Verifying Parallel Algorithms and Programs Using Coloured Petri Nets

Michael Westergaard

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
m.westergaard@tue.nl

Abstract. Coloured Petri nets have proved to be a useful formalism for modeling distributed algorithms, i.e., algorithms where nodes communicate via message passing. Here we describe an approach for automatic extraction of models of parallel algorithms and programs, i.e., algorithms and programs where processes communicate via shared memory. The models can be verified for correctness, here to prove absence of mutual exclusion violations and to find dead- and live-locks. This makes it possible to verify software using a model-extraction approach using coloured Petri nets, where a formal model is extracted from runnable code. We extract models in a manner so we can also support a model-driven development approach, where code is generated from a model, enabling a combined approach, supporting extracting a model from an abstract description and generation of correct implementation code. We illustrate our idea by applying the technique to a parallel implementation of explicit state-space exploration.

Our approach builds on having a coloured Petri net model corresponding to the program and using the model to verify properties. We have already treated generation of code from coloured Petri nets, so in this paper we focus on the translation the other way around. We have an implementation of the translation from code to coloured Petri nets.

1 Introduction

Parallel and distributed computing address important problems of scalability in computer science, where some problems are too large or complex to be handled by just one computer. Until now, the focus has mostly been on distributed algorithms, i.e., algorithms running on multiple computers communicating via a network, as access to parallel computers, i.e., computers capable of running multiple processes communicating via shared memory (RAM), has been limited. For this reason, there are many papers on modeling distributed algorithms, such as network protocols [1–4]. With the advance of cheap multi-core processors and cheap multi-processor systems, access to multiple cores has become more common, and the development and analysis of algorithms for parallel processing becomes very interesting. As parallel computing allows much faster communication between processes, tasks that were not previously feasible or efficient to do concurrently, become interesting. In this paper, we present our experiences developing parallel algorithms with synchronization mechanisms. The algorithms are verified by means of *coloured Petri nets* (CPNs) [5]. This work was motivated by our need for a parallel state-space exploration algorithm. In this paper, we provide an approach that allows us to extract a model for analysis from a program or abstractly described algorithm in a systematic way. We do this in a way that allows us to automatically generate a (skeleton) implementation of the algorithm subsequently. We use a simple state-space algorithm as an example, but the approach has also been used for other parallel algorithms, such as parts of a protocol for operational support [6], and is generally applicable for other parallel algorithms as well. The method can also be used for non-parallel algorithms, but we focus on the new challenges arising when moving from sequential to parallel processing.

Classically, formal models can partake in a development in two different ways: by extracting an implementation from a model, which we call *model-driven software engineering*, or by extracting a model from an implementation, which we call *model-extraction*. Our focus in this paper is on model-extraction but in a way that allows us to also do code generation, thereby allowing a new combined approach. The model-driven engineering approach is shown in Fig. 1 (top) and shows that we start with a model that is verified according to one or more properties. If it satisfies the desired properties, we can extract a program, otherwise we refine the model. Examples of this approach are within hardware synthesis [7, 8], using a CPN simulator to drive a security system [9], or general code generation from a restricted class of CPNs [10]. The model-extraction approach is shown in Fig. 1 (bottom). Here, we do not start with a model, but rather with a program. From the program, we extract a model and verify it for correctness.

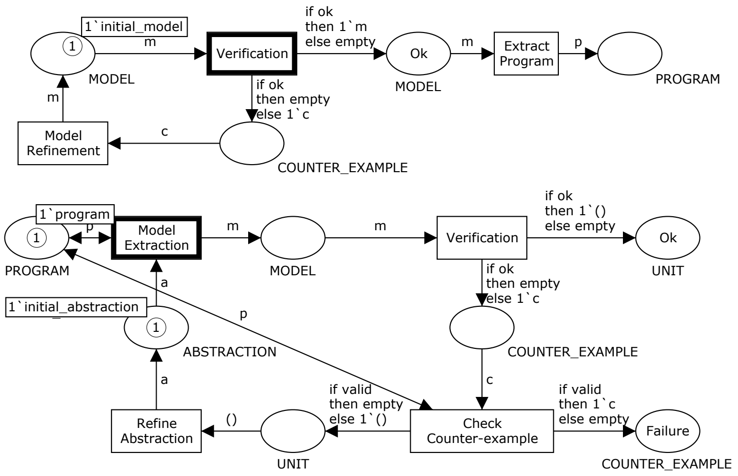


Fig. 1. Model-driven software engineering (top) and model-extraction (bottom)

If an error is found, the resulting error-trace is replayed on the original program to determine if it can be reproduced there. If not, the abstraction used to extract the model is refined and the cycle restarts. This approach is rarely used in the high-level Petri net world, but is employed by, e.g., FeaVer [11] to translate C code to PROMELA code usable in SPIN [12], Java PathFinder [13] to translate Java programs to PROMELA, SLAM [14] for automatically translating C device drivers to boolean programs, BLAST [15] for model-checking C programs, and many other tools.

The model-driven software engineering and model-extraction approaches have different strengths and weaknesses. The main strength of the model-driven software engineering approach is that it is possible to verify an algorithm before implementation and we can even get a guaranteed-correct (template) implementation with little or no user-interaction. The disadvantage is that the approach is of little use for already existing software. The model-extraction approach precisely alleviates this by extracting a model from an existing implementation automatically, ensuring there is correspondence between the model and implementation. The main disadvantage is that we need an implementation of a, perhaps faulty, algorithm before analysis can start.

We would like to provide a translation supplying as many of the strengths of these approaches as possible. Here we aim at supporting model-extraction in a way that allows subsequent code generation. In [10] we introduce the sub-class of CPNs called *process-partitioned coloured Petri nets* (PP-CPNs), which allows us to generate executable code from a PP-CPN model, thus supporting the model-drive software engineering approach. The focus of this paper is to support the model-extraction approach. We aim at doing so in a way that the extracted model later can be used for code generation, i.e., we make sure that the extracted models are PP-CPNs. This has the advantage that not only do we support both approaches in Fig. 1, we also provide foundations for a third merged approach, shown in Fig. 2. Here, we can do round-trip engineering, where we take an implementation as input, verify and correct it on the level of a model, and, instead of manually updating the implementation, use the automatically generated one as input for the next iteration. We can of course also do the modification directly in the code based on the counter-example if desired. When we find no more errors, we directly transfer the model to Ok for program extraction. This is only necessary if we wish to extract a program in a language different from the input language. The model never terminates, reflecting that the development and error checking is an ongoing process.

The use of (a slightly restricted class of) CPNs allows us to refine data-structures as much as required and even using actual data-structures of the original algorithm or program. We assume that we additionally have or can derive an abstraction of all data-types used. Derivation of abstractions of the data-types used can be done by the user or automatically using counter-example guided abstraction refinement (CEGAR) [16] as implemented in SLAM [14] and BLAST [15]. CEGAR automatically improves abstractions by replaying errors found in an abstract model on the original program and using information about

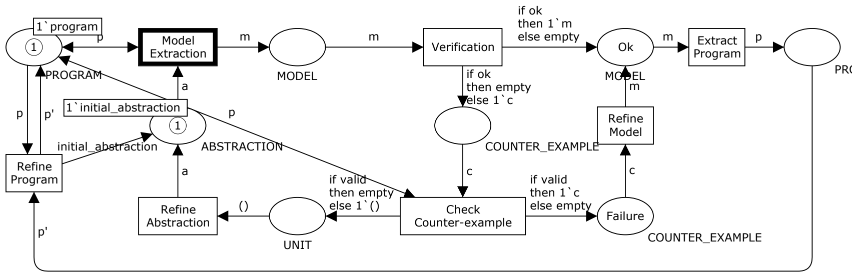


Fig. 2. Approach combining model-driven software engineering and model-extraction

why a given error-trace cannot be replayed in the original program to refine the abstraction. Here we are not concerned with abstraction refinement, and assume the user takes care of this. In this paper we focus on model-extraction, and present an implementation able to automatically generate a PP-CPN model from a program in a simple but expressive language.

The rest of this paper is structured as follows: in the next section, we introduce process-partitioned coloured Petri nets as defined in [10] and a simple algorithm for state-space generation which we use as a running example to illustrate our idea. In Sect. 3, we introduce our approach to generating PP-CPN models from algorithms using a naive parallel version of the algorithm presented in Sect. 2. In Sect. 4, we identify a problem in the original parallelization, fix the problem and show that the problem is no longer present in a modified version. Finally, in Sect. 5, we sum up our conclusions and provide directions for future work. Readers are assumed to have some familiarity with general coloured Petri nets but not PP-CPNs. An earlier version of this paper has been published as [17]. The earlier version did not have an implementation of the translation, so the description of the translation is much more detailed in this version. We also provide more details on abstraction refinement.

2 Background

In this section, we briefly introduce CPNs and process-partitioned CPNs as defined in [10]. We also give a simple algorithm for explicit state-space generation which we use as an example in the remainder of the paper.

Coloured Petri Nets. The definition of process-partitioned CPNs use the definitions and notation of CPNs from [5, Chap. 4] as a basis. CPNs consist of *places*, *transitions*, and *arcs*. Places are typed and arcs have expressions that may contain variables. The model-driven software engineering approach shown in Fig. 1 is a CPN, and here places are drawn as ovals, transitions as rectangles, and they are connected with arcs. Places and transitions can have names; in Fig 1 names are drawn inside the figure representing places/transitions, e.g., we have a place named Ok (the remaining places do not have names) and a transition named Verification. Places have *types*, corresponding to types of variables in normal programming languages; the place types are typically written below or below and to the right of places; in Fig. 1, the place Ok has type MODEL. Places additionally can have a *marking*, a multi-set of *tokens* (values) residing on the place. We write the marking of a place in a rectangle close to the place and the total number of tokens in a circle on the place; in Fig. 1, the unnamed place to the top left has a marking of 1'initial_model, i.e., a single token with value initial_model. All other places contain no tokens, which is per convention not shown. The marking of a place before executing transitions is the *initial marking*; the model in Fig. 1 is shown in its initial marking.

Places and transitions are connected by arcs having *arc expressions*, e.g., if ok then 1'm else empty on the arc from Verification to Ok. Arc expressions are expressions that may contain free typed *variables*. We use the terms *input arc* and *output arc* to refer to all arcs having a given place/transition as source and destination, respectively. We extend this to also include *input place* and *output place* to denote all source places of input arcs of a transition and target places of output arcs.

A transition and an assignment of values to all free variables on arcs surrounding the transition is called a *binding element* or just a *binding*. We write a binding by writing the name of the transition and a list of assignments to all variables in brackets, like Verification⟨m=initial_model,ok=true,c=[]⟩. A binding is enabled in a marking if evaluating the expressions on all input arcs result in a multi-set of tokens that is a subset of the tokens residing on the corresponding input place in the marking. In the example in Fig. 1, the binding element Verification⟨m=initial_model,ok=true,c=[]⟩ is enabled as the unnamed place at the upper left does contain a token with the value initial_model. The binding element Model Refinement⟨m=initial_model,c=[]⟩ is not enabled, as the counter example place contains no tokens (and thus in particular no token with value []). A binding can be *executed*, by removing all tokens from input places according to evaluations of the corresponding arc expressions and adding new tokens to all output places according to arc expressions on output arcs. We say that a transition is enabled if there are any enabled bindings of the transition. In Fig. 1 only Verification is enabled, and enabling of transitions is indicated by a bold outline. Transitions can additionally have *guards*, an extra expression written in square brackets next to it. These further limit the enabling as they have to evaluate to true for the transition to be enabled. No transition in Fig. 1 has a guard, but the transition assign in Fig. 6 (i) has guard *[id1' = id2]*.

Hierarchical Coloured Petri Nets. CPNs have a module concept, where *subpages* are represented by *substitution transitions*, inducing a hierarchy of the pages. Graphically, we draw substitutions as transitions using a double outline. The model in Fig. 6 (c) has two substitution transitions, Then and Else. Parameters to subpages are specified as *port places* that have a direction (In, Out, or I/O). The model fragment in Fig. 6 (c) has three parameters, S of type In, and E and R of type Out. Port places are assigned to *socket places* on the page of the substitution transition. These assignments are not shown explicitly graphically, but are often obvious from context (port and corresponding socket places have the same name or there is only one port place with the correct type). For example, we would use the fragment in Fig. 6 (c) as a subpage of the S1 substitution transition of the fragment in Fig. 6 (b). The port place S of fragment (c) would be assigned to the socket place S of (b), E of (c) to the unnamed place of (b), and R to R. The semantics is defined as replacing the substitution transition by the contents of the subpage, merging places in a port/socket relationship. We call this procedure *flattening*. This can be done automatically and is reversible, so in the remainder of this paper, we allow introducing and removing hierarchy whenever convenient, using it only to aid in presentation.

Process-Partitioned Coloured Petri Nets. In [10] we introduce the notion of *process-partitioned CPNs* (PP-CPNs). All models in the following sections of this paper are PP-CPN models. While this is important for the generation of code from extracted models, it is not important for the actual extraction, so we shall not go into too much detail about the nature of PP-CPNs, but only give a general idea of this subclass. Interested readers are invited to refer to [10] for a full formal definition of PP-CPNs.

PP-CPNs are CPNs, which are partitioned into separate kinds of processes. In this paper, we are only interested in models containing a single kind of process, so we just look at *process subnets* (Def. 2 in [10]). A single process subnet is a PP-CPN, but not necessarily the other way around. In this paper, whenever we talk about PP-CPNs, we assume they consist of exactly one process subnet. A process subnet is a CPN with a distinguished *process colour set* serving as a process identifier. The models in Fig. 7 are examples of PP-CPNs (we provide a detailed description of the models in Sect. 3). The process colour set of this model is PROCESS. The places of a process subnet are partitioned into *process places*, *local places*, and *shared places* (in [10], we additionally introduce *buffer places* for asynchronous communication between processes, but these are not used here). These places correspond to the control flow, local variables, and shared variables of programs. Process places must be typed by the process colour set (in the example, entry and exit and all places s_i are process places), local places must be typed by a product of the process colour set and any other type (in the example, s, ss, and the condition places), and shared places can have any type (in the example, waiting and visited).

In the initial marking, exactly one of the process places contains all tokens of the process colour set and all remaining process places are empty (modeling that all processes start in the same location in the program). In Fig. 7, the entry place contains all (two) processes and none of the other process places contain any tokens. Local places initially contain exactly one token for each process so that if we project onto the component of the process colour set, we obtain exactly one copy of all values of the set (modeling that all local variables must be initialized). This is seen on s, ss, and the condition places of Fig. 7.

All shared places contain exactly one value (modeling that shared variables must be initialized). This is seen on visited and waiting in Fig. 7. All arc expressions must ensure that tokens are preserved.

We have chosen to adopt the notion from [10] that we cannot create new processes or destroy processes even though nothing in our approach breaks if we allow dynamic instantiation and destruction of processes. This is mainly for simplicity as we did not need dynamic instantiation in our examples. We allow a slightly relaxed syntax for PP-CPNs here compared to [10], as we allow using constants on input arcs, relying on pattern matching to determining enabling of transitions depending on input data instead of guards. This is done for legibility of patterns for conditionals and can easily be undone by instead using a variable and checking for the correct value in the guard.

State-space Generation. State-space generation is a means of analysis of formal models, such as the ones specified by means of CPNs. A simple implementation is shown in Fig. 3. We start in the initial marking of the model and compute all enabled bindings. We then systematically execute each, note the markings we reach by executing bindings, store them in Waiting, and repeat the procedure for each of these newly discovered markings. To also terminate in case of loops, we store all markings for which we have already computed successors in Visited and avoid expanding them again. We often call a marking a *state* in the context of state-space analysis.

```
WAITING := MODEL.initial()
VISITED := MODEL.initial()
while not WAITING.isEmpty() do
    s := WAITING.head()
    WAITING := WAITING.remove(s)
    // Do any handling of s here
    for all b in s.enabled() do
        ss := s.execute(b)
        if not VISITED.contains(ss) then
            WAITING := WAITING.add(ss)
            VISITED := VISITED.add(ss)
        endif
    endfor
endwhile
```

Fig. 3. State-space exploration algorithm

3 Approach

We introduce our approach to verifying parallel algorithms by a parallel version of the algorithm for generating state-spaces shown in Fig. 3. The basic idea is to use the loop of Fig. 3 for each process and share the use of Waiting and Visited, naturally with appropriate locking. From this algorithm, we illustrate our approach to extract a PP-CPN model. The approach is completely general, though.

A simple way to parallelize Fig. 3 is shown in Fig. 4. The comments in lines 1, 6, and 23 are reintroduced in a subsequent refinement. In this first version, we initialize as before (ll. 2–3). We have moved the main loop to a separate procedure, *computeStateSpace*. We perform mostly the same loop as before (ll. 8–17), but instead of testing for emptiness and picking an element of the queue in three steps, we do so using a procedure *pickAndRemoveElement* (ll. 7 and 16). The implementation of *pickAndRemoveElement* (ll. 22–32) does the same as we did before, except we return a bottom element *notFound* if no elements are available, and use that in the condition of the loop (l. 8). This forces us to perform the pick in two places: before the first invocation of the loop (l. 7) and at the end of the loop (l. 16). Handling of states (l. 9) and iteration over all enabled bindings (ll. 10–15) is the same as before. Now, instead of checking if a state is a member of Visited and conditionally adding it to the set, we do both in a single step as shown in the procedure *addCheckExists* (ll. 12 and 34–40). We do this under the assumption that adding an element to the set does nothing if the element is already there. If the state was not already in Visited, we add it to Waiting (l. 13). The reason for this re-organization is that we now assume that *pickAndRemoveElement*, *addCheckExists*, and the access to Waiting in line 25 are atomic. We ensure this by acquiring locks in *pickAndRemoveElement* and *addCheckExists* (ll. 24 and 35). This allows us to start two instances of *computeStateSpace* in parallel in line 22. We will not argue for the correctness of either Fig. 3 or Fig. 4, but note that it is easy to convince ourselves that if one is correct so is the other, with the assumption that *pickAndRemoveElement* and *addCheckExists* happen atomically.

```
 1: // bool WAITING
 2: WAITING := MODEL.initial()
 3: VISITED := MODEL.initial()
 4:
 5: proc computeStateSpace() is
 6:    // bool s
 7:    s := pickAndRemoveElement()
 8:    while not s = notFound do
 9:       // Handle s here
10:       for all b in s.enabled() do
11:          ss := s.execute(b)
12:          if addCheckExists(ss) then
13:             WAITING := WAITING.add(ss)
14:          endif
15:       endfor
16:       s := pickAndRemoveElement()
17:    endwhile
18: endproc
19:
20: computeStateSpace() ||
           computeStateSpace()

22: proc pickAndRemoveElement() is
23:    // bool s
24:    lock pick
25:       if WAITING.isEmpty() then
26:          return notFound
27:       endif
28:       s := WAITING.head()
29:       WAITING := WAITING.remove(s)
30:       return s
31:    unlock
32: endproc
33:
34: proc addCheckExists(s) is
35:    lock add
36:       contains := VISITED.contains(s)
37:       VISITED := VISITED.add(s)
38:       return not contains
39:    unlock
40: endproc
```

Fig. 4. Naive parallel state-space algorithm

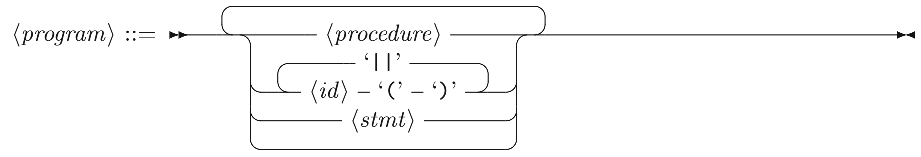
3.1 Model Extraction

To go from Fig. 4 to a PP-CPN model, we first extract the control-flow of the algorithm including generating representations of data, and then we refine the update of data until we can prove the desired properties of the model.

The main idea of our translation is shown in Fig. 5. We start with a program, and perform some simplifications of it to ease translation. We then translate the simplified program to a CPN model using templates for each construct. Finally, we simplify the resulting model to ease analysis.

We assume we have a parsed program with the syntax seen below. All algorithms presented in this paper adhere to the syntax. We use this syntax instead of a stock language for two reasons: To keep the syntax independent of a concrete language and for ease of parsing for simple prototyping. While this may seem off as our goal is to verify existing programs, it suffices for a prototype implementation, and our translation works on abstract syntax only, so adapting to a concrete syntax is only as much work as writing the parser. We allow unparsed content in our programs; this should be further defined to improve abstraction refinement, but this is not our primary focus here, so we have decided to allow this for simplicity.

Our syntax is simple but expressive, and contains most of the elements one would expect in a programming language. A program is a list of procedure definitions, parallel procedure invocations, and statements (which are assumed to be atomic except for procedure invocation). A procedure definition has a list of parameters (which may or may not have a specified type) and comprises a list of statements. Types can either be specific types or any type identifier. Statements are either definitions of variables, assignments of expressions to variables, if statements, while loops, for all loops, repeat loops, sections protected by a lock, returns from a procedure, or an expression. Expressions can be parenthesized or negated, and are either function calls, identifiers, or anything (unparsed content). Expressions can always be followed by a list of method invocations.



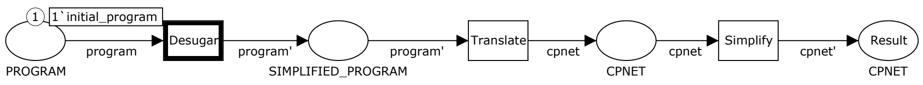
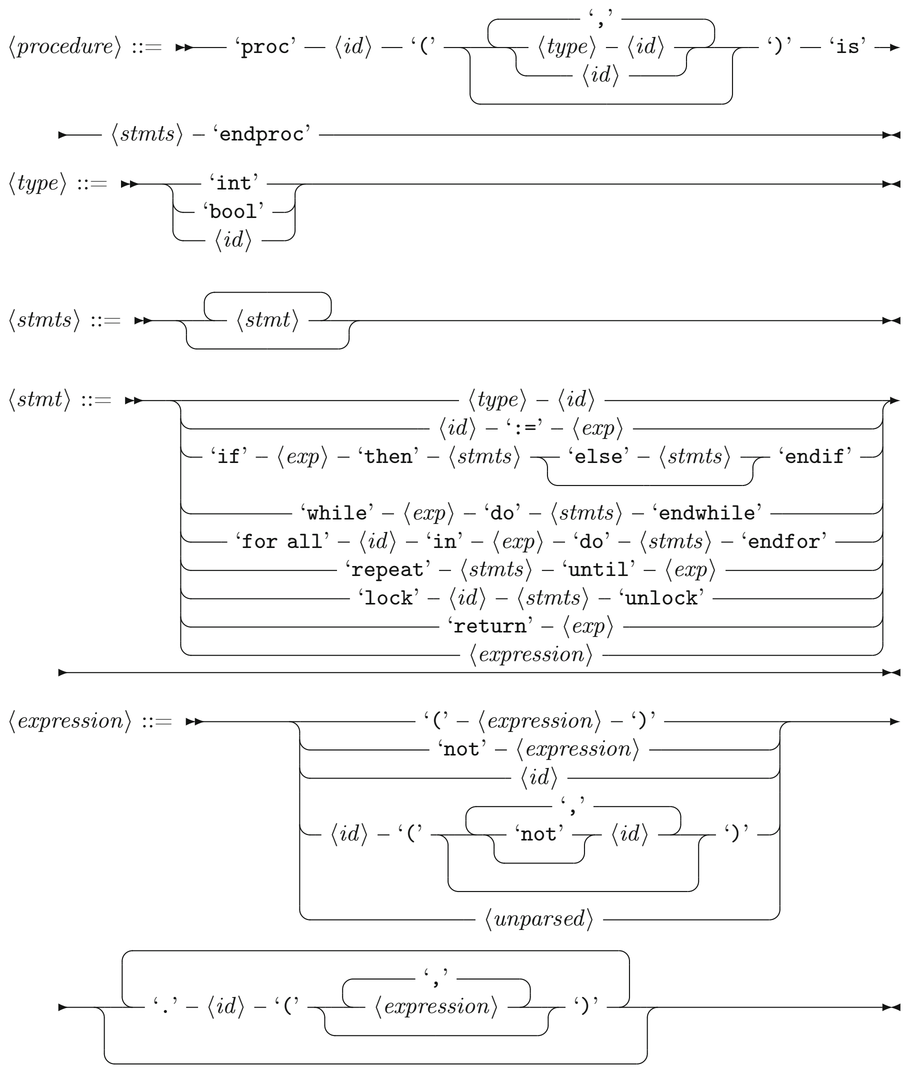
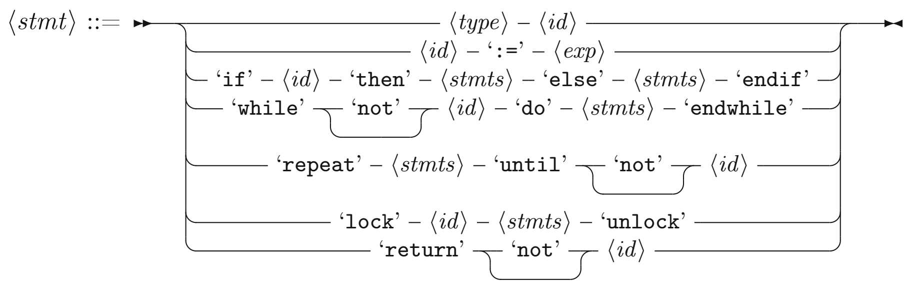
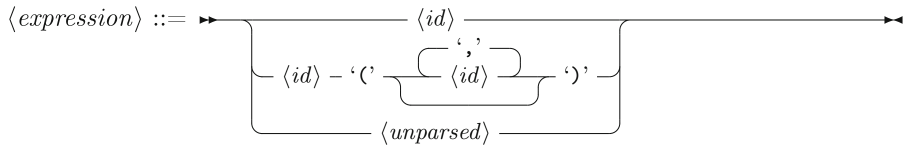


Fig. 5. Translation approach



While our grammar is useful for humans, it is not suitable for translation to CPN models. Instead, we rewrite our programs according to syntactical rules, to simplify the programs. We replace for all loops with while loops, i.e., '`for all`' ⟨*id*⟩ '`in`' ⟨*expression*⟩ '`do`' ⟨*stmts*⟩ '`endfor`' ≡ '`all_`'⟨*id*⟩ '`:=`' ⟨*expression*⟩ '`while`' '`all_`'⟨*id*⟩ '`.`' '`hasMore`' '`(`' '`)`' '`do`' ⟨*id*⟩'`:=`' '`all_`'⟨*id*⟩ '`.`' '`getFirst`' '`(`' '`)`' ⟨*stmts*⟩ '`endwhile`'. Then, we remove all method invocations (as they are not supported by Standard ML, the inscription language of CPN Tools). We do this by replacing method invocation by simple invocation, i.e., ⟨*exp*⟩ '`.`' ⟨*id*⟩ '`(`' ⟨*exp_1*⟩ '`,`' …'`,`' ⟨*exp_n*⟩ '`)`' ≡ ⟨*id*⟩ '`(`' ⟨*exp*⟩ '`,`' ⟨*exp_1*⟩ '`,`' …'`,`' ⟨*exp_n*⟩ '`)`'. We also force the use of the else path in if statements, introducing one with an empty statement list if neccesary. Finally, we simplify statements, so only variables or negated variables are used in expressions other than assignments. This is done using several rules, introducing temporary variables as needed. One such rule is `'while'` $\langle exp \rangle$ `'do'` $\langle stmts \rangle$ `'endwhile'` $\equiv$ `'condition'` `':='` $\langle exp \rangle$ `'while'` `'condition'` `'do'` $\langle stmts \rangle$ `'condition'` `':='` $\langle exp \rangle$ `'endwhile'`. We could also translate repeat loops to while loops, but have chosen not to as this yields simpler models. We allow negations in most conditions (but not in if statements). Thus, we get programs adhering to a simplified grammar replacing the $\langle stmt \rangle$ and $\langle exp \rangle$ productions with:


⟨stmt⟩ ::=
⟨type⟩ – ⟨id⟩
⟨id⟩ – ':=' – ⟨exp⟩
'if' – ⟨id⟩ – 'then' – ⟨stmts⟩ – 'else' – ⟨stmts⟩ – 'endif'
'while' 'not' ⟨id⟩ – 'do' – ⟨stmts⟩ – 'endwhile'
'repeat' – ⟨stmts⟩ – 'until' 'not' ⟨id⟩
'lock' – ⟨id⟩ – ⟨stmts⟩ – 'unlock'
'return' 'not' ⟨id⟩



Extracting the control-flow of a procedure consists of creating the process places and transitions of the model. We do that using templates, very similar to the workflow-patterns [18] for low-level Petri nets. In Fig. 6 we show the patterns necessary to translate programs using our simple pseudo-code language to a PP-CPN model. The first patterns (a-f) define the basic control flow and are atomic actions, sequence of statements, conditional, while loop, repeat loop, and critical section, corresponding to productions 3–6 of $\langle stmt \rangle$ and $\langle stmts \rangle$. The type P is the process colour set and for each pattern, the place S is the start place, E the end place, and R a special place pointing to the end of the procedure. All places created are process places except for the Mutex place, which is a shared place, and the condition places, which are local places. The intuition of each fragment is that we start a block in S and execute towards E. At any point, we may also do early termination, going to R. Fragments (a-e) should be self-explanatory; in fragment (f) we make sure to release the mutex no matter how we leave the critical section. Also, the Mutex place is a shared place, which means it is shared among all instances of the fragment. The top level of each procerdure is shown as fragment (g), where we see we get all input parameters (if any), p_id1. . . p_id$k$, all global variables (g_id1. . . g_id$m$, and all local variables (l_id1. . . l_id$n$). On normal termination we move a token from E to R to ensure a single exit point, and store the return value on Return Value. Explicit returns are handled by fragment (h), which diverts control to the R place and updates the Return Value. All global and local variables, parameters, and the return value place is also passed on to each fragment, but this is not shown here for simplicity.

In the initial abstraction, we approximate the type of all variables with UNIT, and local and shared places are not connected to transitions for reading. All conditions are of type BOOL and assigned values non-deterministically. For each variable, we introduce a place with the same name as the variable and two CPN variables, one with the same name as the original variable and a version with a

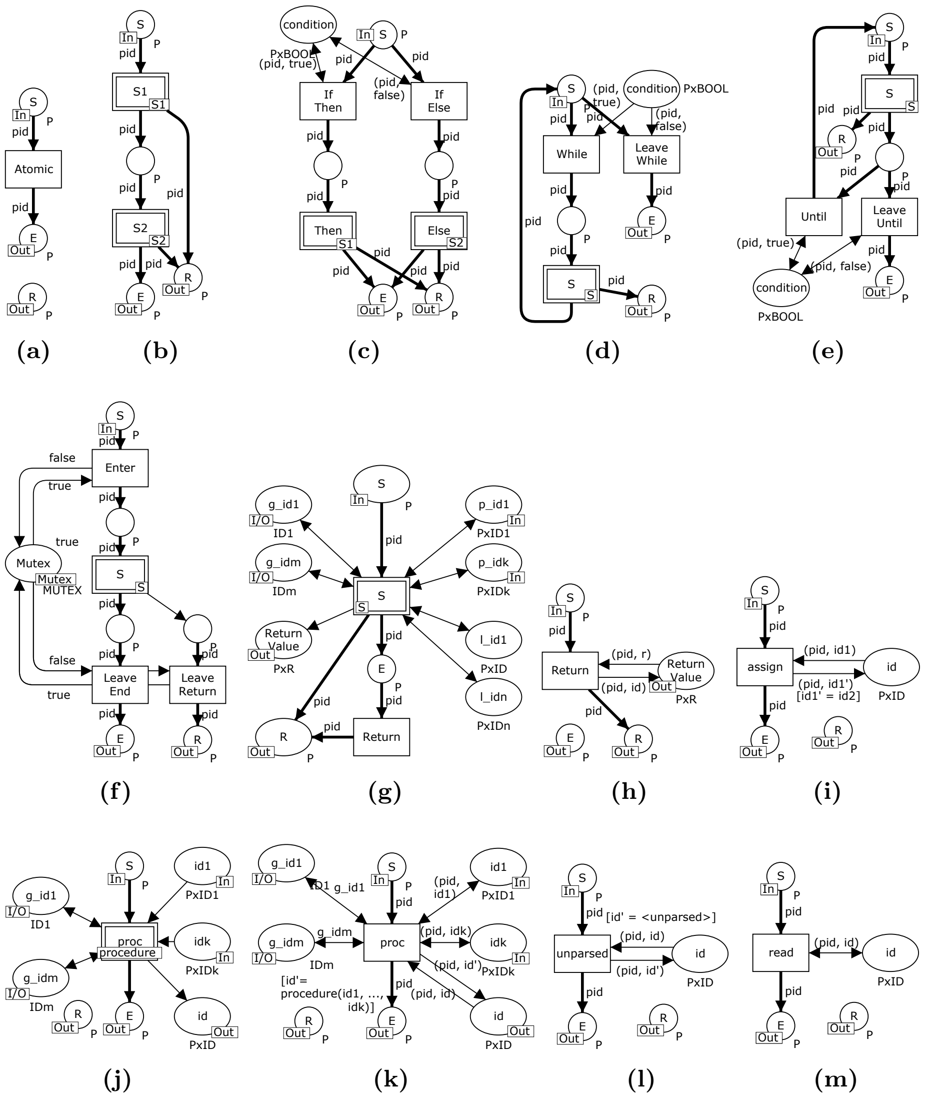


Fig. 6. Patterns for control structures

prime. The first is used for reading old values and the primed version is used for updating with new values. The following fragments are all described as if they are using local variables; they can all use global variables, in which case the process id component is removed. Updating values is handled by fragment (i). Procedure invocation is handled by fragments (j) and (k). (j) handles the case where a procedure is defined in the program and (k) when it is not (i.e., constitute a library function or a function whose action we abstract away). Invocation in both cases assigns values (id1. . . id$k$) for each of the parameters of the procedure, and binds the return value to the correct return value (id). We note that procedure fragments only have one exit for processes, so we only connect the E place to the invocation, leaving the R place unconnected (so a return only returns from the inner-most procedure). Pattern (l) and (m) are used for abstraction refinement and are introduced in the next subsection.

Applying this extraction to the *computeStateSpace* procedure of Fig. 4, we obtain Fig. 7 (left). The model has been manually laid out, we have simplified the generated names for legibility, and we have hidden the process id flowing along the bold arcs. The details are not important and the model is provided only to show the unreduced output of the translation. We translate procedures without use of hierarchy in our implementation, so all places and transitions pertaining to a single procedure are contained in a single page (but we have a page for each procedure). We can see the two loops, one from the branch-off around the middle of the long chain and back to the beginning and one from the bottom to the middle. We can see 3 branches, corresponding to the two loops and the if statement. We note that due to the program simplification, we get significantly more transitions than we have lines in the procedure. All transitions have a name from the line number used to generate it and a sequence number to ensure all transitions have unique names. The model in Fig. 7 (left) is good for getting a basic understanding of the control flow of the system, but impractical for analysis and detailed understanding as a lot of steps do not contribute to the behavior, especially the long lines of transitions not accessing any data and that unconditionally move from one state to the next. This also causes state-space explosion without providing benefits for analysis. It is therefore desirable to remove such transitions. We additionally merge lock releases with the last transitions of each procedure, also omitting an unconditional step (not applicable in this model). Doing this, we obtain the model in Fig. 7 (right). We can now see the 3 procedure calls as substitution transitions and global, local, and 4 generated variables as the marked places.

Abstraction Refinement. The initial abstraction allows execution of traces not allowed in the original program. In the model in Fig. 7 (right), it is possible to first terminate process 1 and have process 2 continue computation, i.e., execute:

// Process 1 enters and leaves *pickAndRemoveElement* in l7_10
l8_11⟨pid=computeStateSpace(1)⟩ $\rightarrow$

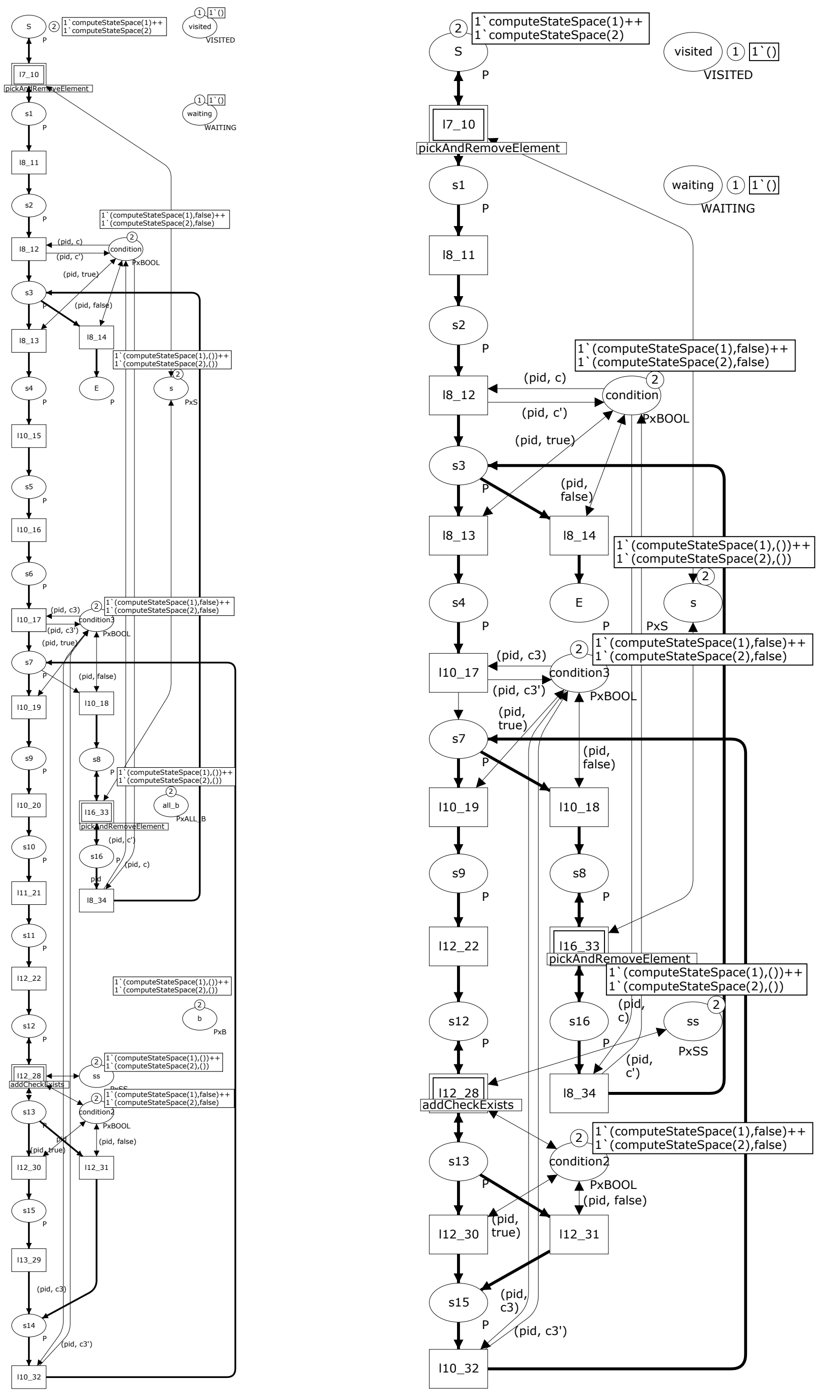


Fig. 7. Control-flow of model (left) and model after reduction (right)

l8_12⟨pid=computeStateSpace(1),c=false⟩ →
l8_14⟨pid=computeStateSpace(1)⟩ → // Terminate process 1
// Process 2 enters and leaves *pickAndRemoveElement* in l7_10
l8_11⟨pid=computeStateSpace(2)⟩ →
l8_12⟨pid=computeStateSpace(2),c=true⟩ →
l8_14⟨pid=computeStateSpace(2)⟩ → // Process 2 continues

This is not possible in Fig. 4. The model does find all possible interleavings of the process, though, so if the state-space does not contain any erroneous states, neither will the program.

Abstraction refinement consists of using more elaborate types on local and shared places and of applying pattern (m) of Fig. 6 to bind identifiers instead of non-determinism. Refinements must limit the behavior of previous models (which formally must be able to simulate any refinement if we ignore local and shared places), and must be true to the original program. If we add an explicit type to a variable in our program, our tool automatically applies pattern (m) for variables in all expressions. Furthermore, our tool will change the type of the place corresponding to the declared type accordingly. We need to parse (parts of) expressions in order to find all places a variable is read. If ⟨*unparsed*⟩ content is used, it is inserted verbatim in the model without any processing using pattern (l) of Fig. 6. This is done for simplicity of implementation of our prototype and means that an expression using the ⟨*unparsed*⟩ production reads a variable, this is not detected.

Here we do a simple refinement to avoid the situation where one process can decide that Waiting is empty just to have the other immediately afterward decide it is not. For this, we refine the type of Waiting to a BOOL, indicating whether the Waiting set is empty or not, and refine the type of s to indicate whether *notFound* was returned from *isEmpty*. Refining s and Waiting we obtain the model in Fig. 8. At the left, we see the refined model for *computeStateSpace* with changes highlighted. Due to the size of the model, we have just zoomed in on the changed parts. We have manually added place-/transition-names corresponding to Fig. 7 (right) to process places and transitions. We have more transitions in the refined model as they no longer can be reduced away due to refinement. To the right, we see the generated code for *pickAndRemoveElement* (top) and *pickAndRemoveElement* (bottom). All global variables are shared among all pages, local variables on subpages correspond to the actual parameters on *computeStateSpace*, the return value is mapped as well, and so are the entry-/exit-points. We have two instances of *pickAndRemoveElement* with different entry-/exit-point mappings. All ports are of type I/O due to a technicality in our translation, but as the port type carries no semantics, this is no problem. We get the refinement by explicitly declaring the types of s and Waiting in lines 1, 6, and 23 in Fig. 4.

Refining a variable means the place corresponding to a variable is accessed and updated according to the program. The value of waiting is initially false (as we add the initial state in line 2 of Fig. 4) and we do not care about the initial value

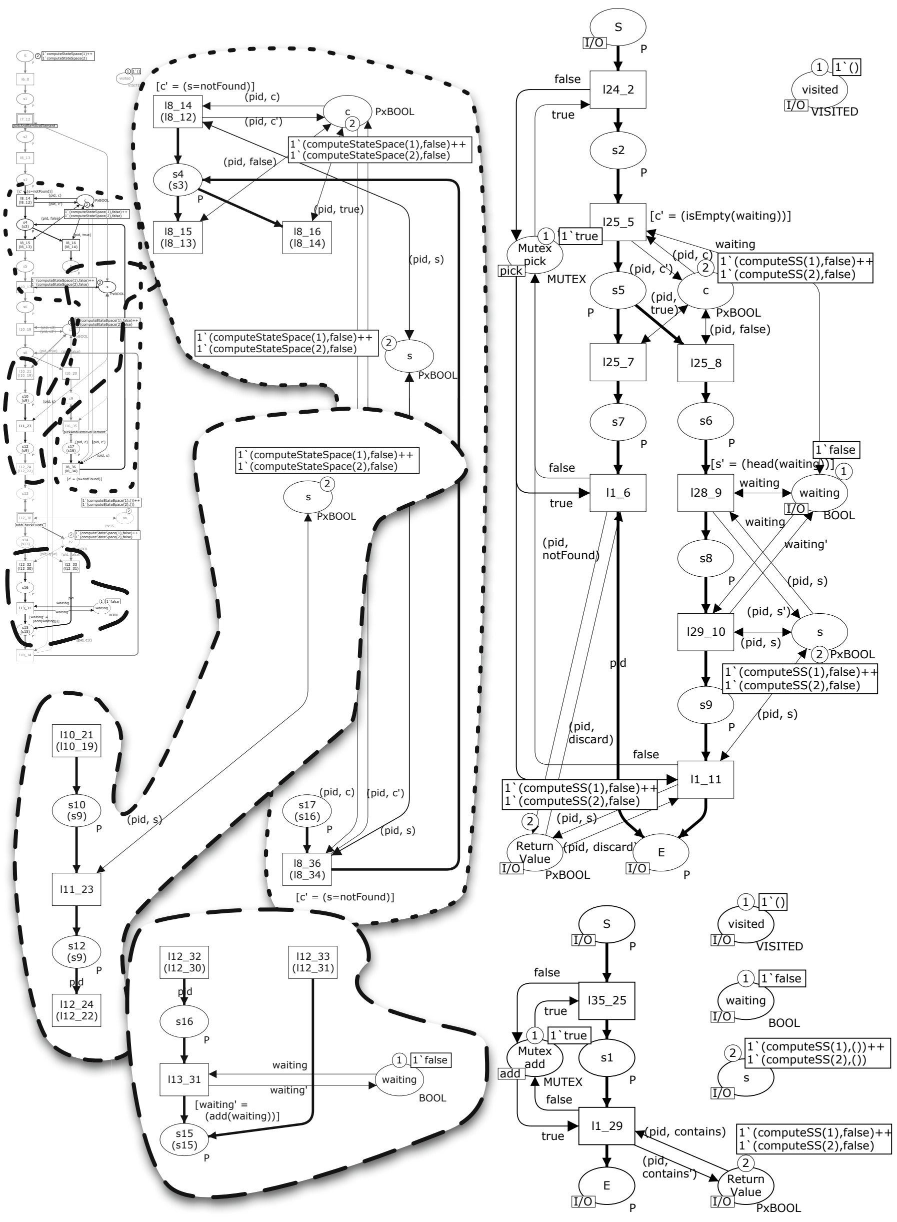


Fig. 8. Refined CPN model of state-space algorithm

of s, as it will be set before it is read. We make sure to read and update the values of waiting and s faithfully. Most of this happens in the *pickAndRemoveElement* fragment to the right in Fig. 8. We generate guards for updates, but it is up to the user to implement the functions used. In our example, we see a guard for the transition l25_5 setting c' based on the value of *isEmpty*(*waiting*). We implement the functions isEmpty, head (used in l28_9), and add (used in l13_31). isEmpty is the identity function (as the abstract value of waiting models whether the data-structure is empty). head returns the negation of its parameter (if the set is empty we cannot return a value, otherwise we can), and add always returns false (as adding an element always makes the data-structure non-empty). We have removed generated guards for l29_10 and l10_39 to make the result non-deterministic. We have explicitly added a reference to s in *computeStateSpace* transitions l8_14 and l8_36 as it is used as part of an unparsed expression (s = notFound). We can no longer execute the erroneous trace on this refined model.

4 Analysis

The main reason we started this work in the first place was to analyze parallel algorithms. Our focus is on new problems arising when creating parallel algorithms, not on proving correctness of serial algorithms. We therefore assume that algorithms are correct under certain mutual-exclusion assumptions, and search for such violations. We are also interested in potential dead- and live-locks. Assuming a valid refinement, we ensure absence of safety violations in the model guarantees absence in the real program as we can simulate all executions of the algorithm. This includes proving absence of mutual-exclusion violations. We cannot use our approach to ensure the absence of dead-locks, as we deal with over-approximations of the possible interleavings and further restricting the behavior may introduce new dead-locks, but we can still find dead-locks and remove them from the implementation.

We can do state-space analysis of the derived models from Figs. 7 and Fig. 8, and obtain state-spaces with 47,141 states for the original, unreduced model (Fig. 7 (left)), 18,139 states for the reduced model (Fig. 7 (right)), showing that reduction is important, and 15,934 states for the refined model (Fig. 8). We easily prove that the mutex property is not violated for the two critical regions.

Dead-locks and Live-locks. As all processes have a distinguished start and end-state, we can recognize dead-locks and live-locks in the model. A *dead-lock* is a state without successors (a *dead state*) where not all processes reside on E. We can find dead-locks in CPN Tools using the query:

```
1 fun terminal node =
2   Mark.computeStateSpace 'E 1 node = P.all()

4 List.filter (fn n => not (terminal n)) (ListDeadMarkings())
```

The function terminal tests whether the Marking of E on the page computeStateSpace (the top level) contains all tokens of P and line 4 removes all terminal states from the dead states (markings in CPN Tools terminology) and only returns non-terminal states. None of the models in Figs. 7 and Fig. 8 have dead-locks; the models in Fig. 7 have 64 dead states, where all process ids reside on E and the conditions have various values. The model in Fig. 8 has 48 dead states, where all process ids reside on E, Waiting is true, s is notFound for all processes, and conditions have various values, except for the ones controlling termination.

Live-locks are harder to recognize. We only consider live-locks in the absence of dead-locks. A model has a *strong live-lock* if the dead states of the model do not constitute a *home space*, i.e., if it is not always possible to reach one of the dead states. A strong live-lock in the model does not necessarily imply a live-lock in the original algorithm, but can be used to identify parts of the original program that should be further investigated. None of the models in Fig. 7 have strong live-locks. We can test this in CPN Tools using the query:

```
1  HomeSpace (ListDeadMarkings())
```

A model may have a *weak live-lock* if its state-space has a loop. A loop may also just indicate that a loop may execute an unbounded number of times. Both models in Fig. 7 and the refined one in Fig. 8 have loops. This is caused by the loops in the algorithm and are perfectly acceptable. A particular interesting kind of live-lock is a loop reachable from a state where E contains tokens. This means that even after one of the processes has terminated, the amount of work done by another process is unbounded. We have already seen that Fig. 7 exhibits this due to too abstract modeling, i.e., that process 1 may decide that Waiting is empty initially and terminate, just to have 2 decide it is non-empty and continue computation. We have seen this is not possible in the original algorithm, which caused us to refine the model to Fig. 8. We would therefore expect that no such live-lock was present in the refined model. Maybe surprisingly, one such does exist. We can find this by searching for all nodes where E contains tokens which are reachable from themselves via a non-trivial loop. In CPN Tools, we can find these using the query:

```
1   fun selfReachable node =
2   let
3     val nodes = OutNodes node
4     fun test [] = node
5       | test (n::rest) =
6       if n = node
7       then test rest
8       else if (SccReachable (n, node))
9            then n
10           else test rest
11  in
12    test nodes
13  end

15  fun predicate node =
16    if Mark.computeStateSpace'E 1 node = empty
17    then false
18    else selfReachable node <> node

20  PredAllNodes predicate
```

The function selfReachable computes all successors of the given node and for each of these, unless it is the node itself, test whether the given node is reachable from the successor. If the node is, the successor is returned, otherwise the original node is. The predicate tests if E at the top level is empty. If not, it tests if the node is non-trivially reachable from itself. Finally, we apply this predicate to the entire state space. We can use early termination by replacing line 20 by:

```
PredNodes (EntireGraph, predicate, 1)
```

We can replace the 1 by any number of examples we want. We can get a witness using:

```
ArcsInPath(1, List.hd (PredNodes (EntireGraph, predicate, 1)))
```

We can get the shortest path to such a node using:

```
fun shortest (node::nodes) =
let
  fun test path [] = path
    | test path (n::nn) =
    let
      val path' = ArcsInPath(1, n)
    in
      if List.length path > List.length path'
      then test path' nn
      else test path nn
    end
in
  test (ArcsInPath (1, node)) nodes
end

shortest (PredAllNodes predicate)
```

Here, the function shortest uses a helper function, test, which computes a shortest path to a node and compares with the current globally shortest path and returns the shortest one of the two. We can get a list of binding elements for analysis using:

```
List.map ArcToBE (shortest (PredAllNodes predicate))
```

For our example, we get that such a path is achieved by having one process enter *pickAndRemoveElement* and end up setting Waiting to true at l29_10 (which is the first time a choice occurs). Then, the process leaves *pickAndRemoveElement* and the other process starts. The just started process enters *pickAndRemoveElement*, discovers that Waiting is empty, leaves *pickAndRemoveElement* and terminates. Now, the first process continues, sets Waiting to false in l13_31, only stopping when it has performed all the work on its own.

This is also possible in the algorithm in Fig. 4, and even quite likely as the two processes will test Waiting initially, one of them will consume the only element it contains initially, and other processes terminate. This also occurred in reality in our first implementation.

To fix this, we notice that the reason one process terminates prematurely in the previous example is that it decided to terminate while the other can still add new states to Waiting. The idea of an improved algorithm is to ensure that no processes may terminate when others may produce new states. This prompts us to make the improvement seen in Fig. 9. We reuse *addCheckExists* and *pickAndRemoveElement* from Fig. 4, and define a new procedure for picking, *pickWithCounter* (ll. 27–36) which is used in place of the original *pickAndRemoveElement* (ll. 43 and 52). We use MayAdd as a counter of the number of processes which may add new states to Waiting. We add an additional loop around the previous main loop ensuring we only quit when MayAdd is zero.

We use the same approach to translate the program to a CPN model, using the same refinements for s and Waiting and no abstraction of MayAdd. We

```
 1: bool WAITING
 2: WAITING := MODEL.$initial()$
 3: VISITED := MODEL.$initial()$
 4: int MAYADD
 5: MAYADD := 0
 6: ...
27: proc $pickWithCounter()$ is
28:    bool $s$
29:    lock $withCounter$
30:       $s := pickAndRemoveElement()$
31:       if not $s = notFound$ then
32:          MAYADD := MAYADD + 1
33:       endif
34:       return $s$
35:    unlock
36: endproc
37:
38: $computeStateSpace()$ || 
             $computeStateSpace()$

40: proc $computeStateSpace()$ is
41:    bool $s$
42:    repeat
43:       $s := pickWithCounter()$
44:       while not $s = notFound$ do
45:          // Do any handling of $s$ here
46:          for all $b$ in $s.enabled()$ do
47:             $ss := s.execute(b)$
48:             if $addCheckExists(ss)$ then
49:                WAITING := WAITING.$add(ss)$
50:             endif
51:          endfor
52:          $s := pickWithCounter()$
53:          MAYADD := MAYADD − 1
54:       endwhile
55:    until MAYADD=0
56: endproc
```

Fig. 9. More involved state-space algorithm

obtain a model with a state-space with 143,372 nodes, 56 dead states, no dead-locks, and no live-locks. We have no weak live-locks reachable from a state where E contains tokens. Due to the translation being done automatically instead of manually, we actually found an error in the algorithm described in [17]. That algorithm is the same as the one in Fig. 9 except the one in [17] had lines 52 and 53 swapped, which allows one process to terminate while the other is between lines 52 and 53. In [17] we did not catch that due to the manual construction of the model, but with our automatic translation we did.

Verification of the two process case has given us confidence that the algorithm will work with any number of processes. We have also investigated an extended version additionally adding a check-point mechanism where all threads are paused while WAITING and VISITED are written to disk in a consistent configuration.

We also used the method to verify the implementation of a slightly simplified version of the protocol for operational support developed in [6,19]. The protocol supports a client which sends a request to an operational support service, which mediates contact to a number of operational support providers. The protocol developed in [6,19] has support for running all participants on separate machines, but we are satisfied with an implementation running the operational support server and providers on the same machine. We therefore have to send fewer messages, but need to access shared data on the server. We devised a fine-grained locking mechanism and used the method devised in this paper to prove that it enforces mutual exclusion and causes no dead-locks, increasing our confidence that the implementation works.

5 Conclusion and Future Work

We have given an approach for correct implementation of parallel algorithms. The approach allows users to extract a model from an algorithm written in an implementation or abstract language and verify correctness using state-space analysis. The approach also facilitates the generation of skeleton implementation code from the verified model using the approach from [10] as we rely on process-partitioned coloured Petri nets. Finally, we can also combine the two approaches, which facilitates writing an algorithm in an abstract language, extract a model for verification, and then extract a skeleton implementation. We have implemented a translation from a simple language to CPN models. The power of having an implementation is seen by the fact that we caught an error in the algorithm published earlier in [17] which erroneously had two lines swapped.

Verification of software by means of models is not new. Code-generation from models have been used in numerous projects. The approach has been most successful for generating specification of hardware from low-level Petri nets and other formalisms to synthesize hardware such as computer chips [7, 8]. The approach has also been applied to high-level Petri nets to generate lower level controllers [9] and more general software [10]. Model extraction was pioneered by FeaVer [11], which made it possible to extract PROMELA models from C code using user-provided abstractions, and Java PathFinder [13] which did the same for Java programs. The approach has successfully been refined using counter-example guided abstraction refinement (CEGAR) [16] which was first implemented by Microsoft SLAM [14], which extracts and automatically refines abstractions from C code for Microsoft Windows device drivers, and refined by BLAST [15]. While the tools for model-extraction support a full development cycle by abstraction refinement and reuse for modified implementations, the idea of combining the two approaches is to the best of our knowledge new. The combination allows some interesting perspectives. The perspective we have focused on in this paper is the ability to write an algorithm in pseudo-code, extract a model from the code, and generate an implementation in a real language. Another perspective is supporting a full cycle as well, where we extract a model from a program, find and fix an error in the model, and emit code that is merged with the original code, supporting a cycle where we do not need to fix problems in the original code but can do so at the model level. The use of coloured Petri nets instead of a low-level formalism allows us to use the real data-types used in the program instead of abstractions, much like how FeaVer allows using C code as part of PROMELA models, but with the added bonus that the operations are a true part of the modeling language rather than an extension that requires some trickery to handle correctly.

The work presented here is far from done. While our prototype allows us to verify simple but non-trivial examples, it cannot cope with more complicated systems. The main problem is that the state-space is growing very large, even just for the example seen in Fig. 9 with 2 processes (which has a state-space with 143,372 states). We can alleviate this by more reduction rules and more sophisticated translations. For example, we currently only consider transitions on single pages when removing unconditional paths. Also, updates to local variables are done concurrently, which is sub-optimal. We can alleviate this by using partial order reduction [20, 21] or by using transition priorities and giving such transitions high priority. Manually giving all transitions accessing only local data high priority reduces the state-space of the resulting model to 49,162 states.

We would also like to make the implementation more intelligent with respect to how types are handled. Currently, only directly declared types are taken into account. This means the prototype incorrectly generates an unrefined type on Return Value of *addCheckExists* in Fig. 8 even though we can see it is used as a boolean in line 12 of Fig. 4. It would be nice to use type unification to avoid this. It would also be interesting to make more advanced type analysis, propagating refinements, so we only have to refine s either in line 6 or in line 23 of Fig. 4. This would also allow us to not automatically refine the conditions of all conditionals, further simplifying generated models. We also want to improve how complicated expressions are parsed, i.e., reduce how often the $\langle$*unparsed*$\rangle$ production is used. This would remove some manual labor (like adding arcs to places for l8_14 and l8_36 in Fig. 8 due to line 8 of Fig. 4 containing unparsed content). More importantly, more intelligent parsing would allow us to do simple abstraction automatically, only asking users when unknown functions are called. In the longer term, it would be interesting to assist users with this using replay on the original program.

Our current method focuses on parallel algorithms with a fixed number of identical processes, but there is nothing in our approach preventing us from extending this to also handle distributed settings with asynchronous communication using buffer places and different kinds of processes; the code generation in [10] even supports that out of the box. While the fixed number of processes used in this paper works well for simple algorithms, more advanced algorithms may need to spawn processes. Nothing in our approach inherently forbids this, but the code generation in [10] does not support this out of the box. We believe that it should be quite easy to devise a construction for starting new processes and adapt the code generation to handle this.

Our current approach does not support recursive (or mutually recursive) procedures, as we create a static sub-page for each procedure call. This is done for simplicity in the prototype, but could easily be changed to explicit hand-over of control among several top-level pages. This is similar to dynamic process instantiation and interprocess communication and can be implemented by adding to each procedure setup and teardown transitions initializing and removing local variables, and maintaining a mapping between process identifiers.

References

1. Billington, J., Wilbur-Ham, M., Bearman, M.: Automated protocol Verification. In: Proc. of IFIP WG 6.1 5th International Workshop on Protocol Specification, Testing, and Verification, pp. 59–70. Elsevier (1985)
2. Kristensen, L.M., Jørgensen, J.B., Jensen, K.: Application of Coloured Petri Nets in System Development. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 626–685. Springer, Heidelberg (2004)

3. Kristensen, L.M., Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 248–269. Springer, Heidelberg (2004)
4. Espensen, K., Kjeldsen, M., Kristensen, L.: Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 152–170. Springer, Heidelberg (2008)
5. Jensen, K., Kristensen, L.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
6. Westergaard, M., Maggi, F.M.: Modeling and Verification of a Protocol for Operational Support Using Coloured Petri Nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 169–188. Springer, Heidelberg (2011)
7. Yakovlev, A., Gomes, L., Lavagno, L.: Hardware Design and Petri Nets. Kluwer Academic Publishers (2000)
8. IEEE Standard System C Language Reference Manual. IEEE-1666
9. Rasmussen, J.L., Singh, M.: Designing a Security System by Means of Coloured Petri Nets. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 400–419. Springer, Heidelberg (1996)
10. Kristensen, L.M., Westergaard, M.: Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 215–230. Springer, Heidelberg (2010)
11. The FeaVer Feature Verification System webpage, `http://cm.bell-labs.com/cm/cs/what/feaver/`
12. Holzmann, G.: The SPIN Model Checker. Addison-Wesley (2003)
13. Havelund, K., Presburger, T.: Model Checking Java Programs Using Java PathFinder. STTT 2(4), 366–381 (2000)
14. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: Proc. of POPL, pp. 1–3. ACM Press (2002)
15. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The Software Model Checker BLAST: Applications to Software Engineering. STTT 7(5), 505–525 (2007)
16. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50, 752–794 (2003)
17. Westergaard, M.: Towards Verifying Parallel Algorithms and Programs using Coloured Petri Nets. In: Proc. of PNSE. CEUR Workshop Proceedings, vol. 723, pp. 57–71. CEUR-WS.org (2011)
18. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
19. Westergaard, M., Maggi, F.: Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. Submitted to Fundamenta Informaticae
20. Peled, D.: All from One, One for All: On Model Checking Using Representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
21. Valmari, A.: Stubborn Sets for Reduced State Space Generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)