqueue}(x_n, \mathit{emptyQueue}) \ldots))$$

by words $x_n \ldots x_1 x_0 \in \Sigma^*$ to provide better readability.

3 Proving non-termination of asynchronously communicating concurrent programs

As sketched in the introduction, our approach consists of four phases: the replacement of the queue implementation (marked R), the generation of non-termination candidates (marked C), the invariant generation (marked G) and the proof attempt on the candidates (marked P). These parts will be discussed in more detail in the following sections in their chronological order.

3.1 Integrating queues in current non-termination-proof techniques (phase R)

In order to reason over queues we use integers to represent the queues as well as the operations on them. There are several reasons motivating this approach:

- First, the integer representation enables a simple extension of state of the art non-termination proof techniques like [8] and [13]—which mainly can handle / generate invariants for integers—with the ability to reason over arbitrary data types, including queues.
- Second, state of the art theorem provers and constraint solvers provide powerful and sophisticated techniques to deal with integers. Thus a translation to an integer representation promises the possibility to take advantage of the reasoning power of these tools.
- Finally, as we show later, the invariants which we will derive for buffers in their integer representation will not contain explicit quantification. This makes it easier to reason over the thus obtained invariants.

The next section will introduce the integer representation on queues which we use in our approach. Afterwards, it will be discussed how the replacement / translation of queues in phase R can be performed.

3.1.1 Integer representation of queues

If one enumerates all elements of the alphabet Σ of a queue q from 1 to $|\Sigma|$, the content of q can be thought of as a natural number encoded in a positional numeral system with radix $r := |\Sigma|+1$. If, for instance, queue q has the alphabet $\Sigma = \{a, b, c, d\}$ and content $x = \boxed{a}\boxed{b}\boxed{a}\boxed{d}\boxed{d}$, then we can translate the buffer content to $\tau(x) = \boxed{1}\boxed{2}\boxed{1}\boxed{4}\boxed{4}$ by assigning message a the value 1, b the value 2, c the value 3 and d the value 4. Written in a positional numeral system with radix 5 we have $\tau(x) = 12144_5$ and thus we can assign to the buffer content $x = \boxed{a}\boxed{b}\boxed{a}\boxed{d}\boxed{d}$ the integer $\tau(x) = 12144_5 = 1 \cdot 5^4 + 2 \cdot 5^3 + 1 \cdot 5^2 + 4 \cdot 5^1 + 4 \cdot 5^0 = 924_{10}$. The number 0 is assigned to the empty queue. The enumeration of Σ therefore starts with 1 instead of 0. Thus, if one chooses an arbitrary bijective function $\sigma : \Sigma \to \mathbb{Z}_{|\Sigma|}$ one can define an injective function τ' which assigns every queue content $x \in \Sigma^*$ a natural number $\tau'(x) \in \mathbb{N}$ in the following way:

$$\tau' : \begin{cases} \Sigma^* \to \mathbb{N} \\ x \mapsto \begin{cases} 0 & \text{if } x = \epsilon \\ \sum_{i=0}^{n} (\sigma(x_i) + 1) \cdot r^i & \text{if } x = x_n \dots x_0 \end{cases} \end{cases} \quad \text{with } r := |\Sigma| + 1.$$

In order to show the correctness of the integer representation defined by τ' we define an isomorphism between the ADT and the integer representation. The first step to do so is to define a bijection τ between Σ^* and $\mathit{image}(\tau')$:

$$\tau : \begin{cases} \Sigma^* \to \mathit{image}(\tau') \subset \mathbb{N} \\ x \mapsto \tau'(x). \end{cases}$$

τ defines an isomorphism if the operations *enqueue*, *dequeue*, *head* and *isEmpty* are defined on $\mathit{image}(\tau')$ as in the following paragraphs. Note, that we give the definition of the operations in the first line of the equations and prove the isomorphism property in the following ones.

enqueue: Adding a message $x_{-1} \in \Sigma$ to a queue with content $\tau(x) = \sum_{i=0}^{l} (\sigma(x_i) + 1) \cdot r^i$ in integer representation equals the calculation of

$$\begin{aligned} \mathit{enqueue}(\tau(x_{-1}), \tau(x)) : &= \tau(x) \cdot r + \tau(x_{-1}) \\ &= \sum_{i=0}^{l} (\sigma(x_i) + 1) \cdot r^i \cdot r + (\sigma(x_{-1}) + 1) \\ &= \sum_{i=0}^{l+1} (\sigma(x_{i-1}) + 1) \cdot r^i \\ &= \tau(\mathit{enqueue}(x_{-1}, x)) \ . \end{aligned}$$

If one adds, for instance, the message d to the queue *abad*, which results in the queue *abadd*, then this is equivalent to the calculation of $\tau(abad) \cdot 5 + 4 = 1214_5 \cdot 5 + 4 = 12140_5 + 4 = 12144_5 = \tau(abadd)$.

dequeue: Removing the head of the queue q with content $\tau(x) = \sum_{i=0}^{l} (\sigma(x_i) + 1) \cdot r^i$ is equivalent to the calculation of

$$\begin{aligned} \mathit{dequeue}(\tau(x)) : &= \tau(x) \bmod r^l \\ &= (\sum_{i=0}^{l} (\sigma(x_i) + 1) \cdot r^i) \bmod r^l \\ &= \Big(\big((\sigma(x_i) + 1) \cdot r^l \big) \bmod r^l + \big(\sum_{i=0}^{l-1} (\sigma(x_i) + 1) \cdot r^i \big) \bmod r^l \Big) \bmod r^l \\ &= \sum_{i=0}^{l-1} (\sigma(x_i) + 1) \cdot r^i \\ &= \tau(\mathit{dequeue}(x)) \ . \end{aligned}$$

If one removes, for instance, a message from the queue *abadd*, which results in the queue *badd*, then this is equivalent to the calculation of $\tau(abadd) \bmod 5^4 = 12144_5 \bmod 5^4 = 2144_5 = \tau(badd)$.

head: Reading the head x_l of the buffer x with $\tau(x) = \sum_{i=0}^{l}(\sigma(x_i) + 1) \cdot r^i$ equals the calculation of

$$\begin{aligned} \mathit{head}(\tau(x)) :&= \tau(x) \text{ div } r^l \\ &= (\sum_{i=0}^{l}(\sigma(x_i) + 1) \cdot r^i) \text{ div } r^l \\ &= \sigma(x_l) + 1 \\ &= \tau(\mathit{head}(x)) \ . \end{aligned}$$

For instance, if one wishes to retrieve the head of the queue *abadd*, which is the message a, then this is equivalent to the calculation of $\tau(abadd)$ div $5^4 = 12144_5$ div $5^4 = 1_5 = \tau(a)$.

isEmpty: The function *isEmpty* is defined in the following way:

$$\begin{aligned} \mathit{isEmpty}(\tau(x)) :&= \tau(x) \doteq 0 \\ &= x \doteq \mathit{emptyQueue} \\ &= \mathit{isEmpty}(x) \end{aligned}$$

If we enlarge τ to booleans by the identity function this is equivalent to:

$$= \tau(\mathit{isEmpty}(x))$$

Here we distinguish explicit between the equality predicate $\doteq$ and the equality $=$ symbol of the meta-language.

3.1.2 Using the integer representation

In order to use the integer representation of queues in our approach one has to replace the original implementation of a queue by its integer representation. This is done in phase R. To this end we have to distinguish the following two cases:

- Queues are *first class entities* of the language under consideration, such as it is the case in Promela. In this case the syntactic constructs of the language, such as for instance the message sending expression `q!a`, have to be translated to semantically identical statements expressed in the integer representation of queues. This is done as part of the translation of the execution path in phase T.
- Queues are *not first class entities*, but have to be implemented with the help of the language itself, such as it is the case in Java. In this case an approach would be to replace the original, for instance Java, implementation of a queue by an implementation of the queue in integer representation. Of course, one would have to take care that both implementations share the same interface and are semantically equivalent (which should be the case if the original implementation conforms to the ADT). In this second case it is easier to replace the implementation by the integer representation before the non-termination candidate generation because this relieves us from the necessity to search for and replace operations on queues by their integer representation within phase T.

Note, that in most cases it is not possible to use the ADT at this phase instead of the integer representation because most programming languages do not provide direct support for ADTs.

3.2 Generating non-termination candidates (phase C)

With the help of the interpretation of queues as integers it is in principle possible to extend the non-termination proof approach of [8] directly to programs with FIFO buffers. In this case the system of (linear) updates of [8] will become a system of non linear updates containing the operations on integer represented FIFO buffers. The problem with this approach is that the resulting set of equations contains powers, and that it is thus not linear any more. Though we had a look at any constraint solver we are aware of and which might have been able to solve the constraint system, in the end none of them seemed to be suitable for our needs. Note that the constraint system also contains existential and universal quantification. However, we will take advantage of the idea described in [8] of using concolic execution to generate executions with a lasso shaped control flow. The executions then are not translated to a set of inequalities as in [8], but will be translated to a simple while-program. If the while program does not terminate, then the original program won't terminate either. This while-program is what we call the non-termination candidate.

The concolic execution can be performed by a model checker like SPIN or JPF after slight technical modifications that ensure that lassos can be found independently of any particular property checking. The model checker searches the global state transition graph until it discovers a global control flow loop. The control flow loop might be a loop in the global state transition graph. In this case we immediately know that the program is non-terminating. However, in many cases a program is non-terminating but has no or only extremely long loops in its global state transition graph. Consider for instance the following program:

```
active proctype P(){
  int i = 0;
  do
  :: i++;
  od;
}
```

In these cases the model checker normally is not able to show non-termination by searching for loops in the global state transition graph because of memory and runtime limitations. In order to be able to prove non-termination for those cases we search for global control flow loops instead of loops in the global state transition graph and apply a more sophisticated non-termination proof technique on the found executions.

Example 1. *Consider the model of Figure 2. One lasso shaped execution which is generated by the adopted version of SPIN is the following:*

```
(((i-j)>=1)); f21!a; f12!c; f12?c; f21!b; f21?a; f12!d; ((i>=0)); i++; f12?d; (1);
f21!c; f12!a; f12?a; f21!a; f21?b; f12!d; ((j<-(2))); f12?d; (1); j--; f21!b; f12!b;
f12?b; f21!c; f21?c; f12!d; f12?d; (1);
```

As it will be shown in the next sections, this execution indeed leads to a non-terminating program behavior.

If an execution with a lasso shaped control flow is found, then the model checker generates the sequence of program statements executed on this execution. The sequence of statements always consist of a stem part, which we refer to by `<stem>`, and a cycle part, which we refer to by `<cycle>`. In the second part of the candidate generation (marked T) the executions are rewritten to a program of the following form:

```
<stem>
while(true) {
  <cycle>
}
```

```java
public static void loop() {
  final byte a = 1; final byte b = 2;
  final byte c = 3; final byte d = 4;
  final byte radix = 5;

  int q = 0; int l = 0; int p = 1;

  int i = 0;
  while (true) {
    i++;

    q = q * radix + a; p *= radix; l++;    // enqueue a
    if((q * radix) / p == a) {             // dequeue a
      q = q % (p / radix); p /= radix; l--;
    } else  break;
    q = q * radix + b; p *= radix; l++;    // enqueue b
    q = q * radix + a; p *= radix; l++;    // enqueue a
    if((q * radix) / p == b) {             // dequeue b
      q = q % (p / radix); p /= radix; l--;
    } else  break;
    q = q * radix + b; p *= radix; l++;    // enqueue b
    q = q * radix + a; p *= radix; l++;    // enqueue a
    q = q * radix + b; p *= radix; l++;    // enqueue b
    if((q * radix) / p == a) {             // dequeue a
      q = q % (p / radix); p /= radix; l--;
    } else  break;
    if((q * radix) / p == b) {             // dequeue b
      q = q % (p / radix); p /= radix; l--;
    } else  break;
  }
}
```

Figure 4: Simple example of a non-termination candidate.

The `while(true)`-loop will be exited by a `break` command every time a condition in `<cycle>` evaluates to another value than in the original execution of `<cycle>` after `<stem>`. This way it is ensured that the non-termination of the while-program implies the non-termination of the original program. Figure 4 shows a simple example of a rewritten execution path.

Example 2. *The first part of the execution path of Example 1 rewritten as a Java program is shown in Figure 5.*

Furthermore, we calculate which messages $out(w) \in \Sigma^*$ are added to the queue during one execution w of the loop of the candidate and which messages $in(w) \in \Sigma^*$ are removed from the queue. Consider for example the non-termination candidate of Figure 4. Here $in(w)$ is $abab$ and $out(w)$ is $ababab$. This information is important for the invariant generation in phase G.

Example 3. *In addition to the candidate of Example 2 SPIN determines:*

$$in_{f12}(w) = cdadbd \qquad out_{f12}(w) = cdadbd$$
$$in_{f21}(w) = abc \qquad out_{f21}(w) = abcabc$$

In the example in Figure 4 for each buffer both its length l as well as its leading exponent $p = r^{l-1}$ are logged. This has practical reasons: Since most programming languages, such as Java in the example, do not provide a built-in pow operator, it is convenient to log $p = r^l$

```
public static void run() {
  //mtype
  final byte a = 1; final byte b = 2;
  final byte c = 3; final byte d = 4;
  //proctypes
  P0 p0 = null;  P1 p1 = null;
  //variables
  int f12 = 0;   int f21 = 0;
  int f12l = 0;  int f21l = 0;
  int f12p = 1;  int f21p = 1;
  //processes
  if(p0 == null)  p0 = new P0();
  if(p1 == null)  p1 = new P1();

  while(true) {
    //(((i-j)>=1))
    if(!(((p0.i-p0.j)>=1)))  break;

    //f21!a
    f21=f21*5+a; f21p*=5; f21l++;
    //f12!c
    f12=f12*5+c; f12p*=5; f12l++;

    //f12?c
    if((f12*5) / f12p == c) {
      f12=f12%(f12p/5); f12p/=5; f12l
          --;
    } else  break;

    //f21!b
    f21=f21*5+b; f21p*=5; f21l++;

    [...]
  }
}
```

Figure 5: The first part of the translated execution of Example 1.

in order to calculate, for instance, $q \bmod r^l = q \bmod p$ efficiently. It also simplifies the non-termination proof in phase P if the length l of a queue is logged.

With the help of the values of $in(w)$ and $out(w)$ we generate a non-termination invariant for the candidate as discussed in the next section.

3.3 Generating invariants (phase G)

The idea of the non-termination invariant for queues is that the queue content q should always have a form which cannot disable the executability of the loop. That is, $q = x^i$ and $in(w) = x^j$ $(i, j \geq 0)$ are repetitions of a common sub-word x. For instance, consider again the loop of Figure 4. In this case the queue content is an element of $(ab)^*$. Thus q and $in(w)$ are repetitions of the word ab. If we have determined which messages $in(w) = x_0 \dots x_{n-1}$ are removed from the queue q on an execution w of the loop and which messages $out(w) = y_0 \dots y_{m-1}$ are added to q, then we can formulate the following invariant:

$$(q \geq 0 \wedge l \geq 0) \wedge \big((q = 0 \wedge l = 0) \vee (q = \sum_{i=0}^{l-1} x_{(i \bmod (m-n))}\, r^i \wedge (l - l_0) \bmod (m-n) = 0) \big) \tag{1}$$

Note that l is the length of the queue whereas l_0 is the length of the queue before the first execution of the loop. The invariant states that

- the queue q is a natural number and has at least length 0 $(q \geq 0 \wedge l \geq 0)$, and
- the queue is either empty $(q = 0 \wedge l = 0)$, or
- the length of q minus l_0 is a multiple of $m - n$ and q has the form $q = xx \dots xz$ where $x = x_0 x_1 \dots x_{(m-n-1)}$ and z is a prefix of x. This implies that q and $in(w)$ are in principle repetitions of the same sub-word x. However, the buffer content before the execution of the loop may add a prefix z of x to the end of q.

Of course, this is only the part of the invariant which ensures that the loop will not terminate because of the content of q. The complete non-termination invariant is the conjunction of

Normal part:

```
  p0.i >= 0
& p0.j <= -3
```

Part for buffer ***f12****:*

```
& f12 >= 0
& f12l >= 0
& f12p >= 1
& (
    f12 = 0 & f12l = 0 & f12p = 1
)
```

Part for buffer ***f21****:*

```
& f21 >= 0
& f21l >= 0
& f21p >= 1
& (
    f21 = 0 & f21l = 0 & f21p = 1
  |
      f21 = polynomial(0, f21l - 1, 5,
          f_x123)
    & f21l % 3 = 0
    & f21p = pow(5, f21l)
)
```

Figure 6: Invariant for the non-termination candidate of Figure 5. In this example the radix for buffer `f12` and buffer `f21` is 5. Furthermore the buffer `f12` is empty after each execution of the loop ($in_{f12}(w) = out_{f12}(w)$) and for buffer `f21` it holds $in_{f21}(w) = 123$ and $out_{f21}(w) = 123123$.

Invariant (1) for all queues of the system and the invariants generated by failed proof attempts generated as in [13]. The idea of the invariant we describe above has some similarities with the buffer unboundedness condition of [10]. Note that it is easy to generate the invariant automatically. Furthermore, we suspect that x_i does not have to be a constant but can also be a symbolic variable or expression. However, we currently lack a proof of this assertion.

Example 4. *With the help of $in_{f12}(w)$, $out_{f12}(w)$, $in_{f21}(w)$ and $out_{f21}(w)$ of Example 3 we generate for the example of Figure 2 an invariant consisting of the part for buffer* `f12` *and the part for buffer* `f21` *of Figure 6.*

3.4 Performing proof attempts on the candidates (phase P)

The non-termination proof attempts are performed as in [13] with KeY as theorem prover by verifying the following JavaDL [1] formula:

```
\[{ lasso.run(); }\] false
```

Here, `lasso.run()` is a place holder for the class / method name where the while-program has been implemented in. The formula evaluates to true iff the method in `lasso.run()` does not terminate. During the proof, the invariant of Section 3.3 is used to show the non-termination of the `while(true)`-loop. If KeY is not able to prove the non-termination of the candidate with the help of the initial invariant, then the invariant is refined with the help of the failed proof attempt using the approach of [13]. Afterwards a new proof attempt is started.

Example 5. *While KeY is trying to prove the non-termination of the program in Figure 5, the invariant of Example 4 is refined to the complete invariant of Figure 6. With the help of this invariant KeY is able to prove the non-termination of the program.*

As stated in the introduction, the approach of [13] is not suitable for our needs on its own, because the approach is not applicable to multi-threaded programs. Furthermore, as seen in Sections 3.2 and 3.3, some information has to be derived from the loop with the help of the concolic execution in order to be able to generate an appropriate invariant. Since our invariants

```
powMultPow {
\find (
  pow(x, a) * pow(x, b)
)
\sameUpdateLevel
\replacewith (
    pow(x, a + b)
)
\heuristics(userTaclets1)
};

powEqReduction {
\find (
  pow(Z(n), a) = Z(m)
)
\sameUpdateLevel
\replacewith (
  \if(Z(m) <= 0)
  \then(false)
  \else(\if(a = 0)
    \then(1 = Z(m))
    \else(\if(Z(m) % Z(n) = 0)
      \then(pow(Z(n), a-1) = Z(m) / Z(n
          ))
      \else(false)))
)
\heuristics(userTaclets1)
};

estimatePow {
\find (
    pow(a, x) <= Z(neglit(n))
)
\replacewith (
    false
)
\heuristics(userTaclets1)
};

polynomial_DivPow {
\find (
  polynomial(a, b, Z(r), f) / pow(Z(r),
      c)
)
\sameUpdateLevel
\replacewith (
    polynomial(max(0, a - c), b - c, Z(
        r), permut(f, c % val_n(f)) )
)
\heuristics(userTaclets1)
};
```

Figure 7: Examples of rewrite rules to handle powers and polynomials.

of Section 3.3 contain powers and polynomials we need to be able to handle those functions at least rudimentary in our proof. Because the KeY system does not provide built in functions for powers and polynomials we have to introduce new uninterpreted function symbols and rules for those function symbols to KeY. This way KeY can be extended with the ability to reason over powers and polynomials. However, writing sound and appropriate rules for the uninterpreted function symbols is a challenge in its own right, and providing these rules is a further contribution of our work. Note, that we are not aware of any theorem prover which is able to handle those functions out of the box.

Figure 7 shows some examples of our rule set. The main strategy behind the rules is to bring terms into a form so that we can take advantage of relations between powers and polynomials to simplify them. Such a simplification is performed for instance by the rule `polynomial_DivPow`. Note, that `polynomial(a, b, r, f)` represents the polynomial $\sum_{i=a}^{b} f(i \bmod n) \cdot r^i$, where f is a function which assigns the positions 1 to $n-1$ values. Furthermore the term $pow(x, a)$ represents the power x^a. In addition to such simplification rules our rule set contains

- rules for commonly known equivalences of powers and polynomials as in the rule `powMultPow`,

- rules to decide the correctness of equations and inequations over powers and polynomials as in the rules `powEqReduction` and `estimatePow`, and

- rules to resolve concrete powers and polynomials like the term x^4.

```
mtype = {a,b,c};

chan f12 = [100] of {mtype};
chan f21 = [100] of {mtype};

active proctype P1(){
  do
   :: f12!a; f21?c -> f12!b;
  od;
}

active proctype P2(){
  do
   :: f12?b -> skip;
   :: f21!c; f12?a -> f21!c;
  od;
}
```

Figure 8: JEJA case study

```
chan ch1 = [10] of {byte};
chan ch2 = [10] of {byte};

active proctype P1(){
  do
   :: ch1!1 -> ch2?1;
  od;
}

active proctype P2(){
inits: skip;
  do
   :: ch1?1 -> ch2!1;
   :: break;
  od;
  do
   :: ch1?1 -> ch2!1; ch2!1;
   :: break;
  od;
  goto inits;
}
```

Figure 9: PEX case study

3.5 Implementation

We have implemented our ideas by extending the SPIN and KeY tools. SPIN searches the state space of Promela models, generates lasso shaped execution paths and translates the Promela code on the execution path to a Java program as presented in Section 3.2. KeY then generates the invariants and runs the non-termination proof attempts with the help of an extended invariant generator and rule base as presented in Sections 3.3 and 3.4.

One might wonder why the Promela paths are translated to Java instead of analyzing the Promela code directly. This translation seems to be questionable since there might be inconsistencies between the semantics of Promela and Java. However, we are confident that no inconsistencies are introduced since the Promela code on the execution paths contains only boolean expressions, assignments and send and receive statements. Those have a quite unique and commonly agreed on semantics. By translating the Promela code to Java we can take advantage of KeYs ability to reason directly over Java code. Otherwise we would have to execute the Promela code symbolically and translate the effect of the Promela code to some logic formula. KeY relieves us from this work if we provide KeY with Java code.

The ability to execute code symbolically as part of the reasoning is one reason why we use KeY. Another reason is that KeY provides a rule base which can be extended very easily. This way we are able to handle the invariants of Section 3.3 as discussed in Section 3.4. Finally, we can take advantage of the invariant generator of [13] and its existing integration into KeY.

4 Experiments

Since our implementation of the approach is still in an early prototype state, we were able to test the approach only on four case studies. The non-termination proof was successful on all of them.

```
mtype = { ini, ack, dreq, data, shutup,
     quiet, dead };

chan M = [100] of { mtype };
chan W = [100] of { mtype };

active proctype Mproc()
{
  W!ini; M?ack;

    timeout ->
    if
    :: W!shutup
    :: W!dreq;
       M?data ->
       do
        :: W!data
        :: W!shutup; break
       od
    fi;

  M?shutup; W!quiet; M?dead
}

active proctype Wproc()
{
  W?ini; M!ack;

  do
   :: W?dreq -> M!data
   :: W?data -> skip
   :: W?shutup -> M!shutup; break
  od;

  W?quiet; M!dead
}
```

Figure 10: DTP case study

Two of the case studies, the JEJA case study of Figure 8 and the PEX case study of Figure 9, are simple models with two processes communicating via two FIFO buffers. Because no further variables are involved, we are able to show the non-termination of these models with a slightly modified version of the concolic execution already in phase M. Of course, we are also able to show the non-terminiation of these models using the invariant generation and the theorem proving part of our method. No refinement steps are necessary for these models. The DTP case study of Figure 10 is a case study over a simple data transport protocol. We are able to show the non-terminiation of the model without any refinement steps for this example as well. Finally, we proved the non-termination of the example of Figure 2. In addition to queues, this example involves two variables which influence the executability of the loop. Notice that none of the approaches mentioned in the introduction is able to handle this example.

We cannot provide exact execution times since our implementation is not yet fully automated. We expect execution times of up to a minute on an Intel$^{\circledR}$ Core$^{\mathrm{TM}}$2 Duo CPU E6750 with 2.66GHz. A reason for the fairly high expected execution time is that KeY needs around 12 seconds each time it starts up and parses the sources, in addition to the computationally expensive theorem proving.

5 Conclusions

We have presented a non-termination proof technique for asynchronous concurrent programs which communicate via FIFO buffers. This approach is to the best of our knowledge

- the first approach which is able to handle non-linear concurrent programs of high level languages and
- the first approach which incorporates data types to non-termination proof techniques of high level languages.

We showed the feasibility of the approach by preliminary experiments with a prototype implemention. The implemention is based on the model checker SPIN and the general purpose verification system KeY. Even though none of the case studies we can currently handle has the size and complexity of real software systems, we are confident that the method will scale to more complex models representing real systems.

Since the second part of our analysis is based on the analysis of concrete execution candidates, we expect that our approach is also applicable to run-time analysis as described in [3]. This claim will be the subject of future research, as well as a fully automated implementation and an extension to encompass the Jeron-Jard test for unboundedness.

References

[1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[2] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM (JACM)*, 30:323–342, April 1983.

[3] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight Detection of Infinite Loops at Runtime. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE09)*, 2009.

[4] Y. Dong, X. Du, G. Holzmann, and S. Smolka. Fighting Livelock in the GNU i-Protocol: a Case Study in Explicit-State Model Checking. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 4(4):505–528, 2003.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[6] Étienne Payet and F. Spoto. Experiments with Non-Termination Analysis for Java Bytecode. *Electronic Notes in Theoretical Computer Science*, 253(5):83 – 96, 2009. Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009).

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[8] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving Non-Termination. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–158, New York, NY, USA, 2008. ACM.

[9] G. Holzmann. *The SPIN model checker: Primer and reference manual.* Addison Wesley, 2004.

[10] T. Jeron and C. Jard. Testing for Unboundedness of FIFO Channels. *Theoretical Computer Science*, 113(1):93–117, 1993.

[11] M. Kamel and S. Leue. Formalization and Validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(4):394–409, 2000.

[12] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference*, pages 263–272, New York, NY, USA, 2005. ACM.

[13] H. Velroyen and P. Rümmer. Non-Termination Checking for Imperative Programs. In *Tests and Proofs*, LNCS 4966, pages 154–170, 2008.

[14] G. von Bochmann. Finite State Description of Communication Protocols. *Computer Networks*, 2:361–372, 1978.