Verification of the Busy-Forbidden Protocol

(using an Extension of the Cones and Foci Proof Framework)

P. H. M. van Spaendonck(✉)

Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, Netherlands
P.H.M.v.Spaendonck@tue.nl

Abstract. The busy-forbidden protocol is a new readers-writer lock with no resource contention between readers, which allows it to outperform other locks. For its verification, specifications of its implementation and its less complex external behavior are provided by the original authors but are only proven equivalent for up to 7 threads.

We provide a general equivalence proof using the cones and foci proof framework, which rephrases whether two specifications are branching bisimilar as six properties on the data objects of the specifications. We provide an extension of this framework consisting of four additional properties and prove that when the additional properties hold, the two systems are divergence-preserving branching bisimilar, a stronger version of the aforementioned relation that also distinguishes livelocks.

Keywords: cones and foci proof framework · divergence-preserving branching bisimulation · process algebra · protocol verification · readers-writer lock

1 Introduction

The readers-writer lock problem is a concurrency problem introduced and solved by Courtois et al. [5]. The problem requires a synchronisation protocol that provides safe access to both a shared readers section, which can be used simultaneously by any number of threads, as well as an exclusive writer section, which can not be used by more than one thread at any given time and only when the readers section is not in use.

In [9], Groote et al. introduce a new readers-writer lock called the busy-forbidden protocol. This locking protocol is of particular interest as it has no resource contention between readers, and therefore provides a significant speedup over other locks when having high readers section workloads.

To ensure the correctness of the protocol, the authors give process algebraic specifications of both the implementation of the new algorithm as well as a

This publication is part of the PVSR project (with project number 17933) of the MasCot research programme which is financed by the Dutch Research Council (NWO).

Published by Springer Nature Switzerland AG 2023
H. Hojjat and E. Ábrahám (Eds.): FSEN 2023, LNCS 14155, pp. 126–141, 2023.
https://doi.org/10.1007/978-3-031-42441-0_10

specification of its external behavior. The authors applied model checking and proved the implementation and external behavior equivalent for up to 7 threads using the mCRL2 toolset [4], but they were unable to do this for more concurrent threads due to the statespace of the implementation becoming too large.

But as readers-writer locks often use a large number of concurrent threads, a general correctness proof for the busy-forbidden protocol is desired. We opt to prove the process algebraic specifications of the implementation and external behavior to be equivalent. The advantage of this technique over contract-based approaches, such as Floyd-Hoare logic [12], and its extension for parallel composed systems by Owicki and Gries [15,16], is that the much smaller equivalent model can also be used for the modeling and verification of systems built on top of the busy-forbidden protocol. We consider this a significant advantage, as this is the typical use-case for readers-writer locks, e.g. the parallel term library which the protocol was originally designed for.

We prove the equivalence of the implementation and its external behavior by using the cones and foci proof framework, originally proposed in [11] by Groote and Springintveld and later generalized by Fokkink et al. in [6]. This framework simplifies the often complex and cumbersome branching bisimulation proof by reducing it to a small set of propositions on the data objects occurring in the implementation and specification. If these propositions are shown to hold, it follows that the two systems are equivalent modulo branching bisimulation.

The proof framework has already been used in several case studies to prove implementation and specification models equivalent, such as the verification of the 1-bit sliding window protocol in [2], a complex leader election protocol in [7], and a part of the IEEE P1394 high-speed bus protocol [1] in [17].

Since the equivalence relation proven by the cones and foci proof framework does not distinguish livelock, we first provide an extension to the framework such that it can also be used to prove equivalence modulo divergence-preserving branching bisimulation. This relation is a stronger version of branching bisimulation that does distinguish livelocks [8]. Our extension provides four additional propositions on the data objects in the implementation and specification models, that, when shown to also hold, imply the equivalence of the two processes modulo divergence-preserving branching bisimulation. We give a soundness proof of this extension and use it to prove the equivalence of implementation and specification of the busy-forbidden protocol.

2 The Busy-Forbidden Protocol

We first discuss the busy-forbidden protocol. An overview of its implementation using pseudocode is given in Table 1. The `enter_-` and `leave_shared` functions are used to have a thread p enter or leave the readers section. Similarly, `enter_-` and `leave_exclusive` provide functionality for safe access to the writer section.

The protocol uses two binary flags per thread and a single mutex. The first flag, the *busy* flag, indicates that a thread is either working or going to work inside of the readers section. The second flag, the *forbidden* flag, indicates that

a thread is not allowed to enter the readers section. All flags are initially set to *false*. The mutex, called *mutex*, enforces exclusive access to the writer section.

Table 1. Pseudocode description of the busy-forbidden protocol

`enter_shared`(*thread p*) :
  p.busy := *true*;
  while *p.forbidden*
  p.busy := *false*;
  if *mutex.timed_lock*()
  mutex.unlock();
  p.busy := *true*;
	`enter_exclusive`(*thread p*) :
	  mutex.lock();
	  while exists *thread q* **with**
	  ¬*q.forbidden*
	  select *thread r*
	  r.forbidden := *true*;
	  if *r.busy* **or sometimes**
	  r.forbidden := *false*;

`leave_shared`(*thread p*) :
  p.busy := *false*;
	`leave_exclusive`(*thread p*) :
	  while exists *thread q* **with**
	  q.forbidden
	  select *thread r*
	  usually do
	  r.forbidden := *false*;
	  sometimes do
	  r.forbidden := *true*
	  mutex.unlock();

When entering the readers section, a thread sets its *busy* flag and enters iff its *forbidden* flag is *false*. If the *forbidden* flag is *true*, the *busy* flag is set back to *false* to avoid deadlock and the process is repeated again. To reduce resource contention on the flags, a *mutex.timed_lock*() can be used without altering the externally visible behavior of the protocol [9]. Upon leaving the readers section, the thread sets its *busy* flag back to false.

A thread that wants to enter the writer section must first acquire the mutex. This ensures that no other thread can be in the writer section simultaneously and that only the given thread is altering the *forbidden* flags. Once the mutex has been acquired, the thread sets the *forbidden* flag of each thread, but will immediately undo this if the *busy* flag of the same thread is *true*. To prevent a thread that is acquiring the writer section from locking out some reader threads while still waiting for others to leave the readers section, random *forbidden* flags can sometimes be set back to *false*. The writer section is entered once all *forbidden* flags are *true*. Upon leaving, all *forbidden* flags are set back to false and the mutex is released. During this, random *forbidden* flags can be set back to *true*. This prevents each iteration that occurs while leaving, from becoming externally visible and significantly reduces the number of states in the external specification.

The externally visible behavior of the protocol is given in Fig. 1 and, as we will prove later, provides an equivalent overview of how threads interact

via the protocol. Individual threads move from node to node. Transition labels ending with `call` represent the identically named function being called by a thread moving across, and those ending with `return` represent those function calls terminating. All transitions not labeled as such represent some sequence of internal calculations that occurs during these function calls. Transitions labeled with a guard, i.e. starting with **if**, only allow a thread to progress if the given condition is met.

The *Free* node represents a thread not interacting with the protocol and being outside of any section. Each thread initially starts out in this node. The *Shared* and *Exclusive* nodes represent the readers and writer sections, respectively.

A thread starting to acquire the readers lock enters the *EnterShared* (*ES*) node. The thread stays in the *ES* node as long as its *forbidden* flag is *true*. As repeatedly checking the flag is discouraged through the *timed_lock* call, the internal loop is labeled as *improbable*. When the *forbidden* flag is evaluated to *false*, the thread moves to the *LockedOffExclusive* (*LOE*) node. After this, it is no longer possible for any other thread to enter the writer section until the readers section is completely freed. The *LeaveShared* (*LS*) node represents a thread leaving this section.

When a thread tries to acquire the writer lock, it enters the *EnterExclusive* (*EE*) node. Once the thread acquires the *mutex* variable, it will move to the *SetAllForbidden* (*SAF*) node and it will not be possible for any other thread to acquire the writer lock before it is released by this thread. The loop in the *SAF* node represents a *forbidden* flag being set back to *false*; this transition is labeled as *improbable* as this only rarely occurs. Once the last *busy* flag is evaluated to *false*, exclusive access is attained and the thread will move to the *LockedOutShared* (*LOS*) node before officially terminating the function call.

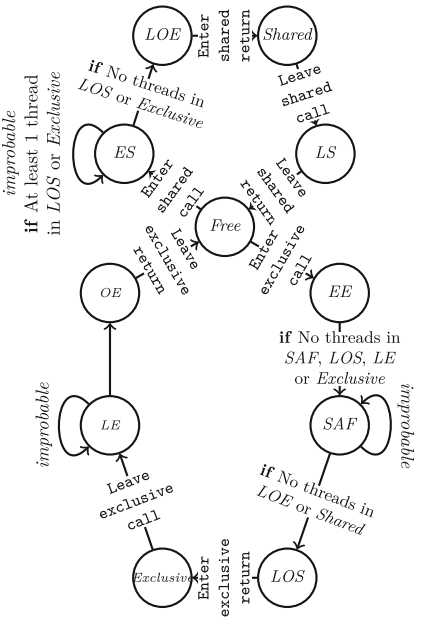

Fig. 1. The external behaviour

When the thread starts releasing the writer lock, it enters the *LeavingExclusive* (*LE*) node. Similar to the *SAF* node, a thread within the *LE* node can repeatedly turn the *forbidden* flag off and on again, thus never fully opening up the readers section. Because a *forbidden* flag is only very rarely set back to *true* when releasing the lock, this transition is also labeled as *improbable*. Once the last *forbidden* flag is set to *false*, this is no longer possible and the thread moves to the *OpenedExclusive*

(*OE*) node, after which it will officially terminate the function call and move back to the *Free* node.

We can use the model of the external behavior to reason about certain safety properties. For example, from the guarded transitions from *ES* to *LOE* and from *SAF* to *LOS*, we can quickly see that the *Shared* and *Exclusive* sections can not be populated simultaneously, as they require the other respective section to be empty. The guarded transition from *EE* to *SAF* also assures that only a single thread can be present in the *Exclusive* section at any given time.

3 Linear Process Equations

Both the implementation of the pseudocode shown in Table 1 and the external behavior have been modeled in the mCRL2 language [10]. The mCRL2 language is based on the Algebra of Communicating Processes [3] and Calculus of Communicating Processes [14].

The mCRL2 language models processes using a combination of states and actions. States represent a collection of internal values that are used to calculate which actions can occur and what the resulting state will be. Actions represent any sort of atomic event such as calling a function, or setting or reading a flag. An action consists of a label and a possible set of data parameters, e.g. the action $lock(p)$ has *lock* as the label and p as the data parameter. Parameters can be of varying types such as booleans, algebraic data types, and mappings. The exact data types used within the busy-forbidden models are given later.

A special action τ, the so-called hidden or internal action, is used to represent an action that is externally not directly visible. We use distinct action labels for internal actions to be able to easily distinguish between them. We explicitly state which actions should be considered to be τ actions.

We require all process algebraic equations to be in a clustered linear form, see Definition 1. This form specifies for each action when it can occur and what the resulting state will be. The $\sum_{e:S}$ operator models the application of the non-deterministic choice operator $+$ over all elements in some set S. We also allow process equations in which the $\sum$ operators are split into separate smaller $\sum$ operators and individual $+$ operators.

Since the cones and foci proof framework concerns itself only with the actions that are enabled in a single given state, the clustered normal form becomes especially useful, as we can directly infer for any given state if an action is enabled and what the resulting state will be. In [19], Usenko shows that any mCRL2 specification can be transformed into a clustered linear process equation.

Definition 1. A clustered linear process equation (LPE) is a process specification of the form:

$$X(d{:}D) = \sum_{a:Act} \sum_{e_a:E_a} c_a(d, e_a) \rightarrow a(f_a(d, e_a)) \cdot X(g_a(d, e_a)),$$

where D is the set of states, Act is the set of action labels including τ, E_a is an indexed set of all data types that need to be considered for label a, the boolean

function $c_a(d, e_a)$ specifies when the action a with parameters resulting from the function $f_a(d, e_a)$ is enabled in state d, and $g_a(d, e_a)$ gives the resulting state from taking this action from state d.

Often we end up in a situation in which the set of states D also contains unreachable states. As we are only interested in the reachable states, we introduce the notion of an invariant in Definition 2. An invariant is a predicate on states in an LPE such that when it holds for a given state $d{:}D$, it also holds for all subsequent states.

Definition 2. Given a clustered *LPE* X as per Definition 1. A predicate $\mathcal{I}$ on the set of states D is called an invariant iff the following holds: for all $a{:}Act, d{:}D$ and $e_a{:}E_a$,

$$\mathcal{I}(d) \wedge c_a(d, e_a) \Rightarrow \mathcal{I}(g_a(d, e_a))$$

4 Equivalence and the Cones and Foci Proof Framework

As stated before, we prove the model of the implementation and the specification of the busy-forbidden protocol equivalent modulo divergence-preserving branching bisimulation. We define this equivalence relation in Definition 4, which is based on the definitions used in [13] and has been adapted to work with process equations instead of transition systems. In Definition 3, we provide some syntactic glue to make this shift between labeled transition systems and clustered LPEs more intuitive.

Definition 3. Given a clustered LPE as per Definition 1, states $d, d' \in D$, and action l, we define the following relations:

- $d \xrightarrow{l} d'$ iff there is an action a with an associated data type e_a such that $l = a(f_a(d, e_a))$, the condition $c_a(d, e_a)$ holds, and $g_a(d, e_a) = d'$.
- $d \xrightarrow{l}{}^* d'$ iff there is a finite sequence of states $d_0, \ldots, d_k$ such that $d_0 = d$, $d_k = d'$ and for all $0 \leq i < k$ we have $d_i \xrightarrow{l} d_{i+1}$.

Definition 4. Given two clustered LPEs as per Definition 1 with sets of states D and D'. A relation R on the states $D \times D'$ is a divergence-preserving branching bisimulation iff the following conditions for all states $s \in D$, $t \in D'$, and actions $l \in Act$ hold:

(B_1) If sRt and $s \xrightarrow{l} s'$ for some state $s' \in D$, then either $l = \tau$ and $s'Rt$, or there are states $t', t'' \in D'$ such that $t \xrightarrow{\tau}{}^* t' \xrightarrow{l} t''$, sRt', and $s'Rt''$.

(B_2) If sRt and $t \xrightarrow{l} t'$ for some state $t' \in D'$, then either $l = \tau$ and sRt', or there are states $s', s'' \in D$ such that $s \xrightarrow{\tau}{}^* s' \xrightarrow{l} s''$, $s'Rt$, and $s''Rt$.

(D_1) If $sR\,t$ and there is an infinite sequence of states $(s_n)_{n \in \mathbb{N}}$ such that $s = s_0$, and $s_k \xrightarrow{\tau} s_{k+1}$ and $s_k R\,t$ for all $k \in \mathbb{N}$, then there is a state $t' \in D'$ such that $t \xrightarrow{\tau} t'$, and $s_k R\,t'$ for some $k \in \mathbb{N}$.

(D_2) If $sR\,t$ and there is an infinite sequence of states $(t_n)_{n\in\mathbb{N}}$ such that $t = t_0$, and $t_k \xrightarrow{\tau} t_{k+1}$ and $sR\,t_k$ for all $k \in \mathbb{N}$, then there is a state $s' \in D$ such that $s \xrightarrow{\tau} s'$, and $s'R\,t_k$ for some $k \in \mathbb{N}$.

Two clustered LPEs with respective initial states d_0 and d_0' are *divergence-preserving branching bisimilar* iff there is a divergence-preserving branching bisimulation R such that $d_0 R\, d_0'$.

Note that in (divergence-preserving) branching bisimulation, τ-actions are said to be externally visible iff their begin- and endpoint are not equivalent.

In [11], it is noted that in communicating systems, equivalent states often have a "cone-like" structure as is shown in Fig. 2. In this figure, equivalent states are grouped together in the *cone* C. In the *focus point* state *fc*, all externally visible actions of said cone, i.e. a and b, are enabled. For all other states in which not all externally visible actions are simultaneously enabled, such as d or the states along the edges, there is always a path of *internal actions*, i.e. τ actions within the cone, that ends in the state *fc*. We show one such path for the state d, using the dashed arrows.

If a given system consists of such "cones", the cones and foci proof framework can be used to prove equivalence. To do so, we must provide a *state mapping* $h : D \to D'$ that maps states in the implementation to their equivalent state in the specification, a *focus condition* $FC : D \to \mathbb{B}$ that indicates if a state should be considered a focus point, i.e. all externally observable actions are enabled, and a well-founded ordering $<_M$ on D that orders states by their distance to a focus point. We must then prove that a small set of requirements are met by the LPEs and the provided *state mapping, focus condition* and ordering.

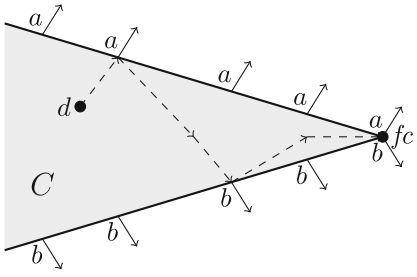

Fig. 2. A cone C with focus point *fc*

Any τ action in the implementation that does not leave a cone, i.e. the state mapping h maps begin- and endpoint to the same state, is renamed to *int* (short for *internal* action). This allows us to easily distinguish between τ actions that are externally observable, i.e. that are preserved in our specification, and those that are not. While an *int* action is considered a τ action, we exclude them from the set of actions Act.

In Theorem 1, we extend the proof framework towards divergence-preserving branching bisimulation with a labeling p on cones that labels cones as either divergent (Δ) or non-divergent (∇), and four additional requirements on the LPEs. The divergent τ-loops in the specification, i.e. a τ transition with the same begin- and endpoint, are renamed to *int* to relate these to the divergent internal behavior in the implementation, i.e. repeatable paths of *int* actions.

Theorem 1. Consider a clustered linear process equation of an implementation with initial state d_0 and some invariant $\mathcal{I}$ that holds in d_0,

$$X(d{:}D) = \sum_{a:Act\cup\{int\}} \sum_{e_a:E_a} c_a(d,e_a) \to a(f_a(d,e_a)) \cdot X(g_a(d,e_a)),$$

and a clustered linear process equation of a specification with initial state d'_0,

$$Y(d'{:}D') = \sum_{a:Act\cup\{int\}} \sum_{e_a:E_a} c'_a(d',e_a) \to a(f'_a(d',e_a)) \cdot Y(g'_a(d',e_a)).$$

The LPEs X and Y are divergence-preserving branching bismilar if there is a *state mapping* $h : D \to D'$, a *focus condition* $FC : D \to \mathbb{B}$, a well founded ordering $<_M$ on D, and a cone labeling $p : D' \to \{\Delta, \nabla\}$ such that $h(d_0) = d'_0$ and the following requirements hold for all states $d{:}D$ in which invariant $\mathcal{I}$ holds:

I If not a focus point, there is at least one internal step such that the target state is closer to the focus point:

$$(\neg FC(d)) \Rightarrow (\exists e_{int}{:}E_{int}.\ c_{int}(d,e_{int}) \wedge g_{int}(d,e_{int}) <_M d)$$

II For every internal step, the mapping h maps source and target states to the same states in the specification:

$$\forall e_{int}{:}E_{int}.c_{int}(d,e_{int}) \Rightarrow h(d) = h(g_{int}(d,e_{int}))$$

III Every visible action in the specification must be enabled after a finite number of *int* actions for each corresponding focus point: For all $a{:}Act$

$$\forall e_a{:}E_a.(FC(d) \wedge c'_a(h(d),e_a)) \Rightarrow (\exists d_{int}{:}D.d \xrightarrow{int}{}^{*} d_{int} \wedge c_a(d_{int},e_a))$$

IV Every visible action in the implementation must be mimicked in the corresponding state in the specification: For all $a{:}Act$

$$\forall e_a{:}E_a.c_a(d,e_a) \Rightarrow c'_a(h(d),e_a)$$

V Matching actions have matching parameters: For all $a{:}Act$

$$\forall e_a{:}E_a.c_a(d,e_a) \Rightarrow f_a(d,e_a) = f'_a(h(d),e_a)$$

VI For all matching actions in specification and implementation, their endpoints must be related: For all $a{:}Act$

$$\forall e_a{:}E_a.c_a(d,e_a) \Rightarrow h(g_a(d,e_a)) = g'_a(h(d),e_a)$$

I_Δ Any internal action in the specification is part of an *int*-loop:

$$\forall_{e_{int}}{:}E_{int}.c'_{int}(h(d),e_{int}) \Rightarrow g'_{int}(h(d),e_{int}) = h(d)$$

II_Δ The cone labeling indicates whether or not a specification state allows an *int*-loop:

$$p(h(d)) = \Delta \Leftrightarrow (\exists e_{int}{:}E_{int}.\ c'_{int}(h(d), e_{int}))$$

III_Δ A cone is labelled as divergent if and only if it is possible to take an internal action in its focus points:

$$FC(d) \Rightarrow (p(h(d)) = \Delta \Leftrightarrow \exists e_{int}{:}E_{int}.c_{int}(d, e_{int}))$$

IV_Δ All internal transitions within a non-divergent cone must bring us closer to a focus point:

$$\forall e_{int}{:}E_{int}.((p(h(d)) = \nabla \wedge c_{int}(d, e_{int})) \Rightarrow g_{int}(d, e_{int}) <_M d)$$

Proof. We define $R \subseteq D \times D'$ as $\{\langle d, h(d)\rangle \mid d \in D \wedge \mathcal{I}(d)\}$.

Proving that R is a branching bisimulation, i.e. proving conditions B_1 and B_2 from Defininition 4, follows the same general proof structure as is used in both [6], and [11]. We give a concise proof sketch.

Condition B_1. Consider the states $d, d'{:}D$, and label $l{:}Act \cup \{int\}$ such that $d \xrightarrow{a} d'$. As per Requirement II , if $l = int$ then $h(d) = h(d')$. If $l \neq int$, then we have $h(d) \xrightarrow{l} h(d')$ as per Requirement IV , and VI.

Condition B_2. Consider the states $d{:}D, d'_2{:}D'$, and label $l{:}Act \cup \{int\}$ such that $h(d) \xrightarrow{l} d'_2$. If $l = int$, then $h(d'_2) = h(d)$, as per Requirement I_Δ . If $l \neq int$, then there is a state $d_2{:}D$ such that $d \xrightarrow{int}{}^* d_2$ and $FC(d_2)$ as per Requirement I and $<_M$ being well founded. As per Requirements III and VI, there are states $d_3, d_4{:}D$ such that $d \xrightarrow{int}{}^* d_2 \xrightarrow{int}{}^* d_3 \xrightarrow{l} d_4$ and $h(d_4) = d'_2$. From Requirement II follows that all states along the *int* path are related to $h(d)$.

We show that the branching bisimulation R is also divergence-preserving by proving the two remaining conditions.

Condition D_1. Consider the pair $\langle d, h(d)\rangle \in R$ and an infinite sequence $(d_n)_{n\in\mathbb{N}}$ over states in D such that $d_0 = d$ and for any $n \in \mathbb{N}$ we have $h(d_n) = h(d)$ and $d_n \xrightarrow{int} d_{n+1}$. We show that there is some $e_{int}{:}E_{int}$ such that $c'_{int}(h(d), e_{int})$ and $g'_{int}(h(d), e_{int}) = h(d)$. If $h(d)$ is labeled Δ then this directly follows from Requirements I_Δ, and II_Δ.

Assume, for sake of contradiction, that $h(d)$ is labeled as ∇ instead. Since $<_M$ is a well-founded ordering on D, the sequence $(d_n)_{n\in\mathbb{N}}$ contains some minimal element $d_\perp$ such that no other element in the sequence is smaller than $d_\perp$. However, as per Requirement IV_Δ, any outgoing *int* action from $d_\perp$ must have an endpoint that is smaller than $d_\perp$, and thus the state that comes directly after $d_\perp$ in the sequence would have to be smaller, contradicting that $d_\perp$ is the minimal element.

Condition D_2. Consider the pair $\langle d, h(d)\rangle \in R$ and an infinite sequence $(d'_n)_{n\in\mathbb{N}}$ over states in D' such that $d'_0 = h(d)$ and for any $n \in \mathbb{N}$ we have $d\ R\ d'_n$, i.e. $d'_n = h(d)$, and $d'_n \xrightarrow{int} d'_{n+1}$.

Since $h(d)$ allows an `int`-loop, we have $p(h(d)) = \Delta$ as per Requirement II_Δ. If d is not a focus point then this action is enabled as per Requirement I. If d is a focus point then this action is enabled as per Requirement III_Δ , since its corresponding cone is labeled as Δ. Requirement II gives us that the endpoint of this internal action is related to $h(d)$. Thus, if a state in the specification diverges then so do the related states in the implementation.

We thus conclude that the relation R is a divergence-preserving branching bisimulation. □

5 Models of the Specification and the Implementation

We now discuss the models of the specification and implementation of the busy-forbidden protocol, such that we can use the extended proof framework to prove them equivalent in Sect. 6. From here on, we use N to denote the number of concurrent threads and we define $P = \{p_1, \ldots, p_N\}$ to be the set containing all N threads.

The linear process equation of the external behavior of the busy-forbidden protocol is given in Table 9 in the appendix [18]. The set S contains the nodes shown in Fig. 1. Each state in the specification is represented using a mapping s that maps each thread to its current node, with each thread starting in the *Free* node. The set of specification states for N threads is denoted by D'_N. Note that each condition in the specification is the same as the conditions shown in Fig. 1. The *improbable* actions are considered to be *int* actions.

The linear process equation of the implementation is given in Table 10 in the appendix [18]. All non-`typewriter` font actions are considered to be τ actions and *italicized* actions are specifically considered to be *int* actions. The set of implementation states D_N is given in Definition 5. A part of each state consists of N substates, with each substate giving the state of that specific thread. The set of substates is given in Definition 6, in which substates corresponding to the same node are grouped together.

Definition 5. Each state in the linearized process of the busy-forbidden implementation for N threads is defined as the tuple

$$d = \langle d_{p_1}, d_{p_2}, \ldots, d_{p_N}, \mathit{busy}, \mathit{forbidden}, \mathit{mtx} \rangle : D_N, \text{ in which:}$$

- $d_{p_1}, d_{p_2}, \ldots, d_{p_N}$ are the substates of threads 1 through N.
- $\mathit{busy} : P \to \mathbb{B}$ is the mapping that keeps track of all the busy flags, in which $\mathit{busy}(p)$ is the current value of the busy flag of thread p.
- $\mathit{forbidden} : P \to \mathbb{B}$ is the mapping that keeps track of all the forbidden flags in the same way as the *busy* mapping.
- *mtx* is a boolean that indicates whether the mutual exclusion variable *mtx* is locked or unlocked.

Definition 6. The set of substates for each individual process is defined as the union of the following sets:

- $Free = \{Free\}$, $ES = \{ES_1, ES_2, ES_3, ES_4\}$, $LOE = \{LOE\}$, $Shared = \{Shared\}$, $LS = \{LS_1, LS_2\}$, $EE = \{EE\}$, $LOS = \{LOS_1, LOS_2\}$, $Exclusive = \{Exclusive\}$, and $OE = \{OE_1, OE_2\}$,
- $SAF = \{SAF_U | U \subset P\} \cup \{SAF_{p_x,U} | p_x{:}P, U \subset P\} \cup \{SAF^{undo}_{p_x,U} | p_x{:}P, U \subset P\}$,
- and $LE = \{LE_U | U \subseteq P \wedge U \neq \emptyset\}$.

Note that the singleton sets, such as *Free*, contain a single state with the same name as the set and do not contain themselves.

In the initial state of the implementation for N threads, all substates are set to *Free*, *busy* and *forbidden* map each thread p to *false* and *mtx* is set to *false*.

Since the state tuple contains a large number of elements, we use a shorthand notation for writing down the resulting state. All elements which remain the same are not listed and are abbreviated with "*etc.*". A substate or the *mtx* variable being changed in the resulting state is denoted with the "=" operator, where the lefthand side is assigned the value on the righthand side, e.g. $d_p = ES_2$ indicates that the substate of thread p becomes ES_2 in the next state. The function update $f[e \mapsto n]$ specifies that in the next state $f(x)$ equals the new value n if $x \approx e$ and otherwise equals its original value.

We introduce the Invariants 1, 2, and 3. These exclude some unreachable states and show that for any given state, the exact values of *busy*, *forbidden*, and *mtx* can be inferred from just the set of substates, i.e. $d_{p_1}, d_{p_2}, \ldots, d_{p_N}$. In the proof of Invariant 1, we show that the value of *mtx* can be inferred from just the set of substates and that it is not possible to have multiple threads simultaneously present in the set of states fenced off by the mutex operations. We show that the values of the *busy* and *forbidden* flags can also be inferred from just the set of substates in the proofs of Invariants 2 and 3.

The exact proofs for these invariants can be found in the appendix [18]. All of them follow the same general structure. Namely, the actions that result in a thread entering or leaving the given set of states, e.g. B, are the exact same actions that result in the value, e.g. $busy(p)$, being altered. And thus the exact values can be inferred from just the set of substates.

Invariant 1. The following invariant holds in the initial state and all subsequent states of the implementation: Given any state $d{:}D$ as per Definition 5,

$$\exists p{:}P.\ d_p \in M \Leftrightarrow mtx, \text{ and } \forall p_x, p_y{:}P.\ d_{p_x}, d_{p_y} \in M \Rightarrow p_x = p_y,$$

where $M = SAF \cup LOS \cup Exclusive \cup LE \cup \{OE_2\}$.

Invariant 2. The following invariant holds in the initial state and all subsequent states of the implementation: Given any state $d{:}D$ as per Definition 5,

$$\forall p{:}P.d_p \in B \Leftrightarrow busy(p), \text{ where } B = LOE \cup Shared \cup \{ES_1, ES_4, LS_2\}.$$

Invariant 3. The following invariant holds in the initial state and all subsequent states of the implementation: Given any state $d{:}D$ as per Definition 5,

$$\forall p{:}P.forbidden(p) \iff \exists q{:}P.d_q \in F,$$

where $F = LOS \cup Exclusive \cup \{LE_U | U \subset P \wedge p \in U\} \cup \{SAF_U | U \subset P \wedge p \in U\} \cup \{SAF_{p,U} | U \subset P\} \cup \{SAF^{undo}_{p,U} | U \subset P\}$.

6 Correctness of the Busy-Forbidden Protocol

The state mapping, focus condition, state ordering and cone labeling used during the equivalence proof are given in Definitions 7, 8, 9, and 10, respectively. These data objects only need to use substates since the values of the *busy*, *forbidden*, and *mtx* data objects can be directly inferred from the substates in any given state.

Definition 7. We define our state-mapping $h : D_N \rightarrow D'_N$ as follows:

$$h(\langle d_1, d_2, \ldots, d_N, busy, forbidden, mtx\rangle) = s \text{ where } s(p) = h_P(d_p) \text{ for any } p{:}P.$$

The mapping h_P, referred to as the substate-mapping, maps each substate to the specification state with the same name as the set, shown in Definition 6, that it belongs to, e.g. $h_P(ES_3) = ES$ and $h_P(SAF_{\{p_1,p_3,p_4\}}) = SAF$.

Definition 8. We define our focus condition $FC : D_N \rightarrow \mathbb{B}$ as follows:

$$FC(\langle d_{p_1}, d_{p_2}, \ldots, d_{p_N}, busy,\ forbidden,\ mtx\rangle) = \bigwedge_{p_x:P} FC_{p_x}(d_{p_x}),$$

where $FC_{p_x}(d_{p_x}) \stackrel{def}{=} p_x \in \{Free, ES_1, LOE,\ Shared, LS_1, EE, SAF_\emptyset, LOS_1, Exclusive, LE_{\{p_x\}}, OE_1\}$. We refer to the predicate FC_{p_x}, for any given $p_x{:}P$, as the sub-focus condition.

Definition 9. Given two states $d = \langle d_{p_1}, d_{p_2}, \ldots, d_{p_N}, busy, forbidden, mtx\rangle$ and $d' = \langle d'_{p1}, d'_{p2}, \ldots, d'_{pN}, busy', forbidden', mtx'\rangle$, we define the ordering on these states as follows:

$$d <_M d' \stackrel{def}{=} \bigwedge_{p:P} d_p <_p d'_p,$$

where, given some thread $p{:}P$, the ordering $<_p$ on its substates is defined such that only the following holds:

- $ES_1 <_p ES_2 <_p ES_3 <_p ES_4$, $LS_1 <_p LS_2$, $LOS_1 <_p LOS_2$, and $OE_1 <_p OE_2$,
- $SAF_{p_x,U} <_p SAF_U$ iff $p_x \in U$ for any given $U{:}\mathcal{P}(P)$ and $p_x{:}P$, $SAF_{U\setminus\{p_x\}} <_p SAF_{p_x,U}$ for any given $U{:}\mathcal{P}(P)$ and $p_x{:}P$, $SAF_U < SAF^{undo}_{p_x,U'}$ for any given given $U, U'{:}\mathcal{P}(P)$ and $p_x{:}P$,
- $LE_U <_p LE_{U'}$ iff $U \subset U' \wedge p \in U$ or $p \in U \wedge p \notin U'$ for any given $U, U'{:}\mathcal{P}(P)$

Definition 10. We define the cone labeling $p : D'_N \rightarrow \{\Delta, \nabla\}$ as follows: Given any state $s{:}D'_N$, $p(s) = \Delta$ iff $\exists q{:}P.s(q) \in \{SAF, LE\} \vee (\exists q : P.s(q) = ES \wedge \exists q'{:}P.q' \in \{LOS, Exclusive\})$ otherwise $p(s) = \nabla$.

The specification indicates that if there is one thread in the *ES* node and one thread in the *SAF* node, either one of them should be able to progress to the next node. This is not simultaneously possible in the implementation, as progressing to the *LOE* node requires the *busy* flag to be *true* and the *forbidden* flag to be *false*, while progressing to the *LOS* node requires all *busy* flags to be false and all *forbidden* flags to be *true*. Thus, the subfocus point of each node is chosen such that the external actions are enabled directly given that they would also be enabled in the specification, with the exception of $SAF_\emptyset$ which is used as the focus point of the *SAF* node.

We show that there is a path of *int* actions from this to some state d_{int} in which the transition to *LOE* is enabled. This is outlined in Theorem 2 for which the proof is given in the appendix [18]. The general idea behind the proof is that if the *forbidden* flag is set before it is read by the thread in the *ES* node, the *busy* flag will be set back to false. Repeating this, leads to all *busy* flags being *false* and all *forbidden* flags being *true*, thus enabling the transition to *LOE*.

We now conclude by proving the implementation and specification of the busy-forbidden protocol equivalent in Theorem 3.

Theorem 2. Given some state $d{:}D$, some thread $p_{SAF}{:}P$, and some data configuration $e_\tau{:}E_\tau$ such that $FC(d)$ and $c'_\tau(h(d), e_\tau)$ hold, $h(d)(p_{SAF}) = SAF$ and $g'_\tau(h(d), e_\tau) = LOE$. There must be some state $d_{int}{:}D$ such that $d \xrightarrow{int}{}^* d_{int}$ and $c_\tau(d_{int}, e_\tau)$ hold and $h(g_\tau(d_{int}, e_\tau))(p_{SAF}) = LOE$.

Theorem 3. The LPE of the implementation given in Table 10 and the LPE of the specification given in Table 9 are divergence-preserving branching bisimilar.

Proof. To prove the aforementioned equivalence, we show that all ten requirements given in Theorem 1 hold using Invariants 1, 2, and 3, and the state mapping, focus condition, ordering and cone labeling, given in Definitions 7, 8, 9, and 10, respectively. From the linear process equation, it is relatively easy to see that Requirements I, II, V, VI, $\text{I}_\triangle$, and $\text{II}_\triangle$ are not invalidated. As such, we refer the reader to their extended proofs, found in the appendix [18].

Both the implementation and specification contain exactly three externally observable actions that are not always enabled. For these actions, we show that if the action in the specification is enabled, the same action is also enabled in the corresponding focus point in the implementation, and if the action in the implementation is enabled, the corresponding specification action is also enabled, thus showing that Requirements III, and IV hold.

The first action is the $\text{load}(Forbidden(p), false, p)$ action in ES_2 and the τ transition from the *ES* to the *LOE* node in the specification. The *load* action is only enabled when $forbidden(p)$ is *false*, and the τ transition in the specification is only enabled if there are no threads in *LOS* or *Exclusive* node. As per Invariant 3, these conditions hold exactly when they hold in the corresponding focus points.

The second action is the $\text{lock}(p)$ action in *EE* and the τ transition in the *EE* node in the specification. The *lock* action is only enabled when *mtx* is *false*, and the τ transition in the specification is only enabled if there is no thread in

the *SAF*, *LOS*, *Exclusive*, and *LE* node. As per Invariant 1, these conditions, again, hold exactly when they would hold in the corresponding focus points of the implementation.

The third action is the load($Busy(p_x), false, p$) action in $SAF_{p_x,U}$ and the τ transition from the *SAF* to the *LOS* node in the specification. The *load* action is only enabled when $Busy(p)$ is *false* and the τ transition is only enabled if there is no thread in the *LOE* and *Shared* nodes. As per Invariant 2, if $busy(p)$ is *false* then the *LOE* and *Shared* node are empty and thus, if the action is enabled in the implementation, it is also enabled in the specification. As per the same invariant, the only focus points in which the action would not be enabled while it would be in the corresponding specifications state, are the ones in which a thread is in the *SAF* node, i.e. some thread $p{:}P$ has the substate $SAF_{\emptyset}$. In these cases, as per Theorem 2, there must be some finite path of *int* actions to some state d_{int} in which this action is enabled.

In the corresponding focus points for the *SAF* and *LE* cone, there is always at least one internal action enabled. In the focus point for the *ES* cone, the $load(Forbidden(p), true, p)$ action is enabled iff $forbidden(p)$ is *true*. As per Invariant 3, the only focus points in which $Forbidden(p)$ is *true* are the ones in which the *LOS* or *Exclusive* node are occupied. In all other focus points, there are no further internal actions enabled. Thus Requirement III_{Δ} holds.

If a cone is labelled as non-diverging (∇), then each thread should be in one of the following nodes: *Free*, *LOE*, *Shared*, *LS*, *EE*, *LOS*, *Exclusive*, or *OE*, or *ES*, given that there are no threads present in either *LOS* or *Exclusive*. With the exception of the $load(Forbidden(p), true, p)$ action in the *ES* node, all the internal actions within these nodes take us closer to a focus point. As per Invariant 3, *forbidden* is *true* only if there is a thread present in either the *LOS* or *Exclusive*, *LE*, or *SAF* node, which are known to be empty. Thus Requirement IV_{Δ} also holds and the implementation and specification are divergence-preserving branching bisimilar as per Theorem 1. □

7 Conclusion and Future Work

We have extended the cones and foci proof framework [6,11] with four additional requirements, i.e. Requirements I_{Δ}, II_{Δ}, III_{Δ}, and IV_{Δ}, such that it can be used to prove divergence-preserving branching bisimulation. We have proven this extension to be sound and have used it to prove the implementation and specification of the novel busy-forbidden protocol [9] to be equivalent.

We note some opportunities to extend upon the work in this paper:

- The completeness of the extended cones and foci proof framework has not been formally proven. We assume its completeness due to the weakening of Requirement III, and it is of similar interest as to whether this Requirement can be made stronger without loss of our assumed completeness.
- As mentioned before, the original cones and foci proof framework has been used for the verification of the sliding window protocol [2]. The communication channels used by this protocol are unreliable and thus allow divergence.

As such, the sliding window protocol could provide an interesting case study for our extension of the cones and foci proof framework.
– The diverging loops in the external behavior are considered to be *improbable*, as such, we abstract away any actual, but potentially informative, probabilistic analysis of the protocol.

References

1. IEEE standard for a high performance serial bus. IEEE Std 1394–1995, pp. 1–384 (1996). https://doi.org/10.1109/IEEESTD.1996.81049
2. Badban, B., Fokkink, W., Groote, J.F., Pang, J., Pol, J.V.d.: Verification of a sliding window protocol in μcrl and pvs. FAC **17**(3), 342–388 (2005). https://doi.org/10.1007/s00165-005-0070-0
3. Baeten, J., Weijland, W.: Process algebra, Cambridge tracts in theoretical computer science, vol. 18. Cambridge University Press (1990). https://doi.org/10.1017/CBO9780511624193
4. Bunte, O., et al.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 21–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_2
5. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM **14**(10), 667–668 (1971). https://doi.org/10.1145/362759.362813
6. Fokkink, W., Pang, J., van de Pol, J.: Cones and foci: a mechanical framework for protocol verification. Formal Methods Syst. Des. **29**(1), 1–31 (2006). https://doi.org/10.1007/s10703-006-0004-3dBLP:journals/fmsd/FokkinkPP06
7. åke Fredlund, L., Groote, J.F., Korver, H.: Formal verification of a leader election protocol in process algebra. Theor. Comput. Sci. **177**(2), 459–486 (1997). https://doi.org/10.1016/S0304-3975(96)00256-3
8. van Glabbeek, R., Luttik, B., Trcka, N.: Computation tree logic with deadlock detection. Logical Methods Comput. Sci. **5**(4) (2009). https://doi.org/10.2168/LMCS-5(4:5)2009
9. Groote, J.F., Laveaux, M., van Spaendonck, P.H.M.: A thread-safe term library, pp. 422–459 (2022). https://doi.org/10.1007/978-3-031-19849-6_25
10. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. The MIT Press, Cambridge (2014)
11. Groote, J., Springintveld, J.: Focus points and convergent process operators?: a proof strategy for protocol verification. J. Logic Algebraic Program. **49**, 31–60 (2001). https://doi.org/10.1016/S1567-8326(01)00010-8
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
13. Luttik, B.: Divergence-preserving branching bisimilarity. EPTCS **322**, 3–11 (2020). https://doi.org/10.4204/EPTCS.322.2
14. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
15. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta informatica **6**(4), 319–340 (1976). https://doi.org/10.1007/BF00268134
16. Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Commun. ACM **19**(5), 279–285 (1976)

17. Shankland, C., Van Der Zwaag, M.: The tree identify protocol of IEEE 1394 in μcrl. Formal Aspects Comput. **10**(5), 509–531 (1998)
18. van Spaendonck, P.H.M.: Verification of the busy-forbidden protocol, August 2022. https://doi.org/10.48550/arxiv.2208.05334
19. Usenko, Y.S.: Linearization of μcrl specifications. In: Proceedings of 3rd Workshop on Verification and Computational Logic, Technical Report DSSE-TR-2002-5. Department of Electronics and Computer Science, University of Southampton. Citeseer (2002)