Rewriting TLA+ Fairness Conditions for Symbolic Model Checking

Sage Łuszczyk[1]
Supervisor(s): Dr. B. Ahrens[1]**, Dr. A. Lukina**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Sage Łuszczyk
Final project course: CSE3000 Research Project
Thesis committee: Dr. B. Ahrens, Dr. A. Lukina, Dr. T.J. Coopmans

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Model checking is a formal verification method that uses software to confirm a model of a system fulfills chosen properties. Systems in the TLA+ language are composed of states and transitions. The Apalache model checker for TLA+ is currently unable to reason about whether a system can take a given transition, while TLC can. This is important for verifying properties that assert that a given transition will eventually be taken if it possible. We apply a method from literature for rewriting these properties into a format Apalache accepts on three case studies, with the aim of finding improvements to the method. An equivalent rewrite is found for all cases, though requiring incremental modifications to the literature method. We find that expressions which depend on the updated values of variables after a transition must be distributed across disjunctions. We also note that expressions which depend on the results of prior expressions should be at the same or greater nesting level, and that if the system can pick from a set of new values for a variable, the value chosen must remain constant for the entire rest of the expression, also requiring indenting. From these results, we propose a general method for further research, relying on repeated additional nesting, applied whenever an expression is dependent on a previous expression's variable assignments. We leave its validation for future work.

1 Introduction

To quote Edsger W. Dijkstra, "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness." [1]. Engineers who wish to verify that their systems are correct can use formal methods to create such proofs. The two notable approaches are developing these proofs manually, or alternatively, modeling the system, and having a model checker verify it automatically [2, p. 8]. A model checker takes as input a specification, and the properties it should fulfill, which are both usually written in a dedicated language. TLA+ is one such language, and is especially relevant given its use by companies like Amazon, Microsoft, and Thales to verify fault-tolerant distributed systems [3, p. 27].

The two main model checkers for TLA+ are TLC and Apalache. TLC explicitly enumerates every state of the system [4, p. 60], while Apalache transforms the specification into a solvable logical formula [5, p. 2-3]. Due to this difference in approach, they do not support the same subset of TLA+ properties. Konnov et al. [6, p. 95-96] find that Apalache is capable of checking that a system does not exhibit incorrect behavior, but not that it will eventually accomplish a given goal, unlike TLC. However, a method of rewriting such properties into a format Apalache can verify has since been implemented [7]. This method does not yet apply to properties that involve the assertion that a certain transition between states is eventually taken if possible, known as a fairness condition. The Apalache documentation recommends that such conditions are specified manually [8].

We thus investigate if a rewriting approach can also be applied to allow Apalache to check liveness properties with fairness conditions. Offtermatt et al. [9, p. 25-26] briefly propose a set of rules that can be used for such a rewriting, but this approach has not yet been verified.

Our research focuses on applying these rewrite rules to a series of increasingly complex case studies, verifying if their manual application results in a correct rewrite, and if not, iteratively proposing modifications. We then propose a more general, albeit unverified method by combining insights from the case studies. Section 2 provides further background information for the problem, using a simple motivating example. Section 3 presents our contribution to the problem. Section 4 elaborates on the method used to study the rewrite rules. Section 5 presents a simple toy example case study. Section 6 analyses the application of the rewrite rules to Dijkstra's EWD840 termination detection protocol. Section 7 elaborates on the results of the more complex Folklore Reliable Broadcast Algorithm case study. Section 8 presents a discussion of the findings in the prior sections, with synthesis of an improved general method, which we suggest as a further research direction. Section 9 contains the conclusions and limitations of the study, and proposals for future work.

2 Background

2.1 Model Checking in TLA+

This subsection aims to introduce model checking using a traffic light as a motivating example. We explain safety, liveness, and fairness in the context of TLA+, and how the TLC and Apalache model checkers verify properties.

Consider a system, consisting of a traffic light and a button pedestrians can press to request for the light to turn green. This example is heavily inspired by an Apalache tutorial on temporal properties [10]. We wish to model this system, determine the properties it should have and the behavior it should exhibit, and verify that our description does indeed exhibit this behavior. The system starts with the light at red and without a switch to green being requested. When the button is pushed, the system should register that a request has been made. Once this is the case, the light should eventually turn green, and then eventually back to red, returning to the starting state. Such a system can be modeled as a finite state machine (Figure 1). We consider time to flow in discrete steps, and assume that for each state, the system can stutter (not update any variables, thus remaining in the current state).

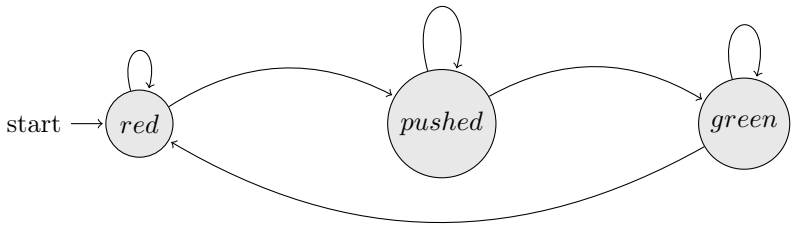

Figure 1: FSM representation of traffic light, as modeled in TLA+

We now introduce a method of modeling and verifying this system. TLA+ (Temporal Logic of Actions) is a language for writing formal system specifications and properties. The language is built on top of linear temporal logic, which is a version of standard propositional logic which is capable of reasoning about time, by including predicates such as "always" and "eventually". TLA+ properties can be divided into safety and liveness. A safety property asserts that the system does not exhibit incorrect behavior, for instance the light turning green without a request. A liveness property asserts that a desired outcome occurs, for instance that the light will turn green eventually if requested. This can be expressed in the form below, where `[]` represents "always", and `<>` represents "eventually".

```
RequestWillBeFulfilled == [](requestedGreen => <>isGreen)
```

An issue with this property is that the traffic light may "stutter" indefinitely after the button is pushed, meaning that the light never turns green. To be able to verify this property, we must introduce the notion of fairness, specifically weak fairness. In TLA+, a transition from one state to the next is known as an "action". Weak fairness is the assumption that if an action is eventually always possible to take, it will always eventually be taken. In TLA+, this can be written as [9, p. 25]:

```
WF_vars(Action) == <>[](ENABLED <<Action>>_vars) => []<><<Action>>_vars
```

Here, `WF_vars(Action)` is the assumption of weak fairness on a given action, with the variables in `vars` changing when the action is taken. The angled brackets around `«Action»_vars` specify that when the action is taken, the variables must not all remain unchanged, which prohibits stuttering behavior. The expression is equivalent to writing `Action /\(vars # vars')`, where `#` is the "not equal to" operator, and `vars'` is the value of `vars` in the next state. The fairness assumption is thus crucial for verifying properties of systems that can stutter. Strong fairness is highly similar, aside from the fact it relates to an action that is only cyclically enabled, rather than permanently enabled.

TLA+ combines temporal logic for specifying liveness properties, with simpler predicate logic for safety properties [11, p. 2]. Specifications can then be verified using a model checker. Model checkers are tools that, given a system specification and its properties, verify that the specification fulfills the given properties.

The two main model checkers for TLA+ are TLC and Apalache. TLC explicitly enumerates every state the system can reach, based on the initial state and possible transitions [4, p. 60]. For our example, it begins at the *red* state, then follows the transitions we specified, thus reaching the *pushed* and *green* states. It verifies that the properties we specify hold for every state. Apalache is a bounded symbolic model checker, meaning it instead transforms the specification and its properties into a logical formula, which an SMT (Satisfiability Modulo Theories) solver then verifies [5, p. 2-3]. This formula is generated for a certain number of steps, meaning that Apalache verifies that properties hold in the first k steps. When introducing Apalache, Konnov et al. [5, p. 22-24] ran several benchmarks, noting a speedup from days to seconds for several problems. They use a consistent testbed, and a variety of representative benchmark specifications, supporting the validity of their results. However, they show that Apalache's performance mainly improves when testing inductive invariants, while we focus on bounded model checking. There, Apalache runs faster for versions of the same problem with, for instance, more nodes in the system, but is slower for smaller problem sizes. We see their results as still applicable in our case, due to them supporting use of Apalache in model checking larger problem sizes.

2.2 Model Checker Limitations

This subsection presents the currently known limitations of TLA+ model checkers. We evaluate Konnov et al.'s [6] findings, focus on Apalache's limitations in verifying temporal properties, and present a method to enable it to apply fairness conditions via rewriting. Since TLA+ is a powerful and general language, neither model checker is able to verify all constructs that can be expressed in it.

2.2.1 State Space Explosion

The exponential growth in possible states as a system model becomes more complex is a severe limitation of model checkers in general, and applies to TLC especially, as highlighted by Konnov et al. [6, sec. 3.1]. Their paper investigates the limitations of the two TLA+ model checkers via a case study on Safra's algorithm for distributed termination detection. They mention that the finite state approach can not handle infinite state spaces, and as such requires bounds to be configured for parameters such as the amount of messages passed in a protocol. For Apalache, they note that, as an SMT-based solver, it is capable of reasoning about infinite state spaces. Their paper shows how the model checkers handle increasing problem size for the case study, but does not find hard boundaries on what either can not verify due to not being expressive enough.

2.2.2 Temporal Properties

Konnov et al. [6, p. 95-96] highlight the tradeoff of using an SMT solver for model checking, which is that Apalache is unable to check temporal properties, and requires bounds on the amount of steps for the model to run. They however find that, for their case, the temporal property they specified can be reformulated as an invariant, allowing it to be checked regardless. Of note, Konnov et al.'s [6] paper was published in 2022, and as such the stated limitations of Apalache have changed since then, thus statements about them were re-verified before we accepted them. However, due to the slow pace of development of TLC, the rest of their findings remain applicable.

Apalache's main limitation of not being able to verify temporal properties can to some extent be overcome automatically by converting liveness properties to equivalent safety properties [7]. For the Eventually and Always operators, this is accomplished by applying Biere et al.'s liveness-to-safety conversion [12, sec. 2]. As the Apalache documentation [10] notes, for the traffic light example, the method searches for a counterexample to the liveness property. This is a loop of states that the system can keep repeating perpetually, that results in it never fulfilling the chosen liveness property.

A counterexample for `RequestWillBeFulfilled` would be a loop consisting of the system staying in the state where the button is pushed, but never taking the transition for the light turning green. The method adds several additional variables to track the loop state, values of variables, and the evaluated values of temporal properties within the loop. Such a rewrite adds significant complexity, but makes it possible to check liveness properties by rewriting them into equivalent safety properties.

2.2.3 Rewriting Properties

The previously mentioned approach is not sufficient to implement checking if an action is enabled, and thus can not implement the TLA+ `ENABLED`, `WF` (weak fairness), or `SF` (strong fairness) operators, with official documentation recommending that the user attempt to specify them manually [7] [8]. The boundary between temporal properties that can and can not be verified by Apalache thus depends on the encoding used. The `WF` predicate can be manually rewritten in the following form [9, p. 25]:

```
ManualWF == <>[](ENABLED <<Action>>_vars) => []<><<Action>>_vars
```

The issue is with Apalache's support for `ENABLED`, the predicate that checks if a given action can be taken in the current state. Offtermatt et al. [9, p. 25-26] briefly propose that Apalache's transition finder can preprocess `ENABLED` predicates into a form accepted by Apalache, in order to allow for support of fairness. They suggest that actions can be split into conditions (`x >= 5`), and assignments (`x'=x+1`, assigning the value of `x` at the next step). To rewrite an action into its `ENABLED` predicate, uses of updated variables (`x'`) can simply be syntactically replaced with the assigned value (`x+1`), and all assignments can be replaced with `TRUE`. We take this as the initial set of rewrite rules. They note that not all actions are supported by the transition finder, and this approach is thus incomplete, but do not elaborate further. The relevant slide [9, p. 26] also only considers actions that are conjunctions of these assignments and conditions, not including disjunctions or more complex TLA+ structures. No study has been attempted to verify the soundness of this rewriting approach.

3 Our Rewrite Method Contribution

This section elaborates on the existing knowledge gap in Apalache's limitations, and how our research in applying rewrite rules contributes to the understanding of these limitations.

Given that experimental verification of Offtermatt et al.'s [9, p. 25-26] rewrite rules has not yet been performed, it is currently unclear if Apalache is able to verify arbitrary liveness properties by simple rewriting, or if further complexity exists. If the initially proposed rules are insufficient, potential improvements would further expand the body of knowledge. Thus, there exists a knowledge gap in methods of verifying liveness properties with fairness conditions using Apalache.

This paper aims to answer the research question "How can liveness properties with fairness conditions in TLA+ be rewritten to allow Apalache to check them?". We accomplish this via an iterative process of applying and refining rewrite rules. The end result is an improved set of rewrite rules, albeit not one that has been shown to apply to all specifications. We then propose a more general method based on insights from our findings. Accordingly, the research question can be divided into subquestions, with each case study contributing to both iteratively.

- Can the given weak fairness condition be rewritten with the currently known method?
- If not, how can the method be modified to allow it?

4 Methodology

4.1 Experiment Methodology

This subsection explains and justifies the iterative process we apply to each case study, consisting of applying rewrite rules, discovering issues, and proposing solutions. We aim to answer the research question "How can liveness properties with fairness conditions in TLA+ be rewritten to allow Apalache to check them?". This is done on the basis of case studies, for which `ENABLED` predicates are written and verified to be correct, and then used as part of manually specified fairness conditions.

4.1.1 Assumptions

Our research is based on the assumption that a proposed rewrite is correct if and only if TLC finds it to be equivalent to its own builtin `ENABLED` predicate. We verify this automatically by model checking, minimizing human error. However, this means that we can only find that a rewrite is correct for a given model. To address this, we consider several case studies of increasing complexity, creating more opportunities to disprove a rule that was assumed to hold based on one model.

4.1.2 Metrics

To determine if a rewrite rule is correct, we apply it to an action whose `ENABLED` predicate we wish to determine. The rule is then judged on whether it can be applied to the given case, and if so, if the output is correct. We measure this checking equivalence to TLC's implementation:

```
BiImpliEnabled == ENABLED <<Action>>_vars <=> ManualEnabled
```

As a second metric, we consider whether a correct rewrite allows Apalache to successfully model check a previously uncheckable liveness property, by instead model checking:

```
ManualLiveness == ManualWF(ManualEnabled) => Liveness
```

We check if Apalache accepts the expression, which is considered most important because the aim of the research is to produce `ENABLED` rewrites that Apalache could use. If so, we confirm that the new property can be verified by model checking without returning errors. We use a step amount of 5 instead of the default 10 due to the significant increase in computation time when checking further into the future.

4.1.3 Hypotheses

For each case study, we begin with the hypothesis "Applying the rewrite rules found up to this point yields a correct rewrite". If this is disproved, by the expression being too complex to apply the rules to, or the TLC bi-implication failing, we propose a potential reason. This produces the follow-up hypothesis "The rewrite rules fail due to X, and Y is a sufficient fix". If the proposed modification to the rewrite rules yields a correct rewrite on the specification, this is taken as an indicator of correctness.

4.1.4 Experiment Approach

Each case study begins with identifying the weak fairness condition that the authors used. We then apply the rewrite rules we previously found to the `ENABLED` part of the target `WF` predicate, beginning with those proposed in literature [9, p. 25-26]. If TLC finds the rewrite to be incorrect, we propose a reason, and present updated rewrite rules that resolve the issue.

We verify correctness by then applying these new rules, and checking the result with TLC. If successful, we also confirm that Apalache can check the specification's liveness properties with our rewritten fairness condition. We use Apalache version 0.57.0, and TLC version 2026.05.18.174321, accessed via the official TLA+ for Visual Studio Code extension version 2026.5.181751.

4.2 Case Studies

This subsection justifies our specific selection of increasingly complex case studies. We assume that selecting specifications that already used WF conditions would be more informative, since the authors had already decided the chosen property was important. The liveness properties of these specifications would then also rely on WF. We consider the TLA+ examples repository to be a good source of relevant specifications to search through, due to Konnov et al. [5, p. 21] relying on the same repository for their benchmarks.

We thus consider three case studies, each focusing on a different issue. The final choices for the case studies are presented in Table 1. The Traffic Light example enables verification of the rewrite method's applicability to the problem that it was initially introduced alongside [9, p. 8-26]. The EWD840 specification allows us to investigate the effect of expressing an action as a disjunction of smaller actions. The bcastFolklore case reveals how the inclusion of guard conditions on the updated values of variables and use of the "there exists" predicate affects the rewrite procedure.

Benchmark	Description	Action Form	Source
Traffic Light	Toy example described in Section 2	Simple conjunction	Apalache Documentation [10]
EWD840	Distributed termination detection protocol	Disjunction of conjunctions	TLA+ Examples [13], Dijkstra [14]
bcastFolklore	Distributed reliable broadcast algorithm	Nested conjunctions and disjunctions, use of existential quantifier	TLA+ Examples [13], Tran et al. [15, sec. 3.2]

Table 1: Chosen case studies, including form of actions we applied rewrite rules to

5 Traffic Light

We initially consider the traffic light example discussed in Section 2, focusing on the `RequestWillBeFulfilled` property. The TLA+ source code for the model is accessible from the Apalache documentation [10]. The relevant action and its rewrite is also present in Appendix A.

5.1 Rewrite Attempt

No WF condition was present in the specification initially, because the intent of the example was to demonstrate a failing liveness property. To ensure that the traffic light would eventually turn green, we assert that the transition to green will eventually occur if it is possible. We thus focus on the `SwitchToGreen` action, which was specified by the Apalache developers as:

```
SwitchToGreen == SwitchToGreen_Guard /\ SwitchToGreen_Effect
SwitchToGreen_Guard == ~isGreen /\ requestedGreen
SwitchToGreen_Effect == isGreen' = TRUE /\ requestedGreen' = FALSE
```

The weak fairness condition for `WF_vars(SwitchToGreen)` is, when written in full:

```
<>[](ENABLED <<SwitchToGreen>>_vars) => []<><<SwitchToGreen>>_vars
```

The `ENABLED` predicate of interest is then:

```
ENABLED <<SwitchToGreen>>_vars <=> ENABLED(SwitchToGreen /\ (vars # vars'))
```

The rewrite rules are applied to the above, yielding:

```
ManualEnabled == ~isGreen /\ requestedGreen /\ TRUE /\ TRUE /\ (vars # vars')
```

However, `vars` is not a variable, but a formula, defined by the authors as `«isGreen, requestedGreen»`, a tuple of the two variables. TLC does not permit us to include primed variables in these statements if they are to be used in fairness conditions. Thus, `vars'` must be expanded using the prior definition, and the updated values syntactically replaced:

```
vars # vars'
<=> vars # <<isGreen', requestedGreen'>>
<=> vars # <<TRUE, FALSE>>
```

After removing redundant $TRUE$ clauses, the final predicate is:

```
ManualEnabled == ~isGreen /\ requestedGreen /\ vars # <<TRUE, FALSE>>
```

TLC confirms bi-implications on the `ENABLED` predicate and the WF formula it was a part of. Apalache was able to verify the following `ManualWF`-dependent property:

```
Liveness == (<>[](ManualEnabled) => []<><<SwitchToGreen>>_vars)
            => RequestWillBeFulfilled
```

Thus, for the case of the traffic light, adding the `vars # vars'` constraint to the formula which is then processed by the initially specified rules enables successful model checking of liveness properties with Apalache. However, we note that removing the non-stuttering constraint still yields a rewrite that TLC finds to be equivalent by bi-implication.

6 EWD840 Termination Detection Algorithm

We now consider a specification of the EWD840 protocol, originally presented by Dijkstra [14], and specified in TLA+ by Merz [16]. EWD840 detects if all the nodes in a system have completed their work and terminated. The first node can initiate a probe, causing a token to be passed around the nodes, which is used to determine termination. The specification contains the WF condition `WF_vars(System)`, where `System` is the action:

```
System == InitiateProbe \/ \E i \in Node \ {0} : PassToken(i)
```

`InitiateProbe` and `PassToken` are simple formulas, similar structurally to the traffic light case:

```
Action == Guard /\ Assignment
```

We present the specification and its rewrite in terms of this simplified structure, to highlight the relevant aspects. The non-shortened version is present in Appendix B.

6.1 Rewrite Attempt

Similarly to the traffic light, we must include the `vars # vars'` non-stuttering condition. However, this is not possible by applying the rewrite rules directly due the structure of `System`, and by extension `«System»_vars`, which is:

```
<<System>>_vars ==
    \/ (Guard1 /\ Assignment1)
    \/ \E i \in Set : (Guard2(i) /\ Assignment2(i))
    /\ (vars # vars')
```

It is not possible to apply the rewrite rules to create `ENABLED «System»_vars`, because `vars'` depends on whether the assignment applied is `Assignment1` or `Assignment2(i)`. Therefore, the rules found up to this point are insufficient to rewrite this expression.

6.2 Rule Modification

We propose our first addition to the rewrite rules. We base it on documented behavior of TLC, which "evaluates $p \vee q$ by first evaluating p and, if it equals $FALSE$, then evaluating q" [11, sec. 14.2.2]. Thus, the branches of a disjunction can be thought of as separate "sub-actions". We propose that a condition that must be true for every branch can be separately evaluated in every branch. Thus, if an action is in the form of a disjunction of conjunctions, the non-stuttering condition should be distributed across the disjunction branches. For our case studies, this would result in:

```
<<System>>_vars ==
    \/ (Guard1 /\ Assignment1 /\ (vars # vars'))
    \/ \E i \in Set : (Guard2(i) /\ Assignment2(i) /\ (vars # vars'))
```

We apply the `vars'` rewrite from Section 5, such that `«Assignment1Vars»` is the fully written out value of the `vars` n-tuple after `Assignment1`. The final `ENABLED` rewrite is then:

```
ENABLED <<System>>_vars ==
    \/ (Guard1 /\ vars # <<Assignment1Vars>>)
    \/ \E i \in Set : (Guard(i) /\ vars # <<Assignment2Vars(i)>>)
```

TLC confirms this manual rewrite and `ENABLED «System»_vars` as equivalent by bi-implication. However, we also find that the bi-implication still holds if the non-stuttering condition is commented out from every branch, and only guard conditions are included. Apalache was able to verify the manually specified property `ManualWFSystem => Liveness`, where `Liveness` was included in the original specification.

In summary, we find that for EWD840, the initial rewrite rules are under-specified for creating an `ENABLED` predicate from the action that WF is applied to. This is due to the action being in disjunctive form, while the non-stuttering condition is present as a separate conjunction that requires access to the updated variable values. We propose that we should treat sub-actions in disjunctions as independent, with independent assignments, and distribute the non-stuttering condition across them. This updated rule-set produces a correct rewrite for EWD840 according to our metrics.

7 Folklore Reliable Broadcast Algorithm

We now move to Tran et al.'s [17] specification of the algorithm for reliable broadcast by message diffusion. The aim of the protocol is to distribute messages across a set of potentially faulty processes, such that all non-faulty processes hold the same log of messages, and every message sent by a non-faulty process is accepted [15, sec. 3.2]. In this version of the specification, a process can receive either an INIT or ECHO message. Upon receiving either type of message, it accepts, and then broadcasts an ECHO. The next step for a process is specified as follows:

```
Step(self) ==
  /\ Receive(self)
  /\ \/ UponV1(self)
     \/ UponAccept(self)
     \/ UponCrash(self)
     \/ UNCHANGED << pc, sent, nCrashed, Corr >>
```

Here, `UponV1` can be taken when the process receives an INIT message, and `UponAccept` when receiving an ECHO. The process can also crash.

7.1 Rewrite Attempt

This specification is interesting due to the weak fairness condition the authors specify:

```
WF_vars(\E self \in Corr:
    /\ Receive(self)
    /\  \/ UponV1(self)
        \/ UponAccept(self)
        \/ UNCHANGED << pc, sent, nCrashed, Corr >> )
```

The issue differentiating this specification from other case studies is its use of the `\E` ("there exists") predicate. The previously mentioned weak fairness condition is a conjunction of receiving a message, together with how the process can react to the message. Notably, the latter require access to the set of messages received, which is updated (`rcvd'=...`) within `Receive(self)`. A shortened version is presented below, with the full relevant code in Appendix C.

```
\E msgs \in SUBSET (Proc \times M):
    /\ msgs \subseteq sent
    /\ rcvd[self] \subseteq msgs
    /\ rcvd' = [rcvd EXCEPT ![self] = msgs ]
```

This results in the initially proposed rewrite procedure failing. `UponAccept` contains the statement `rcvd'[self] # {}`. Furthermore, access to `rcvd'` is required in order to verify the `vars # vars'` requirement of the fairness condition. This condition is distributed across every sub-action, using the method modification found in Section 6. If the procedure is directly applied, `rcvd'` must be replaced with the entire above definition. However, there is no trivial way to execute this, due to the assignment being wrapped in a `\E` condition outside of scope.

7.2 Rule Modification

We begin by considering the method TLC uses to evaluate actions. When evaluating the original action, it iterates over the set used in `\E`, consistently using one value of `msgs` per iteration, and hence keeping `rcvd'` constant across the entire expression. This also means we can not take a syntactic replacement approach. If the entire `\E msgs` clause is copied each time, it would be possible for each to evaluate a different `msgs`, and thus consider a different value of `rcvd'`. This would therefore allow mutually exclusive conditions on `rcvd'` to simultaneously hold true, because each could be evaluated on a different value.

To align the behavior of the manually rewritten `ENABLED` predicate with how TLC would evaluate the underlying action, `msgs` should be bound in the entire expression. The most straightforward approach is to place the entire rewritten expression into the `\E msgs` block, adding deeper nesting as such:

```
\E msgs \in SUBSET (Proc \times M):
    /\ msgs \subseteq sent
    /\ rcvd[self] \subseteq msgs
    /\ (remaining expression)
```

This is reasonable, because TLC requires assignments to be made before the new value is used in an expression [11, sec. 14.2.6]. Thus, it is not possible for `rcvd'` to appear earlier than it is assigned. This means that this can be repeatedly nested if multiple `\E` clauses are used in an expression. `rcvd'` can then simply be replaced with `[rcvd EXCEPT ![self] = msgs ]` in each location, because `msgs` is in scope everywhere. Our rewrite, which binds `msgs` in this way, successfully model checks as correct by bi-implication, while more closely matching the way TLC handles expressions. Apalache was able to verify the manually specified property `ManualWFSystem => Liveness`, where `Liveness` is a conjunction of four of the five liveness properties in the original specification (`UnforgLtl`, `Unforg`, `CorrLtl`, `RelayLtl`). The remaining property, `ReliableChan`, could not be checked regardless of applying a fairness condition rewrite because the property itself was not accepted by the Apalache type checker.

7.3 Confirmation of Change Necessity

The problematic `rcvd'` expression is used in `vars # vars'` checks and in `rcvd'[self] # {}`. In the previous case studies, the `vars # vars'` condition was found to not be necessary for the manual `ENABLED` predicate to be equivalent to TLC's version. However, this case study does not exhibit the same property. If the `UNCHANGED «...»` section of the action is simply rewritten as `TRUE`, as per the initial rewrite rules, the bi-implication fails, due to the `ManualEnabledAction => ENABLED «Action»_vars` section being falsified. Therefore, it can be concluded that it is, in this case, necessary to check that `vars # vars'`, and thus the complications arising from including this check can not be avoided.

Furthermore, if the `rcvd'[self] # {}` check is commented out, the bi-implication once again fails, due to `ManualEnabledAction => ENABLED «Action»_vars` not holding. Thus, access to `rcvd'` is indeed necessary, so no straightforward method to avoid binding `msgs`, and values bound through `\E` generally, is likely to exist. We therefore consider this transformation or an equivalent to be required for correctly expressing fairness conditions manually, indicating the initially proposed rewrite rules are incomplete. We thus, based on the results of this case study, propose increasing sequential nesting of subsequent expressions when `\E` is used as a modification to the rewrite rules.

8 Discussion

This section summarizes the results, and then interprets them, arguing that the non-stuttering condition must be included, and proposing a more general method of rewriting relying on increasing nesting of expressions based on the assignments they depend on, combined with distributing expressions over disjunctions of assignments. We then suggest cautious use of this method through verifying its results with TLC by bi-implication before further use, until Apalache natively supports fairness.

8.1 Summary

The aim of our research was to answer the question "How can liveness properties with fairness conditions in TLA+ be rewritten to allow Apalache to check them?". Apalache required manually expressing the `ENABLED` predicate used in liveness properties that depended on fairness. We iteratively expanded the method for rewriting actions into their `ENABLED` predicates using case studies. Apalache was then able to check the rewritten liveness properties in all cases.

We were able to successfully apply an initial set of rewrite rules from literature to the traffic light example. Of note, since the formula for `ENABLED` used by fairness adds the non-stuttering condition, we included this assertion in the `ENABLED` predicate of each case study. However, for the traffic light and EWD840 case studies, leaving this condition out still yielded a correct rewrite, while reducing complexity.

The EWD840 case revealed the first major issue, of verifying the non-stuttering condition when the action consisted of a disjunction of separate sub-actions. Including the non-stuttering condition at the top level rather than in each of the sub-actions meant that the updated values of the variables went out of scope. We found that distributing this condition into every sub-action was a valid expansion of the rewrite rules.

The bcastFolklore case revealed that the rules from literature, combined with distributing the non-stuttering condition, could not be successfully applied to a conjunction of a sub-action that used a `\E` for assignment, together with a disjunction of sub-actions that required access to the value of this assignment. We proposed placing the entire rest of the expression within the `\E` block to ensure the assigned value was accessible and consistent. Furthermore, in this case, removing the non-stuttering condition produced an incorrect rewrite.

8.2 Interpretation

8.2.1 Inclusion of Non-Stuttering Condition

Conjoining the non-stuttering condition `vars # vars'` to the action being rewritten made the task more complex, but was ultimately found to be necessary. In Sections 5 and 6, removing these conditions still yielded a correct rewrite by our metrics, potentially indicating that it was unnecessary. In our opinion, this is because the new values of the variables these transitions assign are always different from the previous values. Thus, it is not possible for an action to be taken while leaving all variables unchanged. However, Section 7 shows that there do exist cases where omitting the non-stuttering condition causes our metric to report a failure. We thus still consider its inclusion in the two prior cases to be instructive.

8.2.2 Synthesis of Proposed General Method

We now move to finding a pattern in the modifications we made, with the aim of proposing a more general approach, which could be considered empirically in further research. Our aim is to transform an action into a formula that is true if and only if the variables in the current state fulfill guard conditions, and there exists an assignment of new values of the variables that fulfills constraints on them. However, we can not simply place a statement of the following form at the top of the expression:

```
\E vars_new : (rest of expression)
```

This is due TLC and Apalache both rejecting unbounded `\E`. The statement must therefore be placed closer to where the new value of the variable is assigned. In cases where the assignment is to only one potential value, such as the traffic light, any use of the new variable can instead be replaced with its assignment. We identify two problematic cases, use of a variable at a less deep level of conjunction/disjunction nesting than where it is assigned (EWD840), and choosing the value of a variable from a set rather than directly assigning it (bcastFolklore).

The first issue was resolved by transforming the action in EWD840 using the following pattern:

```
(Assignment1 \/ Assignment2) /\ AssignmentGuard <=>
(Assignment1 /\ AssignmentGuard) \/ (Assignment2 /\ AssignmentGuard)
```

This is strikingly similar to a transformation to DNF. As established in Section 6, the two `Assignments` can be thought of as separate independent sub-actions due to the disjunction, and it is thus safe to consider them independently. `AssignmentGuard` (the non-stuttering condition) uses variables assigned in `Assignment1` or `Assignment2`, and thus must be on the same level as them or deeper. If it were to only use the prior values of variables, this change would not be necessary.

The bcastFolklore case study augments our understanding of the necessary transformations, due to use of `\E`, and its structure as a conjunction of an assignment `Assignment1`, dependent sub-actions with their own assignments (`DepAssignment1, ...`), which depend on the values of the top level assignment, and the non-stuttering condition `DepAssignmentGuard`. The structure of the action is then:

```
/\  Assignment1
/\  \/  DepAssignment1
    ...
    \/  DepAssignmentN
/\ DepAssignmentGuard
```

The transformation we applied combined the non-stuttering distribution insight from EWD840 with adding `Assignment1` into scope for every dependent assignment, yielding our result:

```
/\  Assignment1Guard
/\  \E Assignment1Vars \in Set :
    \/ (DepAssignment1 /\ DepAssignmentGuard)
    \/ ...
    \/ (DepAssignmentN /\ DepAssignmentGuard)
```

Because `Assignment1` does not use new variable assignments from sub-actions placed at deeper levels, it was able to remain on a higher level. If it did, it would instead need to be distributed to each disjunction separately, in the same way as the non-stuttering condition is. However, such a situation can never occur, because TLC requires that new values of variables are assigned before they are used [11, sec. 14.2.6]. Thus, this form of circular dependency is impossible in specifications written for TLC, and assignments placed earlier in an action never require access to later assignments.

We therefore propose that, in general, a rewrite of a TLC-compliant action could be performed by nesting assignments, such that the initial assignment is at the top level of the conjunction/disjunction tree, and dependent assignments and guard clauses are placed at deeper levels, as was done in the case of the `\E` binding. If multiple assignments are combined in a disjunction, then dependent expressions will need to be duplicated, as was done to the non-stuttering condition in EWD840. We suggest applying the proposed method to further case studies to verify its correctness.

In summary, based on the specific modifications we made to the rewrite rules, we propose the reason they were required, and the general solution. We suggest that the distribution of the non-stuttering condition of EWD840 across a disjunction of assignments highlights the need to distribute dependent expressions when facing a disjunction of assignments. We also note that assignments can be dependent on other assignments, but never circularly, and thus can be nested. Finally, if a `\E` binding is used, the new value of the variable chosen should remain consistent, thus requiring nesting of subsequent expressions. Given that direct `=` assignment and `\E` assignment are the only methods of assignment accepted by TLC, we believe we have considered conjunctions, disjunctions, and every available method of variable assignment.

8.2.3 Implications

The research expands Offtermatt et al.'s [9, p. 25-26] suggested method, with the synthesis done in the above subsection generalizing the results of the case studies, improving understanding of Apalache's support of fairness. The case studies themselves present potential expansions of the rewrite rules, which apply to their specific actions, and those of a similar form. However, based on the identified root cause of the issues, and our solutions, we propose a more general nesting-based approach, which leads to less duplication than using DNF, and is possible due to TLC forbidding circular assignment dependencies. If Apalache's support for `ENABLED` is not expanded, the improvements we find across the case studies can be used to model check a greater fraction of specifications that use fairness conditions. The general method we propose in Section 8.2.2 is expected to be applicable in more cases, but has not yet been verified empirically. Use of Apalache for model checking over pure TLC is valuable due to Apalache's superior performance in some cases. To avoid any potential human error or uncaught issues with our approach, we do suggest confirming that the manually specified `ENABLED` is equivalent by using TLC to model check the bi-implication on a smaller problem size, and then Apalache for larger problem sizes.

9 Conclusions and Future Work

9.1 Conclusions

Our research answers the research question "How can liveness properties with fairness conditions in TLA+ be rewritten to allow Apalache to check them?". We propose that actions can be rewritten into their `ENABLED` predicates, which can then be used to manually express fairness, by nesting the conjunctions and disjunctions of the original action depending on the order of dependencies between expressions, combined with distributing expressions over disjunctions of assignments.

Liveness properties that use fairness are unsupported by Apalache due to being unable to verify if a transition is possible in a given state, which is encompassed in the `ENABLED` predicate. The specific aspect of interest is assignment of new values to variables during a transition, as these new values can then be used in other parts of the expression. Based on the modifications we performed to an initial set of simple rewrite rules, we propose a general method to create an `ENABLED` predicate from

an action, thus answering the research question. The EWD840 case study shows the need to distribute expressions that depend on what new values variables are assigned across a disjunction. The bcastFolk case study highlights how assignments that are dependent on previous assignments should be nested. These insights can be used to rewrite fairness conditions in the absence of official support for `ENABLED` predicates by Apalache.

Our proposed synthesized method relies of increasing the level of nesting when a new dependency is introduced. We add a level of nesting when a `\E` predicate is used. We also distribute dependent expressions over disjunctions of assignments. We suggest applying this nesting-based approach to further case studies, in order to confirm its soundness.

9.2 Limitations

The notable limitations of our research are a lack of verification of the synthesized method developed in the Discussion, the metric we use accepting formulas that are not logically equivalent but behaviorally equivalent, our focus on a small subset of TLA+ through practical case study choice, and lack of access to industrial TLA+ specifications.

The paper's main limitation is a lack of strong empirical or formal verification of the ultimately proposed approach. The complete method is a synthesis of the case study results, rather than the subject of the study. Our correctness metrics were only applied to the method iteratively developed in the case studies. That method is itself limited, due to only being expanded with the help of two case studies rather than through a formal approach, and therefore likely not generalizing. We thus propose a general but unverified synthesized rewrite method, and a method that was developed from the case studies, but is not general.

Additionally, the metric we used for deciding if an `ENABLED` rewrite was equivalent to the original relies on TLC model checking, which meant that it could only show that the two formulas produce the same result for every state of a given model. This differs from the expressions being actually logically equivalent, as was seen in EWD840, where removing the non-stuttering condition did not change the result. However, no stronger metric was found, as developing a formal proof of equivalence was challenging due to TLC's `ENABLED` being evaluated based on its internal program state, and not from a logical formula.

We chose to focus on a subset of what is possible to express in TLA+. This was primarily due to our choice of metric, requiring us to use TLC to verify a rewrite as behaviorally equivalent. Thus, formulas not accepted by TLC were not considered, even if Apalache would potentially accept them.

Our case studies were simple compared to specifications used in industry, which meant we did not fully consider the additional potential complexity of commercial systems. These industry specifications were however closed-source, making them inaccessible to us. Use of real-world case studies meant we also skipped aspects of TLA+ not used by engineers to specify systems in practice.

9.3 Future Work

For future work, we would primarily suggest applying a formal approach to the problem, based on the theory of TLA+ and LTLs, and applying proofs rather than empirical verification.

The method we synthesize in Section 8.2.2 has not yet been verified in any way, and thus could be investigated by attempting to apply it to case studies with a similar approach as this paper. The method developed iteratively through the case studies could be applied to other examples, such as ones with increasing nesting complexity, to verify if it requires further expansion. Examples of other potential case studies that use fairness conditions are bcastByz, which is the Byzantine version of bcastFolklore, Paxos consensus, or MongoDB Raft.

The most user-friendly approach to supporting liveness properties would be an automatic translation process. Such a feature could be added directly to Apalache by the developers. Absent that, an automated static translation approach could be developed, making manual rewriting unnecessary, and allowing for verification of the method by automatically applying it to a larger set of case studies.

10 Responsible Research

This section presents the ethical implications of our research, its reproducibility, and our statement on LLM use. We explore the potential negative implications of use of the proposed unverified method in model checking real-world systems, provide further details on how our results can be reproduced, and declare that large language models were not used in our research.

10.1 Research Ethics

Our research advances knowledge of formal verification approaches, which are used to improve reliability of safety-critical software and hardware systems, making the ethical impact overall positive. The largest issue is the correctness of the results. Given the use of formal methods in verifying safety-critical systems, any imprecision may result in real harm. Model checking is also especially relevant to distributed systems such as blockchain platforms, where an error could have financial consequences. Thus, if the rewrite rules found are not sound, and they are used in practice, this is likely to cause actual harm. Notably, the rewrite concerns only liveness properties, and safety properties are unaffected, reducing the effect of an error. We believe the overall risk is minimal, because users can verify the correctness of the rewrite with TLC on a smaller problem size. They can then apply the rewrite, and use Apalache to model check their specification with a larger problem size more efficiently. However we still emphasize that our findings are mainly of theoretical interest, and should not be directly applied to verification of real-world systems, where engineers should instead wait for the Apalache developers to officially add support for fairness.

10.2 Reproducible Research

The work is reproducible, because the original specifications and their rewritten versions are publicly available. Links to the original specifications can be found in the References section. The relevant files from these specifications are released alongside our rewritten predicates, bi-implications, and model checker configuration files on GitHub at https://github.com/sage-systems/liveness-rewriting. Researchers can access and modify the cases themselves, and verify results through model checking. We comply with the licenses these specifications are provided under by providing attribution and copies of the relevant license alongside each specification we use. Instructions on the parameters and commands used for running Apalache and TLC are contained within the README.md files in each subfolder. As was mentioned in Section 4.1.4, we used Apalache version 0.57.0, and TLC version 2026.05.18.174321, accessed via the official TLA+ for Visual Studio Code extension version 2026.5.181751.

10.3 Statement on AI/LLM Use

Large language models were not used during research, for specification writing, or for writing the thesis. All work was performed by humans.

A Traffic Light Rewrite

A.1 Original Action

```
SwitchToGreen_Guard ==
    /\ ~isGreen
    /\ requestedGreen

SwitchToGreen_Effect ==
    /\ isGreen' = TRUE
    /\ requestedGreen' = FALSE

SwitchToGreen ==
    SwitchToGreen_Guard /\ SwitchToGreen_Effect
```

A.2 Rewritten Action

```
ManualEnabled == ~isGreen /\ requestedGreen /\ vars # <<TRUE, FALSE>>
```

B EWD840 Rewrite

B.1 Original Action

```
InitiateProbe ==
    /\ tpos = 0
    /\ tcolor = "black" \/ color[0] = "black"
    /\ tpos' = N-1
    /\ tcolor' = "white"
    /\ active' = active
    /\ color' = [color EXCEPT ![0] = "white"]

PassToken(i) ==
    /\ tpos = i
    /\ ~ active[i] \/ color[i] = "black" \/ tcolor = "black"
    /\ tpos' = i-1
    /\ tcolor' = IF color[i] = "black" THEN "black" ELSE tcolor
    /\ active' = active
    /\ color' = [color EXCEPT ![i] = "white"]

System == InitiateProbe \/ \E i \in Node \ {0} : PassToken(i)
```

B.2 Considered Action

```
<<System>>_vars ==
    \/ InitiateProbe
    \/ \E i \in Node \ {0} : PassToken(i)
    /\ (vars # vars')
```

B.3 Rewritten Action

```
ManualEnabledInitiateProbe ==
    /\ tpos = 0
    /\ tcolor = "black" \/ color[0] = "black"
    /\ vars # <<active, [color EXCEPT ![0] = "white"], N-1, "white">>

ManualEnabledPassTokens ==
    \E i \in Node \ {0} :
        /\ tpos = i
        /\ ~ active[i] \/ color[i] = "black" \/ tcolor = "black"
        /\ vars # <<active, [color EXCEPT ![i] = "white"], i-1,
            IF color[i] = "black" THEN "black" ELSE tcolor>>

ManualEnabledSystem == ManualEnabledInitiateProbe \/ ManualEnabledPassTokens
```

C Folklore Reliable Broadcast Rewrite

C.1 Original Action

```
Receive(self) ==
    /\ pc[self] # "CR"
    /\ \E msgs \in SUBSET (Proc \times M):
        /\ msgs \subseteq sent
        /\ rcvd[self] \subseteq msgs
        /\ rcvd' = [rcvd EXCEPT ![self] = msgs ]

UponV1(self) ==
    /\ pc[self] = "V1"
    /\ pc' = [pc EXCEPT ![self] = "AC"]
    /\ sent' = sent \cup { <<self, "ECHO">> }
    /\ nCrashed' = nCrashed
    /\ Corr' = Corr

UponAccept(self) ==
    /\ (pc[self] = "V0" \/ pc[self] = "V1")
    /\ rcvd'[self] # {}
    /\ pc' = [pc EXCEPT ![self] = "AC"]
    /\ sent' = sent \cup { <<self, "ECHO">> }
    /\ nCrashed' = nCrashed
    /\ Corr' = Corr

OriginalWF == WF_vars(\E self \in Corr:
    /\ Receive(self)
    /\  \/ UponV1(self)
        \/ UponAccept(self)
        \/ UNCHANGED << pc, sent, nCrashed, Corr >> )
```

C.2 Rewritten Action

```
ManualEnabled ==
    \E self \in Corr:
        /\ pc[self] # "CR"
        /\ \E msgs \in SUBSET (Proc \times M):
            /\ msgs \subseteq sent
            /\ rcvd[self] \subseteq msgs
            /\  \/  /\  pc[self] = "V1"
                    /\  vars # << [pc EXCEPT ![self] = "AC"],[rcvd EXCEPT ![self] = msgs ],
                        sent \cup { <<self, "ECHO">> }, nCrashed, Corr >>
                \/  /\  (pc[self] = "V0" \/ pc[self] = "V1")
                    /\  [rcvd EXCEPT ![self] = msgs ][self] # {}
                    /\  vars # << [pc EXCEPT ![self] = "AC"],[rcvd EXCEPT ![self] = msgs ],
                        sent \cup { <<self, "ECHO">> }, nCrashed, Corr >>
                \/ vars # << pc, [rcvd EXCEPT ![self] = msgs ], sent, nCrashed, Corr >>
```

References

[1] E. W. Dijkstra, "The humble programmer", *Commun. ACM*, vol. 15, no. 10, pp. 859–866, Oct. 1972, ISSN: 0001-0782. DOI: 10.1145/355604.361591. [Online]. Available: https://doi.org/10.1145/355604.361591.

[2] E. Verhulst, R. T. Boute, J. M. S. Faria, B. H. Sputh, and V. Mezhuyev, *Formal Development of a Network-Centric RTOS: software engineering for reliable embedded systems*. Springer Science & Business Media, 2011.

[3] R. Bögli, L. Lerena, C. Tsigkanos, and T. Kehrer, "A systematic literature review on a decade of industrial tla+ practice", in *Integrated Formal Methods*, N. Kosmatov and L. Kovács, Eds., Cham: Springer Nature Switzerland, 2025, pp. 24–34, ISBN: 978-3-031-76554-4.

[4] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications", in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, ser. CHARME '99, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 54–66, ISBN: 3540665595.

[5] I. Konnov, J. Kukovec, and T.-H. Tran, "Tla+ model checking made symbolic", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360549. [Online]. Available: https://doi.org/10.1145/3360549.

[6] I. Konnov, M. Kuppe, and S. Merz, "Specification and verification with the tla+ trifecta: Tlc, apalache, and tlaps", in *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*, Rhodes, Greece: Springer-Verlag, 2022, pp. 88–105, ISBN: 978-3-031-19848-9. DOI: 10.1007/978-3-031-19849-6_6. [Online]. Available: https://doi.org/10.1007/978-3-031-19849-6_6.

[7] I. Konnov and P. Offtermatt. "Pdr-017: Checking temporal properties". (Apr. 2022), [Online]. Available: https://apalache-mc.org/docs/adr/017pdr-temporal.html.

[8] "Supported features - apalache documentation". (May 2026), [Online]. Available: https://apalache-mc.org/docs/apalache/features.html.

[9] P. Offtermatt, J. Kukovec, and I. Konnov, "Extending apalache to symbolically reason about temporal properties of tla+", TLA+ Conference, 2022. [Online]. Available: https://conf.tlapl.us/2022/PhilipOfftermattTemporalPropsApalache.pdf.

[10] P. Offtermatt. "Temporal properties and counterexamples". (Feb. 2024), [Online]. Available: https://apalache-mc.org/docs/tutorials/temporal-properties.html.

[11] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003, ISBN: 9780321143068. [Online]. Available: https://books.google.nl/books?id=SeRQAAAAMAAJ.

[12] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking", *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, pp. 160–177, 2002, FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop), ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)80410-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104804109.

[13] L. Lamport, M. A. Kuppe, S. Merz, *et al.*, *TLA+ Examples*. [Online]. Available: https://github.com/tlaplus/Examples.

[14] E. W. Dijkstra, W. Feijen, and A. van Gasteren, "Derivation of a termination detection algorithm for distributed computations", *Information Processing Letters*, vol. 16, no. 5, pp. 217–219, 1983, ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(83)90092-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019083900923.

[15] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Tutorial on parameterized model checking of fault-tolerant distributed algorithms", in Vienna University of Technology, Jun. 2014, pp. 122–171, ISBN: 978-3-319-07316-3. DOI: 10.1007/978-3-319-07317-0_4.

[16] S. Merz, *Termination detection in a ring*. [Online]. Available: https://github.com/tlaplus/Examples/tree/master/specifications/ewd840.

[17] T. H. Tran, I. Konnov, and J. Widder, *bcastFolklore*. [Online]. Available: https://github.com/tlaplus/Examples/tree/master/specifications/bcastFolklore.