Towards 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 modeling parallel algorithms and programs, i.e., algorithms and programs where processes communicate via shared memory. The model is verified for correctness, here to prove absence of mutual exclusion violations and to find dead- and live-locks. The approach can be used in a model-driven development approach, where code is generated from a model, in a model-extraction approach, where a model is extracted from running code, or using a combination of the two, 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.

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, 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 [3, 5, 11, 12]. 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 becomes interesting. In this paper, we present our experiences developing parallel algorithms with synchronization mechanisms developed and verified by means of *coloured Petri nets* (CPNs) [10]. This work was motivated by the requirement 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 example, but the approach has also been used for other parallel algorithms, such as parts of a protocol for operational support [16].

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 model which is verified according to one or more properties. If it is ok, we can extract a program, and otherwise we refine the model. Examples of this approach are within hardware synthesis [9,17], using a CPN simulator to drive a security system [14], or general code generation from a restricted class of CPNs [13]. 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. If an error is found, the resulting error-trace is replayed on the original program to determine if it can be reproduced here. 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 [6] to translate C code to PROMELA code usable in SPIN [8], Java PathFinder [7] to translate Java programs to PROMELA, SLAM [1] for automatically translating C device drivers to boolean programs, BLAST [2] for model-checking C programs, and many other tools. Both of these approaches may terminate the loop without providing a definite response.

The model-driven software engineering and model-extraction approaches have different strengths and weaknesses. The main strength of the model-driven soft-

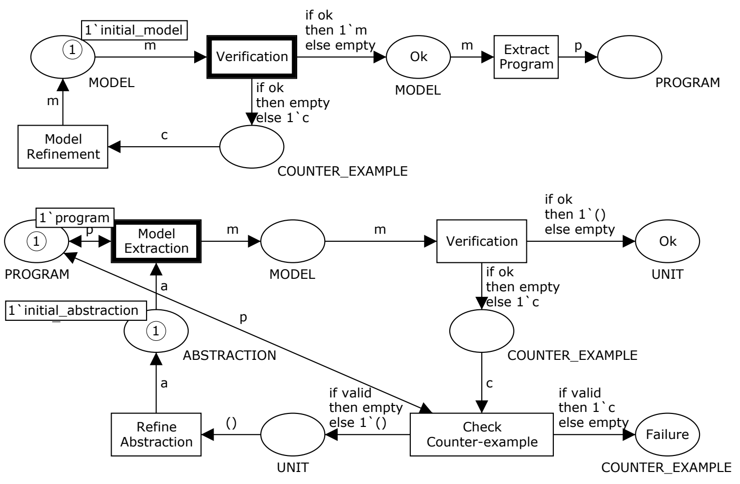

Fig. 1: Model-driven software engineering (top) and model-extraction (bottom).

ware 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. By supporting model-extraction in a way that allows subsequent code generation, we can support both approaches plus a new merged approach, shown in Fig. 2. Here we extract a model from a description in one – possibly abstract – language, manually or automatically refine the abstraction until we can prove the system to be correct (optionally modifying the original description if we find errors), and then extract an implementation. While the last step seems superfluous if we already have an implementation, it can be useful, for example, to use a program written in pseudo-code as input, automatically derive and verify a model, and from the model extract a runnable program in a desired implementation language. Alternatively, we can do round-trip engineering, where we take an implementation as input, verify and correct it on the level of a model, and update the original implementation to reflect the changes.

To do this, we use systematic extraction and abstraction methods that can be derived from a program or from an algorithm written in pseudo code. Our approach builds on *process-partitioned coloured Petri nets* (PP-CPNs) as described in [13]. 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. The restricted class of PP-CPNs allows us to automatically generate executable code from the model. Derivation of abstractions of the data-types used can be done by the user or automatically using counter-example guided abstraction refinement (CEGAR) [4] as implemented in SLAM [1] and BLAST [2]. CEGAR automatically improves abstractions by

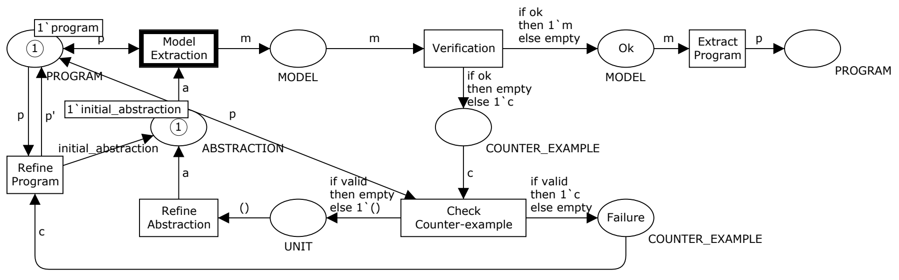

Fig. 2: Our approach combining model-driven software engineering and model-extraction.

replaying errors found in an abstract model on the original program and using about why a given error-trace cannot be replayed in the original program to refine the abstraction.

In this paper we focus on model-extraction. Our goal is to provide a proof-of-concept, so we do certain steps that can be automated by hand, such as the translation from code to a model using patterns. We do not address refinement after discovery of errors in this paper, but assume an external library using CEGAR or a user takes care of that. We have already treated the code generation aspect in [13].

The rest of this paper is structured as follows: in the next section, we introduce process-partitioned coloured Petri nets as defined in [13] and a simple algorithm for state-space generation which we use as 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 use state-space generation to identify a problem in the original parallelization, fix the problem and show that the problem no longer is present in a modified version. Finally, in Sect. 5, we sum up our conclusions and provide directions for future work.

2 Background

In this section, we briefly introduce process-partitioned CPNs as defined in [13]. We also give a simple algorithm for explicit state-space generation which we use as example in the remainder of the paper.

Process-partitioned CPNs. Coloured Petri nets (CPNs) consist of *places*, *transitions*, and *arcs*. Places are typed and arcs have expressions that may contain variables. Places may contain tokens and we call the distribution of tokens on all places a *marking* of the net, and the marking before executing any transitions the *initial marking*. CPNs have a module concept, where *subpages* are represented by *substitution transitions*.

In [13] we introduce the notion of *process-partitioned CPNs* (PP-CPNs). These 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 [13]). A single process subnet is a PP-CPN, but not necessarily the other way around, but 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 process identifier. The model in Fig. 6 (left) is an example of a PP-CPN (we provide a detailed description of the model in Sect. 3). The process colour set of this model is P. The places of a process subnet are partitioned into *process places*, *local places*, and *shared places* (in [13], we additionally introduce *buffer places* for asynchronous communication between processes, but these are not used here). These places correspond to the control flow, the local variables, and shared variables of normal programs. Process places must have the process colour set as type (in the example S and E and all unnamed places are process places), local places must have a product of the process colour set and any other type as type (in the example, s, b, and s'), and shared places can have any type (in the example, Waiting and Visited).

In the initial marking, exactly one of the process places must contain all tokens of the process colour set and the remaining process places must be empty (modeling that all processes must start in the same location in the program). Local places must 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). All shared places must contain exactly one value (modeling that shared variables must be initialized). All arc expression must ensure that tokens are preserved.

We have chosen to adopt the notion that we cannot create new processes or destroy processes from [13] 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.

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 marking 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.

```
 1: WAITING ← {MODEL.initial()}
 2: VISITED ← {MODEL.initial()}
 3: while WAITING ≠ ∅ do
 4:    Pick a s ∈ WAITING
 5:    WAITING ← WAITING \ {s}
 6:    // Do any handling of s here
 7:    for all b enabled in s do
 8:       Execute b in s to get s′
 9:       if s′ ∉ VISITED then
10:          WAITING ← WAITING∪{s′}
11:          VISITED ← VISITED ∪ {s′}
```

Fig. 3: Simple 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 last step, going from a PP-CPN model to implementation code, is handled in [13].

A simple way to parallelize Fig. 3 is shown in Fig. 4 (left). Here, we initialize as before (ll. 1–2). We have moved the main loop to a separate procedure, *computeStateSpace*. We perform mostly the same loop as before (ll. 16–24), but instead of testing for emptiness and picking an element of the queue in three steps, we do so using a procedure *pickAndRemoveElement* (ll. 17 and 24). The implementation of *pickAndRemoveElement* (ll. 4–9) does the same as we did before, except we return a bottom element $\perp$ if no elements are available and use that in the condition of the loop (l. 18). This forces us to perform the pick in two places: before the first invocation of the loop (l. 17) and at the end of the loop (l. 24). Handling of states (l. 19) and iteration over all enabled bindings (ll. 20–21) 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. 22 and 11–14). 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. 23). The reason for this re-organization is that we now assume that *pickAndRemoveElement*, *addCheckExists*, and the access to WAITING in line 23 are atomic, e.g., by creating a data-structure ensuring this or by requesting a lock for each data-structure before the start of an operation and releasing it afterward. This allows us to start two instances of *computeStateSpace* in parallel in lines 26–27. We will not argue for the correctness of neither Fig. 3 nor Fig. 4 (left), 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.

Model Extraction. To go from Fig. 4 (left) 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 the data until we can prove the properties of the model we want.

Extracting the control-flow consists of creating the process places and transitions of the model. We do that using templates, very similar to the workflow-patterns [15] for low-level Petri nets. In Fig. 5 we show the patterns necessary to translate programs using our simple pseudo-code language to a PP-CPN model. From left to right the patterns match an atomic action (*Atomic*), a sequence of two subprograms ($S1; S2$), a conditional branch (**if** *condition* **then** $S1$ **else** $S2$), a while loop (**while** *condition* **do** S), and a critical section (**atomic** S). The type P is the process colour set and for each pattern, the place S is the start place and E the end place. All places created are process places except for the Mutex place, which is a shared place. We put all processes on the start place of the top level. We can add new templates or derive other constructs, such as a simplified conditional branch omitting the else path, a repeat/until loop, a for loop, and a for all loop.

In the initial abstraction, we translate a condition to an unbound boolean variable in the PP-CPN model. We add a local place for each local variable and a shared place for each global variable. These are available on all subpages where they are within the scope. In the initial abstraction, we approximate the type

```
1: WAITING ← {MODEL.$initial()$}
2: VISITED ← {MODEL.$initial()$}
3:
4: proc $pickAndRemoveElement()$ is
5:   if WAITING $= \emptyset$ then
6:     return $\perp$
7:   Pick a $s \in$ WAITING
8:   WAITING ← WAITING $\setminus \{s\}$
9:   return $s$
10:
11: proc $addCheckExists(s')$ is
12:   $result \leftarrow s' \notin$ VISITED
13:   VISITED ← VISITED $\cup \{s'\}$
14:   return $result$
15:
16: proc $computeStateSpace()$ is
17:   $s \leftarrow pickAndRemoveElement()$
18:   while $s \neq \perp$ do
19:     // Handle $s$ here
20:     for all $b$ enabled in $s$ do
21:       Execute $b$ in $s$ to get $s'$
22:       if $addCheckExists(s')$ then
23:         WAITING ← WAITING $\cup \{s'\}$
24:     $s \leftarrow pickAndRemoveElement()$
25:
26: $computeStateSpace()$
27:         || $computeStateSpace()$
```

```
1: WAITING ← {MODEL.$initial()$}
2: VISITED ← {MODEL.$initial()$}
3: MAYADD ← 0
4:
5: proc $pickWithCounter()$ is
6:   $s \leftarrow pickAndRemoveElement()$
7:   if $s \neq \perp$ then
8:     MAYADD ← MAYADD + 1
9:   return $s$
10:
11: proc $computeStateSpace()$ is
12:   repeat
13:     $s \leftarrow pickWithCounter()$
14:     while $s \neq \perp$ do
15:       // Do any handling of $s$ here
16:       for all $b$ enabled in $s$ do
17:         Execute $b$ in $s$ to get $s'$
18:         if $addCheckExists(s')$ then
19:           WAITING ← WAITING $\cup$
              $\{s'\}$
20:       MAYADD ← MAYADD − 1
21:       $s \leftarrow pickWithCounter()$
22:   until MAYADD=0
23:
24: $computeStateSpace()$
25:         || $computeStateSpace()$
```

Fig. 4: Naive parallel state-space algorithm (left) and more involved algorithm (right).

of all variables with UNIT, and local and shared places are not connected to transitions.

Applying this extraction to the *computeStateSpace* procedure of Fig. 4 (left) and flattening it, we obtain Fig. 6 (left). Transitions are named after the line number they correspond to and conditions after the performed tests. Assign To Value stems from expanding the for all loop in line 18 to a while loop and line 19 has been omitted. We have instantiated the process twice (initial marking of S).

Abstraction Refinement. The initial abstraction allows execution of traces not allowed in the original program. In the model in Fig. 6 (left), it is possible to first terminate p(1) and have p(2) continue computation. This is not possible in Fig. 4 (left). 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, of adding arcs for reading and updating local and shared places,

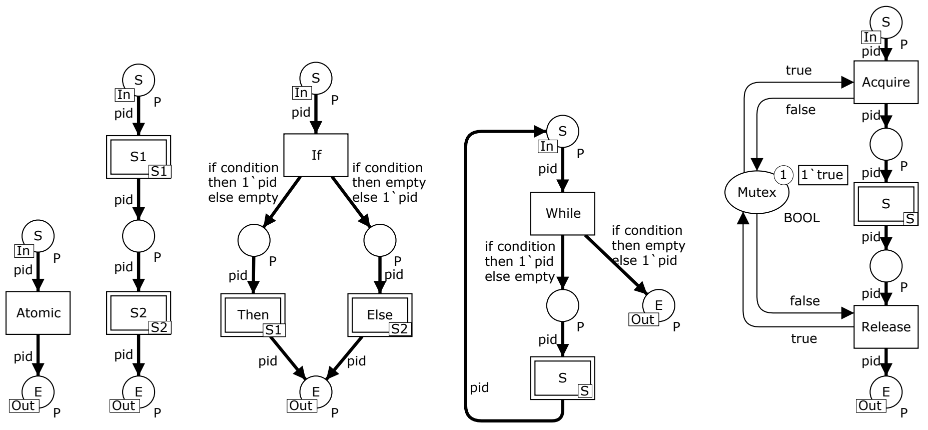

Fig. 5: Patterns for control structures.

and of limiting the values of condition variables, often using values from the local or shared places. 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.

The basic idea is to refine the types of places modeling local and global variables and add tests accordingly. When we cannot determine an exact test or update, we solve it by non-deterministically choosing among the possible values. Here we only 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 $\perp$ was returned from *isEmpty*. The value of Waiting is initially false (as we add the initial state in line 1 of Fig. 4 (left)) and we do not care about the initial value 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. Adding an element to Waiting (l. 23) implies setting the value to true and for picking elements (ll. 17 and 24), we read the previous value of Waiting, use that to set the value of s, and non-deterministically set the value of the waiting set to true or false (though if the value already was true it remains so) as we do not have enough information to know which is the correct answer. We obtain the model in Fig. 6 (right). For readability, we have merged the transitions 17 and 24, but this is not necessary for analysis. We no longer can execute the erroneous trace on this refined model.

4 Analysis

The main reason we started this work in the first place is 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

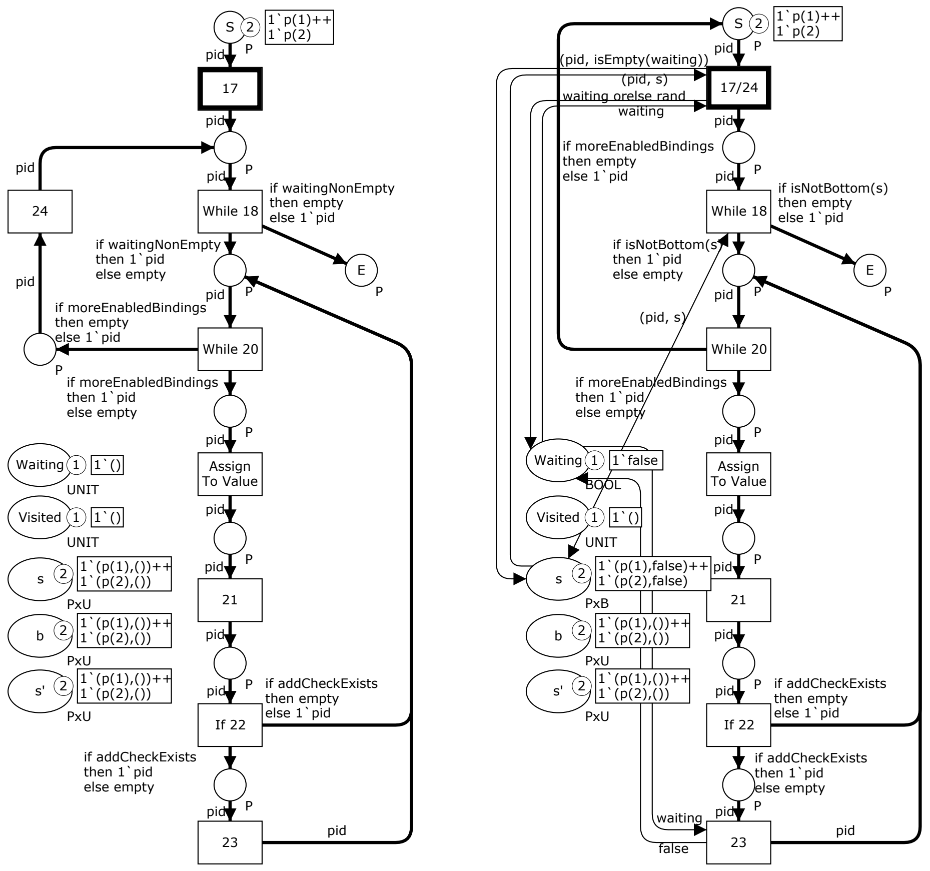

Fig. 6: Control-flow of (left) and a simple refinement of the model (right).

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 can ensure that 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 Fig. 6, and obtain a state-space with 81 states for the abstract model (left) and a state-space with 130 states for the refined model. As the models do not have any critical regions, they of course have no mutual exclusion violations.

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. Neither of the models in Fig. 6 has dead-locks; the model to the left has exactly one dead state, where all process ids reside on E and the shared and local places retain their initial marking. The model in Fig. 6 (right) also has one dead state, where all process ids reside on E, Waiting is true, s is true for all processes, and all remaining local and shared places have their initial value.

Live-locks are a 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. 6 have strong live-locks.

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. 6 have loops, but analysis shows that the transition While 20 is *impartial*, i.e., that in any infinite execution it occurs an infinite number of times. This happens if we compute infinitely many successors (the state-space has infinitely many states), and makes sense in our algorithm.

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 have terminated, the amount of work done by another process is unbounded. We have already seen that Fig. 6 (left) exhibits this due to too abstract modeling, i.e., that process p(1) may decide that Waiting is empty initially and terminate, just to have p(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. 6 (right). We would therefore expect that no such live-lock was present in the refined model. Maybe surprisingly, one such does exist. This is seen by having p(1) check Waiting in 17/24, modify Waiting to be empty, and successfully continue. Then, p(2) checks Waiting, notices it is empty and terminates. Now, p(1) continues. This is also possible in Algorithm 4 (left), 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 of a parallel state-space exploration algorithm using Algorithm 4 (left).

To fix this, we notice that the reason p(2) terminates prematurely in the previous example is that it decided to terminate while p(1) 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 Algorithm 4 (right). We reuse *calculateStateSpace*, *addCheckExists*, and *pickAndRemoveElement* from Algorithm 4 (left) and define a new procedure for picking, *pickWithCounter* (ll. 5–9) which is used in place of the original *pickAndRemoveElement* (ll. 13 and 21). 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 inline the call to *pickWithCounter* for the translation and use a mutex around the call to ensure atomicity.

We use the same approach to translate the model to a CPN model. We maintain the abstraction of WAITING and do no abstraction of MAYADD, i.e., we increment and decrement it according to the algorithm. We implement the mutex around the call to *pickWithCounter* as a place with type UNIT containing a single token acting as the mutex. We then obtain the model in Fig. 7. We have not completely flattened it for readability and to keep resemblance with the previous models in Fig. 6. At the left we have the loop in lines 13–21, which mostly corresponds to Fig. 6 (right). The only changes are that aside from renaming transitions to correspond with the line numbers of Algorithm 4 (right), we have added a transition for the new line 20 causing some rerouting of the flow, added a place representing MayAdd, and changed 13/21 (named 17/24 in Fig. 6) to a substitution transition. We have modeled the new outer loop as a separate page shown in Fig. 7 (top/right). We implement the semantics of a repeat/until loop looping over the page in Fig. 7 (left). We share access to MayAdd. The Test transition is explained later. The page corresponding to the substitution transition modeling *pickWithCounter* is shown in Fig. 7 (bottom/right). Again, we share access to a local place (s) and two shared places (MayAdd and Waiting). In all cases, the S and E places of a subpage is in port/socket relationship with the input and output place of the corresponding substitution transition.

We can use state-space analysis to verify that (for this configuration) we do not violate the mutex property in *pickWithCounter*. The state-space for this model contains 399 states and 934 bindings. We can check this explicitly by looking at the total number of tokens on the four unnamed places in Fig. 7 (bottom/right). This is either 0 or 1, showing that never do we have more than one process inside *pickWithCounter*. The Test transition in Fig. 7 is added to easily test whether it is ever possible for a process to execute lines 15–20 while another process has terminated, i.e., whether it is possible for a process to terminate while there is still work to do. The transition requires a token from E, i.e., a process that terminated and that the value of MayAdd is non-zero (this is handled by the *guard* [n<>0], which has not been explained but exactly ensures this). If Test is enabled in any state, it means that a process has terminated while another has more work to do. State-space analysis shows this is not the case. This is also a safety property, so absence of violations in the model implies absence of violations in the original algorithm. We can prove the property without adding Test by searching for a state where E has tokens and Waiting is false or one of the unnamed places below While 14 contains tokens.

We can convince ourselves that the mutex around *pickWithCounter* is necessary by removing it and repeating analysis. We then get a state where Test is enabled, and we can verify the same error is present in the original model (have one process enter *pickWithCounter* and consume the last element of Waiting,

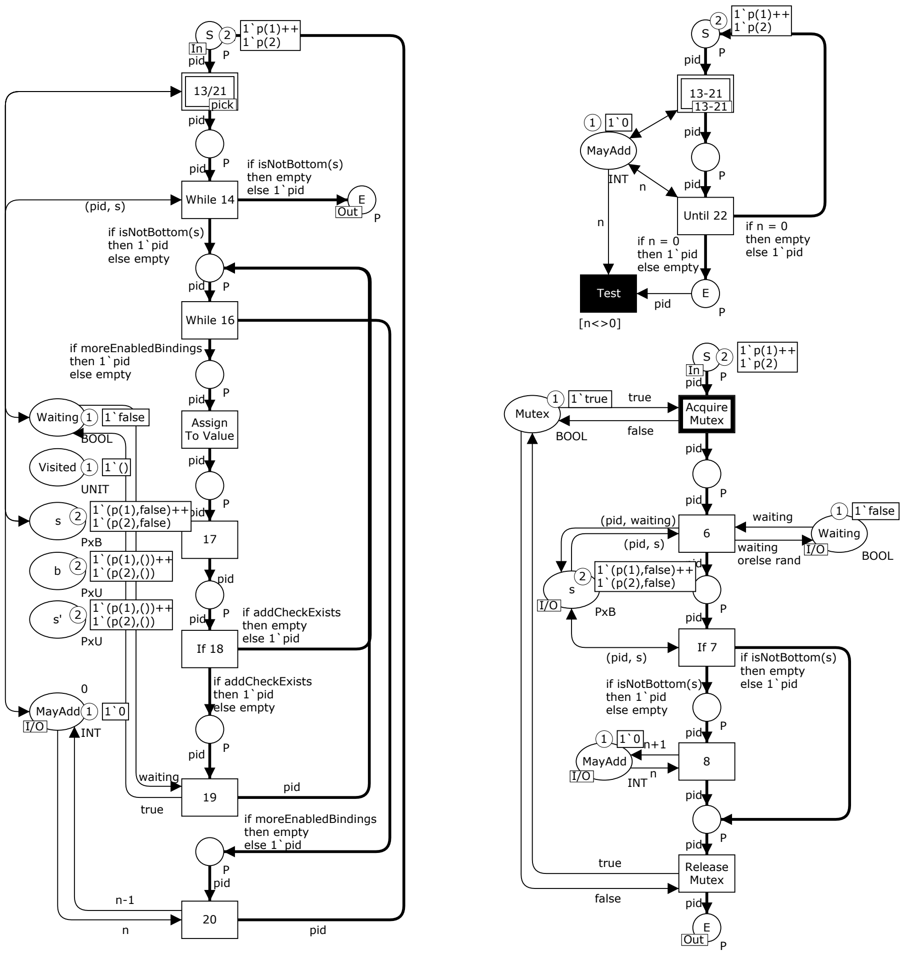

Fig. 7: CPN model of more elaborate state-space algorithm.

then have the other process enter *pickWithCounter*, notice there is no element and return ⊥, and terminate, just to have the first process continue alone).

In addition to the analysis of non-termination, we naturally also check for dead- and live-locks as before and find nothing unexpected. While Algorithm 4 (left) is quite simple and it is probably possible to convince oneself that it is correct without verification, the introduction of MayAdd makes Algorithm 4 (right) sufficiently complicated that correctness is not immediately apparent. We have verified the algorithm with up to 4 processes (yielding 55.709 states), which is the configuration we use in practice. Verification has given us confidence that the algorithm will work with any number of processes. We have also investigated an extended version additionally adding a checkpointing 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 [16]. 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 [16] has support for running all participants on separate machines, i.e., using asynchronous communication, but we are satisfied with an implementation running the operational support server and providers on the same server. We therefore have to send fewer messages, but need to access shared data on the server. We devised a fine-grained locking mechanism using the method devised in this paper and proved that it enforced mutual exclusion and well as caused no dead-locks, increasing our confidence that the implementation works.

5 Conclusion and Future Work

We have sketched 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 [13] 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.

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 [9, 17]. The approach has also been applied to high-level Petri nets to generate lower level controllers [14] and more general software [13]. Model extraction was pioneered by FeaVer [6], which made it possible to extract PROMELA models from C code using user-provided abstractions, and Java PathFinder [7] which did the same for Java programs. The approach has successfully been refined using counter-example guided abstraction refinement (CEGAR) [4] which was first implemented by Microsoft SLAM [1], which extracts and automatically refines abstractions from C code for Microsoft Windows device drivers, and refined by BLAST [2]. 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 on 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 only in the initial stages, but looks very promising. We have several ideas for future work. Currently, we have to manually extract the model from patterns. This is tedious and error-prone, and it would be nice to have automatic extraction. Such a translation should implement the patterns included in Fig. 5, but could also use explicit patterns for repeat/until loops and other constructs. Given an implementation of the translation to and from models, we could look at supporting a full development cycle allowing us to update existing code with changes to the model.

An implementation could also implement reduction rules like the one we used to merge lines 17 and 24 in the model in Fig. 6 (right). We can also add simplifications collapsing long traces of unconditional progress not modifying any data to reduce the state-space without removing behavior. For example, in Fig. 7 we can remove transitions Assign To Value and 17, merging the input place of Assign To Value and the output place of 17, to obtain a smaller state-space of 26.909 states for 4 processes, down from 55.709 states. We can also merge the acquisition of a lock with the first regular transition (merging the transitions Acquire Lock with 6 in Fig. 7 (bottom/right)) and merging releases with the last. This reduces the state-space to 14.841 states.

Currently, we provide abstractions manually. Like SLAM, we could easily replay found errors on the original code and provide assistance in the development of refinements, possibly even making them automatically. In our example, replaying the early termination error trace found in Fig. 6 (left) on Algorithm 4 (left) would show that it is not possible for *isEmpty* to return false initially and that it can only change from returning false to returning true if we execute line 23. Even though we might not be able to provide the abstraction refinement in Fig. 6 (right) fully automatically, providing such diagnostics can be very useful for the user for improving the refinement.

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 [13] 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 [13] 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.

One thing our approach does not support very well at the moment is intra-procedure calls. We can currently simulate this in simple cases by inlining procedure calls, but this is not possible when using recursion. One way to fix it is to view a recursive call as starting a new process for executing the child and waiting for the result. If we support different kinds of processes, communication between processes, and dynamic instantiation of processes, this should be easy to add.

References

1. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of POPL'02*, pages 1–3. ACM Press, 2002.
2. D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering. *STTT*, 7(5):505–525, 2007.
3. J. Billington, M.C. Wilbur-Ham, and M.Y. Bearman. Automated protocol Verification. In *Proc. of IFIP WG 6.1 5th International Workshop on Protocol Specification, Testing, and Verification*, pages 59–70. Elsevier, 1985.
4. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50:752–794, 2003.
5. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *Proc. of ATPN*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.
6. The FeaVer Feature Verification System webpage. Online: `cm.bell-labs.com/cm/cs/what/feaver/`.
7. K. Havelund and T. Presburger. Model Checking Java Programs Using Java PathFinder. *STTT*, 2(4):366–381, 2000.
8. G.J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
9. IEEE Standard System C Language Reference Manual. IEEE-1666.
10. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
11. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
12. L.M. Kristensen, J.B. Jørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In *Proc. of 4th Advanced Course on Petri Nets*, number 3098 in LNCS, pages 626–685. Springer, 2004.
13. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, LNCS, pages 215–230. Springer, 2010.
14. J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proc. ATPN'96*, volume 1091 of *LNCS*, pages 400–419. Springer, 1996.
15. W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
16. M. Westergaard and F.M. Maggi. Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In *Proc. of ATPN*, LNCS. Springer, 2011.
17. A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets*. Kluwer Academic Publishers, 2000.