Proving liveness properties of concurrent programs using petri-nets

Fredrik Loch

Fredrik Loch
VT 2014
Bachelors thesis, 15 Credits
Supervisor: Francisco Hernández
Examiner: Pedher Johansson
Bachelor program in Computer Science, 180 Credits
Abstract

With the increased scale of distributed computations the complexity of liveness proofs have increased. In this paper we endeavor to simplify the process of verifying a concurrent system using well know modeling techniques. The choice of modeling tool as well as the proof is based on future scalability and automation. We translate the formal proof to a petri-net representation and use this to verify basic algorithms.

We show that the formal proof of liveness stated by Owiki and Lamport can be adapted to petri-nets. We also show a modification to petri-nets for increased granularity in loop modeling. This is used to clarify the translation of the original proof to our petri-net representation. With these results we discuss the usefulness of our approach and compare it to other methods of ensuring liveness.
Acknowledgements

I would like to thank my supervisors Francisco and Pedher, that by both encouragement and advice on the theoretical subjects as well as document layout made the work possible. I am also grateful for the support of my fiance, my family and friends that have kept me motivated during the work.
# Contents

1 Introduction 1
   1.1 Distributed systems and correctness properties 2
   1.2 Alternative approaches to proving liveness 3

2 Preliminaries 5
   2.1 The programming language used to describe systems 5
   2.2 Introduction to basic temporal logic 5
   2.3 The formal proof of liveness 6
   2.4 Introduction to the use of proof-lattices 7
   2.5 Definition and usage of petri-nets 8
   2.6 Summary of the choice in modeling language 9

3 Translations and verification of the petri-net representation 11
   3.1 Translation of formal rules to petri-nets 14
   3.2 Verifying petri-net translations against the formal proof 17

4 Discussion 25
   4.1 Automation 25
   4.2 Large scale systems 25
   4.3 Benefits and limitations 26
   4.4 Further work 26

5 Bibliography 27
1 Introduction

A distributed system is a network of computational nodes that provides a service or performs calculations. These nodes are coordinated using message passing, usually distributed systems are characterized by the fact that its components execute concurrently, it lacks a global clock and that it’s components can fail independently [8].

To ensure that a distributed system functions as intended it needs to satisfy certain correctness properties. These properties ensures computational safety as well as liveness. The liveness property ensures that a system reaches a desired state at some point of execution. Violations against these properties could place a system in a illegal state or interrupt execution. Due to the asynchronous and dynamic nature of distributed computations ensuring these properties can be a major challenge. This is especially true as the scale and complexity of the systems increase. The scale and the complexity of the system also increases the consequence of concurrency failures. Since the nodes can fail independently a failure could lead to cascade failures.

To minimize execution costs due to concurrency failures service providers and developers can use different techniques to ensure correctness. For this paper we will adapt one of these methods of ensuring correctness to a modeling language. This is done with the intent to create an algorithmic process for verifying these properties, this process could then be automated. An automated method for verifying liveness would have major benefits for both development of new systems and optimization of existing infrastructure.

Due to it’s algorithmic format we will use a formal proof of liveness described by Owicki and Lamport in [15]. This proof is based on temporal logic and uses the assumptions of safety and fair scheduling to prove liveness. This is accomplished by the description of a set of rules that can be proved to work with the given assumptions. These rules can then be applied to an algorithm to prove that all available paths lead to a desired state. With the rules from the proof we create a translation to a petri-net representation. Petri-nets is a modeling language that focuses on modeling concurrent operations. These representations will then be verified against the formal proof. This will require us to adapt the definitions of liveness in petri-nets to comply with the definitions of liveness for a generic distributed system. To prove the concept of proving liveness using petri-nets models we will use the translation of the proof to verify that a basic system upholds liveness.

After proving this concept we will discuss the usability of the approach compared to other methods of ensuring liveness. Following this there will be a brief discussion about extending this process by automation as well as determining some of the limitations of this solution.
1.1 Distributed systems and correctness properties

Since distributed systems operate without a global clock using message passing they are vulnerable to cascade failures where singular nodes limit the execution. Most distributed and concurrent systems have to address liveness and other correctness properties in some way to ensure a reliable system. Any distributed computation can be vulnerable to both deadlocks and starvation problems which could severely impact or impede computation. To minimize these problems algorithms and communication can be validated using proof’s as showed in [12] or simulations as explained in [13, 4]. The following paragraphs contain definitions of the correctness properties and concurrency failures necessary for our work.

Safety  The safety property basically states that bad things never happens. The definition of a ”bad thing” varies depending on the system but in general it holds that a system violates the safety property if it enters an illegal state. Formally, for a system $S$ where the state $\alpha$ represents deadlock. Safety with respect to $\alpha$ is the assertion that $\alpha$ is false for all states reachable from an initial state $S_0$.

Liveness  The liveness property states that ”something good eventually happens” which means that the system, at some, point will reach a desired state. Since the property works with infinite time as displayed by the use of ”eventually”, there is no way of disproving a system. Formally the liveness property can be summarized with the following. For a system $S$ where $\beta$ is a desirable state, $\beta$ is true for some state reachable from $S_0$. Therefore the liveness property is often split into parts that can be proved separately. If true the system should enter a desired state at some point of the execution. Common parts that will be necessary for our further proof is stated below.

- Fairness: This property states that every process should be executed infinitely often.
- Termination: The property states that from an initial state every behavior of the system leads to a configuration where all the guards (all execution paths is locked in there final state) are false.

The safety and liveness properties are formally independent since the fact an initial state eventually enters a desired state, gives no assurance that the system never enters an illegal state. It is equally true that a system that never enters an illegal state will not necessarily enter a desired state [8, 3, 15].

As stated above a reason for proving correctness is to avoid deadlock and other concurrency errors, these errors are described in the following paragraphs

The following paragraphs contains informal definitions of the common problems of deadlock and starvation cited from [17].
Deadlock

A concurrent program $P$ is said to have a deadlock if $P$ can reach a state such that some process in $P$ is blocked in this state and remains blocked forever.

Starvation

$P$ is said to have a starvation if $P$ can reach a state on a fair cycle such that some process is not deadlocked, live-locked, or terminated in this state, but does not make progress in any state on this cycle.

1.2 Alternative approaches to proving liveness

Since the problem of ensuring liveness in a distributed system is well known, there are multiple approaches to solve the issue. We present a short description of a selection of alternative approaches. These alternatives will later be used as basis for comparison with the solution of this paper. Since we are trying to specify a method that can be automated we will compare on the basis of automation and scalability. This means that we won’t consider alternative approaches that can not be automated or only work, for small scale systems.

A solution to the problem of liveness in concurrent systems can be addressed using simulations. Simulation is intended to create a live-trace, for this to work it is necessary to establish trace inclusion for the entire algorithm. This can be done using simulation or bi-simulation which is used to establish a correlation between a concrete system $A$ and a simulation $B$. $B$ is usually a simplification of $A$ where any solution that is proven correct for $B$ is correct for $A$. An in depth survey of current simulation methods is available by Lynch [13]. Simulations have the benefit of limiting the reasoning to individual states and finite execution fragments as opposed to the entire execution. Although this does not imply live trace inclusion, since the set of live traces is a proper subset of traces as stated by Attie in [4].

Another approach to solving the stated issue is by using backwards reachability to verify termination and liveness. This is done by calculating an under-approximation of the set $\Diamond F$ where $F$ is the set of states in the program $P$. To calculate this set the method focuses on the calculation of the previous step of execution. This limits the amount of non-trivial computations, since they requires a determination of the convergence relation of each state in $P$. This approach is limited since it’s not complete and only shows a subset of the live paths. The strong points of this approach focuses on usability, which makes the solution suited to verify large scale systems. Backward reachability is not in itself complete however it can be used in conjunction with other methods to provide a complete proof. Using backward reachability can still be useful since it can simplify the termination problem so that it can be proved using other techniques [1].
2 Preliminaries

The following is a brief introduction of the area and the background information necessary for the following work.

2.1 The programming language used to describe systems

For our proofs we will use the basic language stated in [15]. The language contains assignments and while loops, as well as cobegin, coend statements. The cobegin and coend identifies the start and stop of statements that can run concurrently. This is language will be used to describe the essential states in basic concurrent systems. Apart from these statements we will use the +,- operators in our examples to display computations, this is not strictly necessary but improves readability.

With this language we can describe algorithms in pseudo-code that can later be used to prove the system. As in the original proof [15] we use ⟨ and ⟩ to denote atomic operations. Algorithm 1 shows how a basic loop is written in pseudo code.

Algorithm 1: Basic algorithm example

```
boolean p:
while ⟨p⟩ do
  a: ⟨x := x + 1⟩
end
```

2.2 Introduction to basic temporal logic

To prove liveness we use a proof based on temporal logic [15]. The proof utilizes some basic boolean operators, ∧ ∨ (and, or) aswell as →, ⇔ and equivalent) [9]. In addition to these logical operators the proof utilizes the following temporal operators [15, 10].

- □ ”For all present and future times it will be true that”
- ◇ ”At some present or future there will be true that”

These operators can be exemplified by Equation 2.1 which states that if P ever becomes true, then Q will be true at the same time or later, thus P leads to Q.

\[(P \leadsto Q) \equiv \square(P \supset Q). \quad (2.1)\]
These temporal operators combined with the language stated in section 2.1 allow us to prove liveness in systems using the proof that is stated below.

2.3 The formal proof of liveness

The following section is a short summary of the rules in the original proof [15]. All figures below are directly transcribed from the proof. There is a short introduction of the required temporal logic available in section 2.2.

For this work we will mainly focus on systems where the safety properties hold. If required these can be proved using the proof outlined in [15]. With a system that upholds safety we can use the following rules and axioms to verify if the system upholds the liveness properties.

The Atomic assignment axiom shown in Equation 2.2 tells us that if the execution enters state $S$ it has to arrive at after $S$ at some point. This is the basis for an atomic operation, since the entire operation $S$ has to execute if execution flow reaches it. The While control flow axiom shown in Equation 2.3 describes the legal states of a system once execution reaches a loop statement $w$. The rule states that the only legal states reachable from at $w$ is at $S$ which is in the loop or after $w$ which is after the loop statement.

Atomic assignment axiom

For any atomic statement $S$:

$$at\ S \leadsto after\ S.$$  \hspace{1cm} (2.2)

While Control flow axiom

For the statement $w$:

$$while\ \langle b\rangle\ do\ s: S\ od,$$

$$at\ w \leadsto (at\ s \lor after\ w).$$  \hspace{1cm} (2.3)

To prove control flow we need to show that if $\{ P \} S \{ Q \}$ is true for some statement $S$ that will eventually terminate, $\{ Q \}$ is true when $S$ terminates. This is shown in the general statement rule shown in Equation 2.4

General Statement rule

$$\{ P \} S \{ Q \}; \Box(in\ S \supset P), in\ S \leadsto after\ S$$

$$\text{in } S \leadsto (after\ S \land Q).$$  \hspace{1cm} (2.4)

For loops we need to be able to determine which of the execution states will be reached. This makes the while exit rule shown in Equation 2.5 necessary. The rule states that if $B$ is true when execution reaches $w$ execution will reach $S$. It also states that if $\sim B$ is true after some point when execution reaches $w$, the control will reach after $w$. 

$$\{ P \} S \{ Q \}; \Box(in\ S \supset P), in\ S \leadsto after\ S$$

$$\text{in } S \leadsto (after\ S \land Q).$$  \hspace{1cm} (2.4)
While exit rule

For the statement $w$: while($b$) do $s$ : $S$ od :

$$(at \ w \land \ □(at \ w \supset B)) \leadsto S;$$

$$(at \ w \land \ □(at \ w \supset \sim B)) \leadsto after \ w.$$  \hspace{1cm} (2.5)

These are the rules necessary for the formal proof, a more detailed explanation of these rules and of those rules for safety properties is available in [15].

2.4 Introduction to the use of proof-lattices

A proof lattice is used to prove non-modular algorithms which is especially useful for concurrent algorithms that have no natural decomposition into provable parts. To prove this kind of algorithm we use proof lattices to construct a hierarchical proof where each individual execution path is proved separately. The proof can then be read in a bottom up manner and, any path through the lattice that ends in the original algorithm can be used as a proof. This proof is considered a hierarchical proof since the path is a totally ordered hierarchical view of the possible subsets of an algorithm [18].

Algorithm 2: Basic algorithm example for use in proof lattice

boolean p:

a: cobegin
   b: $(p := false)$
   □
   c: $(p := true)$
coend

The usage of proof lattices is shown in Figure 1 where the algorithm outlined in algorithm 2 is stated in proof lattice form.

![Figure 1: Proof lattice of Algorithm 2](image)

Each step of the lattice in Figure 1 translates to a row in the algorithm. At each
step we can use the rules stated in the formal proof to show what the resulting state of the row would be.

- at \( b \): as shown in Equation 2.2 we can see that it leads to \( after \ b \)
- similarly \( at \ c \ \rightarrow \ after \ c \)

With these steps it is a fact, that for this system Equation 2.6 is valid.

\[
(at \ b \land at \ c) \rightarrow (after \ b \land after \ c)
\]  
(2.6)

2.5 Definition and usage of petri-nets

This section contains a brief introduction to petri-nets and their usage in our context. Formally a petri-net is a directed, weighted, bipartite graph consisting of places and transitions. A place is usually a "state" or input data while a transition usually represents logic or computation. The formal definition of a petri-net is shown in Table 1. Apart from this definition the transition (i.e., firing) rule shown below describes a valid simulation of a dynamic behavior in a petri-net.

### Table 1  
**Formal definition of a Petri net**

<table>
<thead>
<tr>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Petri net is a 5-tuple, ( PN = (P,T,F,W,M_0) ) where:</td>
<td>( P = {p_1, p_2, ..., p_m} ) is a finite set of places</td>
</tr>
<tr>
<td></td>
<td>( T = {t_1, t_2, ..., t_m} ) is a finite set of transitions</td>
</tr>
<tr>
<td></td>
<td>( F \subseteq (P \times T) \cup (T \times P) ) is a set of arcs (flow relation),</td>
</tr>
<tr>
<td></td>
<td>( W : F \rightarrow {1, 2, 3, ..., } ) is a weight function,</td>
</tr>
<tr>
<td></td>
<td>( M_0 : P \rightarrow {0, 1, 2, 3, ..., } ) is the initial marking,</td>
</tr>
<tr>
<td></td>
<td>( P \cap T = \emptyset ) and ( P \cup T \neq \emptyset ).</td>
</tr>
</tbody>
</table>

A Petri net structure \( N = (P,T,F,W) \) without any specific initial marking is denoted by \( N \). A Petri net with the given initial marking is denoted by \( (N, M_0) \).

### Transition rule

1. A transition \( t \) is said to be **enabled** if each input place \( p \) of \( t \) is marked with at least \( w(p,t) \) tokens, where \( w(p,t) \) is the weight of the arc from \( p \) to \( t \).

2. An enabled transition may or may not fire (depending on whether or not the event actually takes place).

3. A firing of an enabled transition \( t \) removes \( w(p,t) \) tokens from each input place of \( p \) of \( t \), and adds \( w(t,p) \) tokens to each output place of \( p \) of \( t \), where \( w(t,p) \) is the weight of the arc from \( t \) to \( p \).

A basic example of a concurrent system modeled as a petri net is shown in Figure 2. The net represents a concurrent program which begins at the firing of \( t_1 \) which triggers the concurrent firing of \( t_2 \) and \( t_3 \). In the example net the initial marking \( M_0 \) places a token in \( P_5 \).
A more detailed introduction of the use of petri-nets with descriptions of symbols and patterns used to model computational systems is available in [7, 11].

2.6 Summary of the choice in modeling language

To solve complex real world problems we use abstractions to translate and simplify the environment. These abstractions create a model of a problem, and can be used for computations that translate to the original system [2]. For concurrent processes the complexity of the abstractions is greatly increased especially in the case of distributed systems without a global clock. To solve this a multitude of modeling languages have been designed to represent these inherently dynamic systems. These languages rarely attempt to model a complete system but focuses on solving parts of the stated problem.

There are a number of languages that can be used for modeling distributed or concurrent systems, these languages can be separated based on their inherent formality as well as their focus.

Since the work in this thesis will be of a formal nature the chosen language will have to satisfy some level of formality.

To select a language that offers the desired functionality we have made a short survey of some useful languages. These languages have been selected based on their formality and their handling of concurrent execution. The selected languages have then been divided into further subgroups based on their inherent formality or other attributes.

During our division we decided that languages with a focus on displaying the sequence of events in a system should be grouped together. This kind of language is exemplified by sequence-diagrams, a Unified Modeling Language based diagram
that focuses on describing object interaction over time. The focus on time and communication makes this kind of language well suited to display a sequence of events over time as well as handling multiple concurrent events. This focus does however come at the cost of limiting the level of formality, thus the type of language is of limited use for our work [6].

Another group of modeling languages approaches the problem from a mathematical point of view, this allows for a high level of formality at the cost of increased complexity. The high level of formality lead us to use the languages based on mathematical structures as our main candidates. From this group of languages we selected process algebra and petri-nets, two languages that differ both in their structure as well as their graphical representations. Process algebra is an algebraic approach to the study of concurrent processes, it uses algebraic statements to specify a system. With an algebraic representation of the system the language uses calculi to verify and validate the model. The mathematical nature of the language makes it well suited for automation of verification and it can be extended to handle temporal logic even though it’s mainly used to validate algorithms for concurrent calculations [5]. Petri-nets is a graphical modeling language that was created to describe concurrent operations, it’s mathematical base creates a strong foundation of formality as well as a rigid framework that allows for extensions of the original language. The language contains constructs for handling time as well as most properties of distributed systems [7]. However a common feature of the languages with a strong connection to mathematical representations is the high level of formality which leads to a significant level of complexity in the modeling even for very small systems.

The languages presented above all share some common features, especially in being able to handle temporal logic which is a necessity for the proof. Apart from this they also show some ability to model large scale systems as well as having the necessary formalism for the proof.

For this work we will use petri-nets to translate our formal proof. The use of petri-nets is based on the fact that it’s well equipped to handle concurrent processes as well as handling operations over time. Apart from this petri-nets also provide the level of formalism necessary for proofs while still being relatively innovative. It is also useful for automating the validation process of systems. And in contrast with process-algebra and other formula based languages it does provide a significant difference in the modeling form from the original proof. The limitations of petri-nets mainly appear in large scale systems where the model might get extensive. This will in some ways limit the scalability of the petri-nets modeling. Even though petri-nets allows for the use of sub-nets to allow for modeling of larger systems, it will still be required to create a large amount of complex models.
3 Translations and verification of the petri-net representation

Below we show the process of proving a basic algorithm both with the original proof and a petri-net representation of the system. The use of the rules is similar to the way that they are utilized in the original proof [15], which consists of using the rules to determine the state of the system at each step. These steps can then be used to create a hierarchical proof where if each individual part is live the system is live.

With the use of the rules stated in section 3.1 we can create a petri-net representation of our system and use that representation to determine the execution path without being forced to display an execution sequence. The similar usage patterns between the original rules and the petri-net representations allow us to follow the same proofing principles and use proof-lattices to show the validity of each independent step.

To be able to prove liveness using these methods our modeling language needs to support liveness. Since the concept of liveness exists in petri-nets we can use this for our proof, although we have to make slight adjustments to the definition to better fit our usage. In Equation 3.1 we formalize the concept of liveness in petri-nets, this definition states that for a petri-net to be live, all states must be reachable no matter what legal state the system reaches [14].

\[ A \text{ net } W(N, M_0) \text{ is considered live iff, no matter what marking has been reached from } M_0 \text{ it is possible to ultimately fire any transition of the net by progressing through some further firing sequence.} \]

Since the formal definition of liveness in petri-nets is similar to the definition of liveness in distributed systems [8,3,15,4], it can be adapted for our purpose. This adaptation can be achieved by relaxing the reachability demands, which leads to Equation 3.2 where we state that to fulfill liveness, the desired transition must be reachable from all legal states. This differs from the definition stated in Equation 3.1 where all possible states needs to be reachable from any legal state. This differentiation allows us to prove that a finite system reaches a desired state. Even though the adaptation does lower the formal requirements for liveness in the net, it does provide enough strength to ensure liveness for distributed systems.
The net $W(N, M_0)$ is considered live iff, no matter what marking has been reached from $M_0$ it is possible to ultimately fire the desired transition $t_\beta$ by progressing through some further firing sequence. \hfill (3.2)

Similarly to the original proof \[15\] we assume that all operations can be considered atomic. We also assume that the system upholds safety, this can be verified using the safety proof stated in \[15\]. We also extend the petri-net language presented by Halder \[11\] to improve the modeling granularity for loop conditions in our models. The aim for this extension is to model the exit condition of our loop as a separate state. This enables our translations since a separate exit condition can be altered from an external process.

To achieve this we use the syntax of the if/else statement from \[11\] which is used to prioritize between the exit transition and the loop transition. This is shown in Figure 3 where execution enters the loop when the transition $w$ fires. When the state $s$ is active the system does an if/else check and fires exit if possible otherwise $w$ fires. The extension adds a circular endpoint to arcs, the circle represents a ”else” statement and should be accompanied by a regular arrow shaped endpoint. This is shown in Figure 3 where the token in $s$ if possible is used to fire exit otherwise it is used to fire $w$. This allows the loop to run until execution places a token in the exit condition state $b$. Since the state of $b$ should stay the same until this explicitly altered, the exit transition replaces the token in $b$ after firing.

With our extension to the petri-net language and the assumptions of safety and fair scheduling we can translate the rules and axioms to petri-nets. These translations specified in section 3.1 will be used to show the legal states after each operation, which allows us to model and verify the execution flow of our system.
Figure 3: A basic petri-net loop
3.1 Translation of formal rules to petri-nets

This section shows how the rules stated in the proof [15] are modeled using petri-nets. The translations are displayed in a before/after structure where the initial marking is displayed in the first model and the resulting states are showed in subsequent images.

We begin by translating the atomic assignment axiom. We can use the rules to show that if we reach $s$ we will, at some time, reach $after s$. This is shown in Figure 4 where we show the initial and the resulting state of the system.

For any atomic statement

$S : at S \rightsquigarrow after S.$

Figure 4: Atomic assignment axiom

To translate the while control flow axiom we need to show two possible states. This leads to Figure 5 where we can see that the states that are reachable from $w$ is $s$ or $after w$

For the statement $w : while \langle b \rangle do : S od$, $at w \rightsquigarrow (at s \lor after w)$.
The general statement rule shown in Figure 6 shows us that iff $p$ and $q$ is reached we can reach $after s \land q$.

$$
\{P\} S \{Q\}, \Box (in S \supset P), \text{ in } S \leadsto (after S \land Q).
$$

Figure 6: General statement rule
Finally we translate the while exit rule in Figure 7 that read from left to right shows the resulting states possible. With this rule we can see that as long as \( b \) has not been reached the legal state when \( w \) has been reached is \( s \), only if \( b, s \) and \( w \) are reached is the state after \( w \) legal. To model this loop we have adapted the basic loops available in petri-nets to visualize the control state of the loop. The result of this is the addition of the state \( b \) which represents the exit condition of the loop.

For the statement \( w : \text{while}(b) \text{ do } s : S \text{ od} : \)
\[(at \ w \land \Box(at \ w \supset B)) \nRightarrow S; \]
\[(at \ w \land \Box(at \ w \supset \neg B)) \nRightarrow \text{after } w. \]

Figure 7: While exit rule
3.2 Verifying petri-net translations against the formal proof

To exemplify our findings we will utilize a minimal working example of a concurrent system. Our example is shown in Algorithm 3 and it is also described in [15].

Algorithm 3: Basic concurrent system
/* Introducing variables */
boolean p: true
integer x:

/* Begin of concurrent execution at a: and c: */
a: cobegin
  b: \( p := false \)
  c: while \( p \) do
    d: \( x := x + 1 \)
  end
coend

The algorithm described above contains a basic concurrent call where one node enters a while loop \( while(p) \) until \( p = false \), a state that the second node puts \( p \) in at some time. Below the algorithm will be modeled using petri-nets and proved to be live both using our petri-net adaptation of the formal proof as well as the original proof [15]. An informal analysis of the algorithm leads to the conclusion that the system is live since \( P \) will change state to \( true \) at some time during execution.

Original proof The proof for the system shown in Algorithm 3 is stated in [15]. The following will be a brief reiteration of this proof structured around a proof lattice [18], shown in Figure 8.
The process will be documented in a step by step process where each line connects to a point in the proof lattice.

1. At this step we can apply the atomic assignment axiom to the statement b, which leads us to the conclusion that no matter what state b started in, it ends with \( p = false \).

2. Due to the safety properties we can state that once control reaches after b it has to stay there and \( p = false \) will always be true.

3. At this step we can use the while control flow axiom to show that the program must be either in c or after c the box indicates that \( □(after \ b \ ∧ \sim p) \) is true for all descendants of this node.

4. This follows from the fact that in c and at c \( ∧ \ at \ d \) are equivalent.
5. This follows from the atomic assignment axiom is applied to $d$, and the fact that $after\ d$ is equivalent to $at\ c$.

6. Since the surrounding box shows that $\square \sim p$ is true at this node we can apply the while exit rule to show that control eventually leaves the while loop thus making $after\ c$ true at some point.

7. This final implication leads to $after\ c \land after\ b$

In these steps we used the rules of the proof to show that control at some point will leave the while loop, thus ensuring a system that reaches our desired coend statement.

---

**Petri-net representation**  
Apart from constructing a proof using lattices we can construct a model of our system from the ground up to pinpoint any failure points. Below we use this tactic to model the system stated in algorithm 3, and by using the rules stated in section 3.1 we can determine the next legal state of the system for each execution line. If the resulting model reaches our desired coend without violating the stated rules we can be assured that our system is live by the definition stated in [Equation 3.2](#).

As shown in [Figure 9a](#), we can use the atomic assignment axiom to ensure that execution will reach $a$: cobegin. Similarly [Figure 9b](#) can use the atomic assignment axiom which leads us to $b$: and $c$: When we reach [Figure 9c](#) we can see that the atomic assignment ensures that $b$: fires, while the while control flow axiom shows that either $d$: or $after\ c$: will be active. Our fourth step [Figure 9d](#) can use the while exit rule to show that $iff$ transition $b$: fires will we reach the state after the loop. Finally [Figure 9e](#) again uses the atomic assignment axiom to show that the coend transition will fire at some time. This leads us to the statement that our finished model [Figure 9f](#) can be considered live according to the definition stated in [Equation 3.2](#).

---

(a) Initial state of system before cobegin  
(b) State in multiple execution flows after cobegin statement
Figure 9: Step by step model of algorithm 3 using petri-nets
By proving our system using both the rules stated in the original proof as well as our adapted petri-net representation we can show that our model works for the positive case where the system is live. Below we will show that the adaptation works for a non-live system, by first showing that the original proof determines the system to be locked followed by showing the same conclusion using our model. For this proof we will use the system described in Algorithm 4 that enters a deadlocked state since \( p \) never changes state.

**Algorithm 4: Deadlocked system**

```plaintext
/* Introducing variables */
boolean p: true
integer x:

/* Begin of concurrent execution at a: and c: */
a: cobegin
   b: \langle x := 1 \rangle
   c: while \langle p \rangle do
      d: \langle x := x + 1 \rangle
   end
coend
```

Similarly to the correct algorithm, we will show the proof for Algorithm 4 both using the original temporal logic form as well as our adapted petri-net representation. Informally we can see that the system never exits the while statement in \( c \), since the exit condition \( P \) never reaches a state where \( P=false \).

**Original proof** To determine if the system described in Algorithm 4 can be considered live we utilize the techniques stated in the original proof [15]. We use this to create a proof lattice, see Figure 10 that allow us to determine the resulting state for each transaction. In a similar way as the proof for the system described in Algorithm 3 we create a proof lattice and describe each step of the proof to verify our informal conclusion. To be able to show this we need to show that \( \Box \sim p \) is false for all execution paths of our system.
1. At the first step we can show that \textit{at} \textit{b} leads to \textit{after} \textit{b} using the atomic statement axiom.

2. At this step we can use the while control flow axiom to show that the program must be either \textit{in} \textit{c} or \textit{after} \textit{c} the box indicates that $\Box(\text{after} \textit{b})$ is true for all descendants of this node.

3. This follows from the fact that \textit{in} \textit{c} and \textit{at} \textit{c} $\land$ \textit{at} \textit{d} are equivalent.

4. This follows from the atomic assignment axiom is applied to \textit{d}, and the fact that \textit{after} \textit{d} is equivalent to \textit{at} \textit{c}.

5. Since the surrounding box shows no change in the initial state of \textit{p} it is not possible to use the while exit rule to escape the while loop. This leads our system to a deadlock a a final state of \textit{in} \textit{c} $\land$ \textit{after} \textit{b} and not the desired \textit{after} \textit{c} $\land$ \textit{after} \textit{b}.

\textbf{Figure 10:} Proof lattice of algorithm 4
As shown above we can verify that the initial state of \( p \) never changes. This leads to the exit condition of the loop never to be satisfied. Since our system never leaves the while loop it never reaches the desired \( \text{coend} \) and can not be considered live.

**Petri-net representation**  
In a similar way as we proved algorithm 3 we construct our proof through a step by step modeling of our system using the rules stated in section 3.1 to determine the resulting state for each action. If we can show that all paths lead to our desired state we can consider the petri-net and the system to be live according to the definition of liveness showed in Equation 3.2.

Initially we can use the atomic assignment axiom to determine the resulting state from the \( \text{cobegin} \) statement, this is shown in Figure 11a. For the second step we can see that the algorithm enters the while statement at \( c \): and according to the while control flow axiom it will stay in either \( d \) or \( \text{after } c \); the state \( b \): is reached by the other concurrent process, this is shown in Figure 11b. At the third step we can see that the \( b \): state has changed to \( \text{after } b \): according to the atomic assignment axiom, this leads the system to be locked at the \( \text{coend} \) synchronization due to the while exit rule, since \( P \) will always be \( \text{true} \), this is shown in Figure 11c. In Figure 11d we can see that the system is deadlocked, since the desired transition \( \text{coend} \) can not be reached from the while entrance state in \( c \).

![Diagram](image-url)
As shown above we can use our petri-net adaptation to both detect a live and a non-live system, the detection is verified against the original proof. This leads us to conclude that our adaptation can be used to prove liveness properties in distributed systems.
4 Discussion

Below we compare our work from a perspective of usefulness against other approaches for determining liveness of a system. We also discuss the benefits of our approach as well as the possibilities to extend and automate the process in order to verify complex systems. Apart from this, we discuss, shortly, the limitations of our approach and the level of verification required to prove that our approach is correct for all cases.

4.1 Automation

Our solution is currently limited to manual verification, but since our work is based of petri-nets we have positive evidence that the manual process can be automated. We make this assumption based on previous work where task have been automated in petri-net representations [16, 19]. To automate our solution one could use the rules stated in section 3.1 to verify a net. This could be done by following the steps outlined in our proof which if automated could display the execution sequence. To use this method to verify liveness we would then need to verify that execution reaches our desired state.

The subject of automation can be handled using both backwards reachability and simulations. Since our approach is based on a high level of abstraction our solution limits the amount of non-trivial computations, this leads to an improvement from the previously mentioned solutions that require multiple steps of non-trivial computations. Our solution with further development to allow for automation will then provide a functional proof of liveness for small scale system with limited user interaction.

4.2 Large scale systems

Since we use petri-nets to translate the proof we inherit the complexity in modeling large scale systems. This complexity does limit the scalability of our models. However, this limitation is only a drawback when attempting to simultaneously model an entire system. Instead, we can use a divide and conquer approach to prove individual sections of an algorithm and consider those sections as atomic operations. This approach is well suited for object-oriented or modular systems where we can prove subsets of the system and organize these sub-proofs into a hierarchical proof.

Even though this type of approach is viable for the alternative approaches they do contain steps of deduction that forces large parts of the system to be modeled.
These deduction steps also limits the possibility of automating the process. Since our solution is based on a fix set of rules it does allow basic automation.

4.3 Benefits and limitations

With our choice of modeling language we inherit the tools and limitations of the modeling language. Since petri-nets is used extensively in a multiple of research areas there exists a large amount of tools for analysis that can be utilized in all nets. Since we use a visual representation of our system where states and execution flow can be modeled our solution can be used for educational purposes.

Due to the increased usage of distributed systems in production environments a simplified and automated proof of liveness could provide increased operational security. A simpler way of maintaining theses properties would also limit the risk of deadlock and starvation which can leads to underutilization of hardware.

Our approach does inherit the limitations of petri-nets which can lead to complex models for larger systems. This limitation can in some ways be minimized by creating sub-models. Due to our use of petri-nets we are limited to modeling the execution flow, this leaves us without any information of performance. However our approach does model a system with enough information to allow us to prove liveness.

The work done in this thesis have focused on proving the concept of translating profs to petri-nets. The results of the thesis have showed that the concept is sound although further study is necessary to formally verify it’s applicability for all systems.

4.4 Further work

To further extend the method incorporation of time would enable the model to provide information such as performance. There is also the possibility to generate petri-nets from existing code to verify the execution. This combined with automatic verification would allow for a completely automated verification of existing systems. The verification could then be used to detect and highlight problematic areas from a liveness viewpoint. Apart from generating the petri-nets automating the verification of the models is an area where improvements can be beneficial.
5 Bibliography


