Verification of the busy-forbidden protocol
(using an extension of the cones and foci framework)

P.H.M. van Spaendonck

Department of Mathematics and Computer Science,
Eindhoven University of Technology
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 proof using the cones and foci proof framework, which rephrases whether two specifications are branching bisimilar in terms of proof obligations on relations between the data objects occurring in the implementation and specification. We provide an extension of this framework consisting of three additional properties on data objects, When these three additional properties also hold, the two systems are divergence-preserving branching bisimilar, a stronger version of the aforementioned relation that also distinguishes livelock. We prove this extension to be sound and use it to give a general equivalence proof for the busy-forbidden protocol.

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

1 Introduction

In [10], Groote et al. developed a thread-safe parallel term library using a hashtable to ensure that only a single instance of a term can exist in memory at any given moment. A readers-writer lock is used to ensure mutual exclusivity between accessing and inserting elements, done through the readers lock, and garbage collection or rehashing, done through the writer lock.

As more than 99% of the workload occurs in the readers section, a new readers-writer lock called the busy-forbidden protocol is used, which is optimized for high readers section workloads. The authors give process algebraic specifications of both the implementation of the new algorithm as well as a specification of its external behavior and have proven these equivalent for up to 7 threads using the mCRL2 toolset. However they were unable to do this for more than 7 threads due to the statespace of the implementation becoming too large.

Specifications of the external behavior of any given system make it significantly easier to reason about its correctness and how to use it. These specifications are often designed to reflect the desired safety and liveness properties, e.g. the specification of a mutual exclusion algorithm might make it an explicit requirement that the critical section can only be entered if it is empty, thus guaranteeing mutual exclusion. In Section 3 we show how the specification of the busy-forbidden protocol can be used to argue about its correctness.

The correctness of an implementation, such as that of the busy-forbidden protocol, thus becomes contingent on being equivalent to its external specification. As readers-writer locks often use large amount of concurrent threads, a general equivalence proof for the busy-forbidden protocol is desired.

We provide this equivalence proof for any given number of concurrent threads by using the cones and foci proof framework, originally proposed in [9] by Groote and Springintveld and later generalized by Fokkink et al. in [6]. This proof framework simplifies the often complex and cumbersome branching bisimulation proof by reducing it to a small set of propositions on the data

* Supported by project 00795160 (TTW, MASCOT).
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.

Because the equivalence relation proven by the cones and foci framework, i.e. branching bisimulation, does not distinguish livelock, we 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 livelock [8]. Our extension reduces the proof to three additional propositions on the data objects in the implementation and specification models, that, when shown to also hold, imply the equivalence of the models modulo divergence-preserving branching bisimulation. We give a soundness proof of this extension and apply it to the models of the busy-forbidden protocol.

2 Related Work

Formal verification of software is an extensively studied field. The verification of sequential software is often done using Floyd-Hoare logic [11], in which predicate pairs of pre- and postconditions are shown to hold, using a preset set of axioms. In [16, 17], Owicki and Gries provide an extension of this logic for parallel systems, by providing axioms for parallel composition and synchronization.

In a similar fashion, process algebras, such as ACP (Algebra of Communicating Processes) [3], CSP (Communicating Sequential Processes) [4], and CCS (Calculus of Communicating Systems) [14] provide axiom based reasoning for parallel composed processes. However reasoning about large concurrent systems quickly becomes convoluted and thus being able to provide a smaller “equivalent” system is desirable. The smaller model can then be used to reason about the correctness of the actual implementation, such as is done in the Farsite project [2] and at Amazon Web Services [15].

One possible method for proving two systems equivalent is the cones and foci framework. This framework has 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 [1], a complex leader election protocol in [7], and a part of the IEEE P1394 high-speed bus protocol [12] in [18].

3 The busy-forbidden protocol

We first discuss the busy-forbidden protocol. An overview of its implementation using pseudocode is given in Table 1. The protocol uses 2 binary flags per thread and a singular 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 reader section. All flags are initially set to false. The mutex, called mutex, enforces exclusive access to the writer section.

When a thread wants to enter the readers section, it will set its busy flag and wait as long as its forbidden flag is true. The busy flag is temporarily set to false while waiting to avoid deadlock. While waiting, a mutex.TimedLock() can be used to reduce resource contention. The function can be left out without altering the externally visible behaviour of the protocol [10]. Upon leaving the readers section, the thread will simply set its busy flag back to true.

When a thread wants to enter the writer section, it will first acquire the mutex. It will then set 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 acquiring the writer section, from locking out some reader threads while still waiting for others to leave the reader 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.

When leaving, random forbidden flags can also sometimes be set back to true. This prevents each iteration that occurs while leaving, from becoming externally visible and significantly reduces the amount of states in the external specification.

Because successfully entering and leaving the readers section only requires the use of a thread’s own flags, resource contention only occurs when a thread tries to acquire or release the writer
Verification of the busy-forbidden protocol

Table 1: Pseudocode description of the busy-forbidden protocol.

| Function                      | Pseudocode                                                                 |
|-------------------------------|---------------------------------------------------------------------------|
| `enter_shared(thread p)`      | `p.busy := true;` <br>`while p.forbidden` <br>`p.busy := false;` <br>`if mutex.timedLock()` <br>`mutex.unlock();` <br>`p.busy := true;` |
| `leave_shared(thread p)`      | `p.busy := false;`                                                        |
| `enter_exclusive(thread p)`   | `mutex.lock();` <br>`while exists thread q with` <br>`~q.forbidden` <br>`select thread r` <br>`r.forbidden := true;` <br>`if r.busy or sometimes` <br>`r.forbidden := false;` |
| `leave_exclusive(thread p)`   | `while exists thread q with` <br>`q.forbidden` <br>`select thread r` <br>`usually do` <br>`r.forbidden := false;` <br>`sometimes do` <br>`r.forbidden := true` <br>`mutex.unlock();` |

section. Thus, the slowdown due to data sharing is minimal when the writer section is only rarely used.

The externally visible behaviour of the protocol is given in Figure 1 and, as we will prove later, provides an equivalent overview of how threads interact via the protocol. Individual threads move along the transitions between the nodes, called sections, which represent collections of all threads currently residing in that specific section. Transitions labelled as `enter_shared`, `leave_shared`, `enter_exclusive`, and `leave_exclusive` call refer to the identically named functions being called, and those labelled as `enter_shared`, `leave_shared`, `enter_exclusive`, and `leave_exclusive` return refer to those function calls terminating. All transitions not labelled as such represent some sequence of internal calculations that occurs during these function calls. Transitions labelled with a guard only allow a thread to progress if the given condition is met.

The Free section represents a thread not interacting with the protocol and being outside of any section. Each Thread initially starts out in this section.

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

When a thread tries to acquire the writer-lock, it enters the EnterExclusive (EE) section. Once the thread acquires the mutex variable, it will move to the SetAllForbidden (SAF) section 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 section represents a forbidden flag being set back to false, this transition is labelled 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) section before officially terminating the function call.

When the thread starts releasing the writer-lock, it enters the LeavingExclusive (LE) section. Similar to the SAF section, a thread within the LE section 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 labelled as improbable. Once the last forbidden flag is set to false, this is no longer possible and the thread moves to the
OpenedExclusive (OE) section, after which it will officially terminate the function call and move back to the Free section.

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.

Eventual termination of the leave_shared and leave-exclusive functions is also apparent as none of the involved transitions contain any guards and the livelock transition in LE is marked improbable. From the guards in the model, we can also see that eventual termination of the enter_shared and leave-exclusive functions is guaranteed as long as threads will always eventually leave the section they are in and no new threads start trying to enter any of the sections.

4 Linear process equations

Both the implementation of the pseudocode shown in Table 1 and the external behaviour have been modelled in the mCRL2 language. The mCRL2 language is based on the Algebra of Communicating Processes [2] and Calculus of Communicating Processes [14].

The 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 or returning from a function, or setting or reading a flag. Actions consist 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 ∑ 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 ∑ operators are split into separate smaller ∑ operators and individual + operators.

Since the cones and foci 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:

\[ X(d:D) = \sum_{a:Act \ c_a:E_a} \ \sum_{e_a} c_a(d, e_a) \rightarrow a(f_a(d, e_a)) \cdot X(g_a(d, e_a)), \]
where $D$ is the set of states, $Act$ is the set of action labels including $\tau$, $E_a$ is an indexed set of all data types that need to be considered for label $a$, $c_a(d, e_a)$ specifies when action $a$ with parameters $f_a(d, e_a)$ 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 $d$ we are only interested in the reachable states, we introduce the notion of an invariant in Definition 2. A predicate $I$ on the set of states $D$ is called an invariant iff the following holds: for all $a:Act, d:D$ and $e_a:E_a$,

$$I(d) \land c_a(d, e_a) \Rightarrow I(g_a(d, e_a))$$

**5 Equivalence and the cones and foci 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. The definitions of the equivalence relations given in this paper are based on the definitions used in [13] and have been adapted to work with process equations instead of transition systems. In Definition 3 we provide some syntactic glue to make the transition between labelled transition system and clustered LPE 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 data parameters $e_a$ such that $l = a(f_a(d, e_a))$, $c_a(d, e_a)$, and $g_a(d, e_a) = d'$.
- $d \xrightarrow{*} d'$ 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 $l$ holds:

- $B_1$ if $sRt$ and $s \xrightarrow{l} s'$ for some state $s' \in D$, then either $l = \tau$ and $s\tau R t$, or there are states $t', t'' \in D'$ such that $t \xrightarrow{\tau \ast} t'$ and $s\tau R t'$, and $s'\tau R t''$.
- $B_2$ if $sRt$ and $t \xrightarrow{l} t'$ for some state $t' \in D'$, then either $l = \tau$ and $s\tau R t'$, or there are states $s', s'' \in D$ such that $s \xrightarrow{\tau \ast} s'$ and $s'\tau R t', s''\tau R t$, and $s''\tau R t''$.
- $D_1$ if $sRt$ 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 $s_k \tau R t_k$ for all $k \in \mathbb{N}$, then there is a state $t' \in D'$ such that $t \xrightarrow{\tau} t'$, and $s_k \tau R t'$ for some $k \in \mathbb{N}$.
- $D_2$ if $sRt$ 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 $s'\tau R t_k$ 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 $d_0 R d_0'$.

Proving that a relation is a (divergence-preserving) branching bisimulation can be difficult and lead to convoluted proofs. In [13], it is noted that in communicating systems, equivalent states often have a “cone-like” structure as is shown in Figure 2. In this figure, equivalent states are grouped together in the cone $C$. In the foci point state $fc$, all externally visible actions, 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. $\tau$ 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 these “cones”, the cones and foci proof framework can be used to prove equivalence. To do so, we must provide a state mapping \( h : D \rightarrow D' \) that maps states in the implementation to their equivalent state in the specification, a focus condition \( FC : D \rightarrow \mathbb{B} \) that indicates if a state should be considered a focus point, i.e. all externally observable actions are enabled, and a well founded ordering \( \prec_M \) 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.

Any \( \tau \) 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 \( \text{int} \) (short for \( \text{internal} \) action). This allows us to easily distinguish between \( \tau \) actions that are externally observable, i.e. that are preserved in our specification, and those that are not. Any \( \tau \)-loop in the specification, i.e. a \( \tau \) action for which the begin- and endpoint are the same state, is also renamed to \( \text{int} \). While an \( \text{int} \) action is considered a \( \tau \) action, we exclude them from the set of actions \( \text{Act} \).

Both the original Cones and Foci method as well as the version presented in \[6\] require that if two actions \( a \) and \( b \) are enabled in the specification, they must also be simultaneously enabled in each corresponding focus point. As this is often not the case in divergent systems, we relax this requirement by requiring that an action is either directly enabled or in a state that can be reached through a finite amount of \( \text{int} \) actions.

In Theorem 1 we further extend the cones and foci framework with a labelling \( p \) on cones that labels cones as either divergent \( (\Delta) \) or non-divergent \( (\nabla) \), and three additional requirements on the LPEs with regards to this labelling, such that we can also prove two clustered LPEs to be divergence-preserving branching bisimilar.

**Theorem 1.** Consider a clustered linear process equation of an implementation with initial state \( d_0 \) and some invariant \( I \) that holds in \( d_0 \),

\[
X(d:D) = \sum_{a: \text{Act} \cup \{\text{int}\}} \sum_{e_a : \mathcal{E}_a} c_a(d, e_a) \rightarrow a(f_a(d, e_a)) \cdot X(g_a(d, e_a)),
\]

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

\[
Y(d':D') = \sum_{a: \text{Act} \cup \{\text{int}\}} \sum_{e_a : \mathcal{E}_a} c'_a(d', e_a) \rightarrow a(f'_a(d', e_a)) \cdot Y(g'_a(d', e_a)).
\]

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

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

\[
(\text{FC}(d)) \Rightarrow (\exists e_{\text{int}} : \mathcal{E}_{\text{int}}. c_{\text{int}}(d, e_{\text{int}}) \land g_{\text{int}}(d, e_{\text{int}}) <_M d)
\]

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

\[
\forall e_{\text{int}} : \mathcal{E}_{\text{int}}. c_{\text{int}}(d, e_{\text{int}}) \Rightarrow h(d) = h(g_{\text{int}}(d, e_{\text{int}}))
\]

III. Every visible action in the specification must be enabled after a finite amount of \( \text{int} \) actions for each corresponding focus point: For all \( a : \text{Act} \)

\[
\forall e_a : \mathcal{E}_a. (\text{FC}(d) \land c'_a(h(d), e_a)) \Rightarrow (\exists d_{\text{int}} : D. d \xrightarrow{\text{int}} d_{\text{int}} \land c_a(d_2, e_a))
\]
IV Every visible action in the implementation must be mimicked in the corresponding state in the specification: For all $a:Act$

$$\forall e_a : E_a. e_a(d, e_a) \Rightarrow e'_a(h(d), e_a)$$

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

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

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

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

Ia The cone labelling indicates whether or not a specification state allows an int-loop:

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

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

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

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

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

Proof. Define $R \subseteq D \times D'$ as follows:

$$\{ \langle d, h(d) \rangle \mid d \in D \land I(d) \}$$

We first show that $R$ is a branching bisimulation given that Requirements I, II, III, IV, V and VI hold.

Case $B_1$. Consider the pair $\langle d, h(d) \rangle \in R$ such that $d \xrightarrow{l} d_{to}$ for some state $d_{to}:D$ and some action $a:Act \cup \{ \text{int} \}$ with data parameters $e_a : E_a$ such that $l = a(f_a(d, e_a))$. If $l$ is an int action, then from Requirement II it follows that $h(d) = h(d_{to})$ and thus the states $d_{to}$ and $h(d)$ are related. If $l$ is not an int action, it follows from Requirements IV and V that it is also enabled in $h(d)$ with matching parameters. Subsequently, from Requirement VI it follows that $g'_{a}(h(d), e_a) = h(g_a(d, e_a))$ and thus $d_{to} R g'_{a}(h(d), e_a)$.

Case $B_2$. Consider the pair $\langle d, h(d) \rangle \in R$ such that $h(d) \xrightarrow{l} d'$ for some state $d':D'$ and some action $a:Act \cup \{ \text{int} \}$ with data parameters $e_a : E_a$ such that $l = a(f'_a(h(d), e_a))$. Since all int actions in the specification are $\tau$-loops, we have that $d$ and $d'$ are related if $l = \text{int}$. We show that if $l$ is not an int action, then $l$ must be enabled in some state $d_{int}$ that can be reached with a finite amount of int actions.

We first show that if $d$ does not satisfy the focus condition, then there is some state $d_{fc}:D$ such that $d \xrightarrow{\text{int} \land} d_{fc}$ and $FC(d_{fc})$ holds. Let $D_{int}$ be the set of all states reachable from $d$ through int actions that do not satisfy the focus condition, i.e. $D_{int} = \{ d_r : d_r \in D \mid d \xrightarrow{\text{int} \land} d_r \land \neg FC(d_r) \}$. Because $< M$ is a well-founded ordering on $D$, the set $D_{int}$ contains a minimal element $d_{\bot}$. As per Requirement II some internal action in $d_{\bot}$ must be enabled such that the endpoint of that action is smaller than $d_{\bot}$. Since $d_{\bot}$ is a minimal element of $D_{int}$ the endpoint of this action can not be an element of $D_{int}$ and thus $FC$ must old in that state.

As per Requirement III if the focus condition holds in a given state then $l$ must be enabled in some state $d_{int}$ that can be reached with a finite amount of int actions. All states along the path of int actions to $d_{int}$, including $d_{int}$ itself, are related to $h(d)$ as per Requirement II. As per Requirement VII the endpoint of $l$ in either $d$ or $d_{int}$ must be related to $d'$. Thus the relation $R$ is indeed a branching bisimulation. We now show that the relation $R$ is also divergence-preserving, given that Requirements I, II, III, and VII also hold.
such that $d_8 \in P.H.M.\text{ van Spaendonck}$

some $e$ follows from Requirement I

no other element in the sequence is smaller then well-founded ordering on $D$

considered to be $\tau$

outgoing giving the state of that specific thread. The set of substates is given in Definition 6. Substates such that comes directly after $d_\perp$ 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.

Case $D_2$. 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 $d_0 = d$ and for any $n \in \mathbb{N}$ we have $h(d_n) = h(d)$ and $d_n \xrightarrow{\text{int}} d_{n+1}$. We show that there is some $e_\text{int} \in E_\text{int}$ such that $c_\text{int}(h(d), e_\text{int})$ and $g_\text{int}(h(d), e_\text{int})$. If $h(d)$ is labelled $\Delta$ then this directly follows from Requirement $\Box$

We show that we can derive a contradiction if $h(d)$ is labelled as $\nabla$ instead. Because $\prec_\mathcal{M}$ 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, Requirement $\Box$ gives us that any outgoing $\text{int}$ 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.

We thus conclude that the relation $R$ is a divergence-preserving branching bisimulation. $\blacksquare$

6 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 new framework to prove them divergence-preserving branching bisimilar in Section $\Box$

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 behaviour of the busy-forbidden protocol is given in Table $\Box$ in the appendix. The set $S$ contains the sections shown in Figure $\Box$. Each state in the specification is represented using a mapping $s$ that maps each thread to their current section, with each thread starting in the Free section. 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 Figure $\Box$. The improbable actions are int actions.

The linear process equation of the implementation is given in Table $\Box$. The set of implementation states $D_N$ is given in Definition $\Box$. 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 $\Box$. Substates belonging to the same section have been grouped together. All non-typewriter font actions are considered to be $\tau$ actions and italicized actions are specifically considered to be int actions.

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

$$d = (d_{p_1}, d_{p_2}, \ldots, d_{p_N}, \text{busy, forbidden, mtx}) : D_N,$$

in which:

- $d_{p_1}, d_{p_2}, \ldots, d_{p_N}$ are the substates of threads 1 through $N$.
- busy : $P \rightarrow \mathbb{B}$ is the mapping that keeps track of all the busy flags, in which busy($p$) is the current value of the busy flag of thread $p$.
- forbidden : $P \rightarrow \mathbb{B}$ 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:
\( BF(d = \{d_1, \ldots, d_{p_N}, \text{busy, forbidden, mutex}\} : D_N) = \)
\[ \sum_{p \in P} \left( d_p \approx Free \rightarrow \text{enter\_shared\_call}(p) \cdot BF(d_p = ES_1, etc.) \right) + \sum_{p \in P} \left( d_p \approx Free \rightarrow \text{enter\_exclusive\_call}(p) \cdot BF(d_p = EE, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{ES}_1 \rightarrow \text{store}(\text{Busy}(p), true, p) \cdot BF(d_p = ES_2, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{ES}_2 \wedge \text{forbidden}(p) \rightarrow \text{load}(\text{Forbidden}(p), true, p) \cdot BF(d_p = ES_3, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{ES}_2 \wedge \neg\text{forbidden}(p) \rightarrow \text{load}(\text{Forbidden}(p), false, p) \cdot BF(d_p = LOE, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{ES}_3 \rightarrow \text{stored}(\text{Busy}(p), false, p) \cdot BF(d_p = ES_4, busy[p \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{ES}_4 \rightarrow \text{improbable}(\text{Busy}(p), false, p) \cdot BF(d_p = ES_1, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{LOE} \rightarrow \text{enter\_shared\_return}(p) \cdot BF(d_p = \text{Shared}, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{Shared} \rightarrow \text{leave\_shared\_call}(p) \cdot BF(d_p = LS_1, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{LOE} \rightarrow \text{store}(\text{Busy}(p), false, p) \cdot BF(d_p = \text{LS}_2, busy[p \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{LS}_2 \rightarrow \text{leave\_shared\_return}(p) \cdot BF(d_p = EE, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{EE} \wedge \neg\text{mutex} \rightarrow \text{lock}(p) \cdot BF(d_p = \text{SAF}_p, \text{mutex} = \text{true}, etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{SAF}_U \rightarrow \text{store}(\text{Forbidden}(p_x), true, p) \cdot BF(d_p = \text{SAF}_{p \subseteq U}, \text{forbidden}[p_x \rightarrow true], etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{SAF}_{p \subseteq U} \wedge \text{busy}(p_x) \rightarrow \text{load}(\text{Busy}(p_x), true, p) \cdot BF(d_p = \text{SAF}_{U \cup \{p_x\}}, \text{forbidden}[p_x \rightarrow busy], etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{SAF}_{p \subseteq U} \rightarrow \text{load}(\text{Busy}(p_x), false, p) \cdot BF(d_p = \text{SAF}_{U \cup \{p_x\}}, \text{forbidden}[p_x \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{SAF}_{p \subseteq U} \rightarrow \text{load}(\text{Busy}(p_x), false, p) \cdot BF(d_p = \text{SAF}_{U \cup \{p_x\}}, \text{forbidden}[p_x \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{LOE} \rightarrow \text{internal}(p) \cdot BF(d_p = \text{LOE}_2, \text{etc}) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{LOE}_2 \rightarrow \text{enter\_exclusive\_return}(p) \cdot BF(d_p = \text{Exclusive}, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{Exclusive} \rightarrow \text{leave\_exclusive\_call}(p) \cdot BF(d_p = \text{LE}_p, etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{LE}_U \wedge U \approx \{p_x\} \rightarrow \text{store}(\text{Forbidden}(p_x), false, p) \cdot BF(d_p = \text{OE}_1, \text{forbidden}[p_x \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{LE}_U \wedge U \approx \{p_x\} \rightarrow \text{store}(\text{Forbidden}(p_x), false, p) \cdot BF(d_p = \text{OE}_2, \text{forbidden}[p_x \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P, U \subseteq P} \left( d_p \approx \text{LE}_U \rightarrow \text{store}(\text{Lock}(p), true, p) \cdot BF(d_p = \text{LE}_U, \text{forbidden}[p_x \rightarrow false], etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{OE}_1 \rightarrow \text{unlock}(p) \cdot BF(d_p = \text{OE}_2, \text{mutex} = \text{false}, etc.) \right) \]
\[ + \sum_{p \in P} \left( d_p \approx \text{OE}_2 \rightarrow \text{leave\_exclusive\_return}(p) \cdot BF(d_p = \text{Free}, etc.) \right) \]

Table 2: Linear process equation of the busy-forbidden implementation.
Free = \{Free\},
- ES = \{ES_1, ES_2, ES_3, ES_4\},
- LOE = \{LOE\},
- Shared = \{Shared\},
- LS = \{LS_1, LS_2\},
- EE = \{EE\},
- SAF = \{SAF_U|U \subseteq P\} \cup \{SAF_{p_x,U}|p_x:P, U \subseteq P\} \cup \{SAF_{p_x,U}^{undo}|p_x:P, U \subseteq P\},
- LOS = \{LOS_1, LOS_2\},
- Exclusive = \{Exclusive\},
- LE = \{LE_U|U \subseteq P \land U \neq \emptyset\} and 
- OE = \{OE_1, OE_2\}.

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 busy-forbidden 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 amount 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 “\(\ldots\).” A substate or the mtx variable being changed in the resulting state is denoted with the “\(=\)” operator, where the lefthandside is assigned the value on the righthandside, 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.

The invariants used during the equivalence proof help us exclude some unreachable states from the set of states \(D_N\) and show that 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. 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

\[\exists p:\exists d_p \in M \iff mtx, \land \forall p_x, p_y:P, d_{p_x}, d_{p_y} \in M \implies p_x = p_y,\]

where \(M = SAF \cup LOS \cup Exclusive \cup LE \cup \{OE_1\}\).

**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

\[\forall p:\exists d_p \in B \iff busy(p), \text{ where } B = LOE \cup Shared \cup \{ES_2, ES_3, LS_1\}.\]

**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

\[\forall p:\exists d_p \in F, \iff \exists q:P, d_q \in F,\]

where \(F = LOS \cup Exclusive \cup \{LE_U|U \subseteq P \land p \in U\} \cup \{SAF_U|U \subseteq P \land p \in U\} \cup \{SAF_{p,U}|U \subseteq P\} \cup \{SAF_{p,U}^{undo}|U \subseteq P\}.\)
7 Correctness of the busy-forbidden protocol

The state mapping, focus condition, state ordering and cone labelling 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:
\[
h((d_1, d_2, \ldots, d_N, \text{busy, forbidden, mtx})) = s \text{ where } s(p) = h_P(d_p) \text{ for any } p \in P.
\]
The mapping \( h_P \), referred to as the state-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. \( h_P(ES_4) = ES \) and \( h_P(SAF_{(p_1, p_2, p_3)}) = SAF \).

Definition 8. We define our focus condition \( FC : D_N \rightarrow \mathbb{B} \) as follows:
\[
FC((d_{p_1}, d_{p_2}, \ldots, d_{p_N}, \text{busy, forbidden, mtx})) = \bigwedge_{p \in P} FC_{p_x}(d_{p_x}),
\]
where \( FC_{p_x}(d_{p_x}) \overset{\text{def}}{=} p_x \in \{ \text{Free, } ES_2, \text{LOE, } Shared, \text{LS}_2, \text{EE, } SAF_0, \text{LOS}_2, \text{Exclusive, } LE_{(p_x)}, \text{OE}_2 \} \). We refer to the predicate \( FC_{p_x} \), for any given \( p_x \in P \), as the sub-focus condition.

Definition 9. Given two states \( d = (d_{p_1}, d_{p_2}, \ldots, d_{p_N}, \text{busy, forbidden, mtx}) \) and \( d' = (d'_{p_1}, d'_{p_2}, \ldots, d'_{p_N}, \text{busy', forbidden', mtx'}) \), we define the ordering on these states as follows:
\[
d <_M d' \overset{\text{def}}{=} \bigwedge_{p \in P} d_p <_p d'_p,
\]
where, given some thread \( p \in P \), the ordering \( <_p \) on its substates is defined such that only the following holds:
- \( ES_2 <_p ES_1 <_p ES_4 <_p ES_3 \),
- \( LS_2 <_p LS_1 \),
- \( SAF_U <_p SAF_{U'}, \text{ iff } U \subseteq U' \text{ for any given } U, U' : \mathcal{P}(P) \),
- \( SAF_{U'} <_p SAF_{p_x,U'} \text{ iff } U \subseteq U' \text{ for any given } U, U' : \mathcal{P}(P) \) and \( p_x : \mathcal{P} \),
- \( SAF_{p,x,U} <_p SAF_{p,x,U'} \text{ iff } p_x \in U \text{ for any given } U, U' : \mathcal{P}(P) \) and \( p_x : P \),
- \( SAF_{U} < SAF_{p,x,U'}^{undo} \text{ for any given given } U, U' : \mathcal{P}(P) \) and \( p_x : P \),
- \( LOS_2 <_p LOS_1 \),
- \( LE_{U'} <_p LE_U \text{ iff } U \subseteq U' \wedge p \in U \text{ or } p \in U \wedge p \notin U' \text{ for any given } U, U' : \mathcal{P}(P) \).

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

The specification indicates that if there is one thread in the \( ES \) section and one thread in the \( SAF \) section, either one of them should be able to progress to the next section. This is not simultaneously possible in the implementation, as progressing to the \( LOE \) section requires the \( \text{busy} \) flag to be \( \text{true} \) and the \( \text{forbidden} \) flag to be \( \text{false} \), while progressing to the \( \text{LOS} \) section requires all \( \text{busy} \) flags to be \( \text{false} \) and all \( \text{forbidden} \) flags to be \( \text{true} \). Thus, the subfocus point of each section is chosen such that the external actions are enabled directly given that they would also be enabled in the specification, with the exception of \( SAF_0 \) which is used as the focus point of the \( SAF \) section.

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 4 for which the proof is given in the appendix. The general idea behind the proof is that if the \( \text{forbidden} \) flag is set before it is read by the thread in the \( ES \) section, the \( \text{busy} \) flag will be set back to false. Repeating this, leads to all \( \text{busy} \) flags being \( \text{false} \) and all \( \text{forbidden} \) flags being \( \text{true} \), thus enabling the transition to \( LOE \).

We now conclude by proving the implementation and specification of the busy-forbidden protocol equivalent in Theorem 4.
Theorem 2. Given some state $d:D$, some thread $p_{SAF}:P$, and some data configuration $e_r:E_r$ such that $FC(d)$ and $e_{i_r}(h(d),e_r)$ hold, $h(d)(p_{SAF} = SAF$ and $g_r^*(h(d),e_r) = LOE$. There must be some state $d_{int}:D$ such that $d_{int} \neq d$ and $c_{r}(d_{int}, e_r)$ hold and $h(g_r(d_{int},e_r))(p_{SAF} = LOE$.

Theorem 3. The LPE of the implementation given in Table 2 and the LPE of the specification given in Table 10 are divergence-preserving branching bisimilar.

Proof. We show that all nine requirements given in Theorem 1 hold using Invariants 1, 2, and 3 and the state mapping, focus condition, ordering and cone labelling, given in Definitions 7, 8, 9, and 10 respectively. And thus the implementation and specification are divergence-preserving branching bisimilar.

Requirement 1 holds since for each substate that does not satisfy the sub-focus condition, at least one internal action with a smaller endpoint, is always enabled. We can see that none of the int actions in the implementation, i.e. italicized actions in Table 2, have an endpoint that is in a different cone then their beginpoint. Thus, Requirement 1 also holds.

Both the implementation and specification contain exactly three externally visible actions that are not always enabled. We show that if the action in the implementation is enabled, the action in the specification is also enabled, thus showing that Requirement 4 holds. Simultaneously, we show that if an action in the implementation is enabled, the same action is also enabled in the corresponding focus point in the implementation, thus showing that Requirement 3 holds.

The first action is the load(Forbidden($p$),false,$p$) action in ES to the LOE section in the specification. The load action is only enabled when forbidden($p$) is false, and the $\tau$ transition in the specification is only enabled if there are no threads in LOS or Exclusive section. As per Invariant 3 for forbidden($p$) is true iff there is a thread present in one of these sections, and thus when the transition is not enabled in the specification it is also not enabled in the implementation. As per the same invariant, if the action is enabled in the specification it is also enabled in all corresponding focus points.

The second action is the lock($p$) action in EE and the $\tau$ transition in the EE section in the specification. The lock action is only enabled when mtx is false, and the $\tau$ transition in the specification is only enabled if there is no thread in the SAF, LOS, Exclusive, and LE section. As per Invariant 1 if mtx is false iff there is no thread in any of the states belonging to the aforementioned sections or in the OE1 substate. Thus, if the action is enabled in the implementation it is also enabled in the specification. Because the focus condition does not hold in OE1, we have that if the action is enabled in the specification, it is also enabled in the corresponding focus point in the implementation.

The third action is the load(Exclusive($p$),false,$p$) action in SAF$_pu$ and the $\tau$ transition from the SAF to the LOS section in the specification. The load action is only enabled when Exclusive($p$) is false and the $\tau$ transition is only enabled if there is no thread in the LOE and Shared sections. As per Invariant 2 if exclusive($p$) is false then the LOE and Shared section 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 section, i.e. some thread $p:P$ has the substate SAF$_b$. In these cases, as per Theorem 4 there must be some finite path of int actions to some state $d_{int}$ in which this action is enabled. The only visible actions with parameters are the eight enter/leave shared/exclusive call/return actions. We can thus quickly see that the parameters of these actions match and Requirement 4 holds.

Similarly, it should be clear from the linear process equations that the endpoints of all externally visible transitions are also related and thus Requirement 4 is also met. Requirement 4 holds, as the cone labelling $p$ labels a cone as divergent exactly when one of the three improbable loops would be enabled in the specification.

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 4 the only focus points in which Forbidden($p$) is true are the ones in which the LOS or Exclusive section are occupied. In all other focus points,
there are no further internal actions enabled. Thus Requirement II holds.

If a cone is labelled as non-diverging (\(\nabla\)), then each thread should be in one of the following sections: 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\((\text{Forbidden}(p), \text{true}, p)\) action in the ES section, all the internal actions within these sections take us closer to a focus point. As per Invariant 3, \textit{forbidden} is true only if there is a thread present in either the LOS or Exclusive, LE, or SAF section, which are known to be empty. Thus Requirement II also holds and the implementation and specification are divergence-preserving branching bisimilar as per Theorem 1. \(\square\)
References

1. Badban, B., Fokkink, W., Groote, J.F., Pang, J., and Pol, J.v.d.: Verification of a sliding window protocol in $\mu$CRL and PVS. Formal Aspects of Computing 17(3), 342–388 (2005)
2. Baeten, J., and Weijland, W.: Process algebra. Cambridge University Press (1990)
3. Bergstra, J.A., and Klop, J.W.: Algebra of communicating processes with abstraction. Theoretical computer science 37, 77–121 (1985)
4. Bolesky, W.J., Douceur, J.R., and Howell, J.: The Farsite Project: A Retrospective. SIGOPS Oper. Syst. Rev. 41(2), 17–26 (2007). doi: [10.1145/1243418.1243422](https://doi.org/10.1145/1243418.1243422)
5. Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W.: A Theory of Communicating Sequential Processes. J. ACM 31(3), 560–599 (1984). doi: [10.1145/828.833](https://doi.org/10.1145/828.833)
6. Bolosky, W.J., Douceur, J.R., and Howell, J.: The Farsite Project: A Retrospective. SIGOPS Oper. Syst. Rev. 41(2), 17–26 (2007). doi: [10.1145/1243418.1243422](https://doi.org/10.1145/1243418.1243422)
7. Fredlund, L.-å., Groote, J.F., and Korver, H.: Formal verification of a leader election protocol in process algebra. Theoretical Computer Science 177(2), 459–486 (1997). doi: [https://doi.org/10.1016/S0304-3975(96)00256-3](https://doi.org/10.1016/S0304-3975(96)00256-3)
8. Groote, J., and Springintveld, J.: Focus points and convergent process operators: a proof strategy for protocol verification. Journal of Logic and Algebraic Programming 49, 31–60 (2001). doi: [10.1016/S1567-8326(01)00010-8](https://doi.org/10.1016/S1567-8326(01)00010-8)
9. Groote, J.F., Laveaux, M., and Spaendonck, P.H.M. van: A thread-safe Term Library. CoRR abs/2111.02706 (2021)
10. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Commun. ACM 12(10), 576–580 (1969). doi: [10.1145/363235.363259](https://doi.org/10.1145/363235.363259)
11. IEEE Standard for a High Performance Serial Bus. IEEE Std 1394-1995 (1996). doi: [10.1109/IEEESTD.1996.81049](https://doi.org/10.1109/IEEESTD.1996.81049)
12. Luttik, B.: Divergence-preserving branching bisimilarity. Electronic Proceedings in Theoretical Computer Science, EPTCS 322, 3–11 (2020). doi: [10.4204/EPTCS.322.2](https://doi.org/10.4204/EPTCS.322.2)
13. Milner, R.: A Calculus of Communicating Systems. Springer (1980)
14. Shankland, C., and Van Der Zwaag, M.: The tree identify protocol of IEEE 1394 in $\mu$CRL. Formal Aspects of Computing 10(5), 509–531 (1998)
15. Usenko, Y.S.: Linearization of $\mu$CRL specifications. In: Proc. 3rd Workshop on Verification and Computational Logic. Technical Report DSSE-TR-2002-5. Department of Electronics and Computer Science, University of Southampton (2002)
A Process equations of the original implementation model

Tables 5, 6, 7, and 8 contain the process equations used to model a thread \( p: P \) calling the `enter_shared`, `leave_shared`, `enter_exclusive`, and `leave_exclusive` functions, shown in Table 1 respectively. Table 9 contains the process equation of a thread \( p: P \) which repeatedly attempts to enter and then leave either the shared or exclusive section by calling these functions. Table 4 contains the process equations for modelling the busy and forbidden flags and the mutex. The initial composition of these processes is given in Table 2.

The || operator models the parallel composition of processes and allows for both the interleaving of actions as well as for actions to occur in parallel. The `comm` and `allow` operators are used to enforce that only the combinations of actions, listed under the `comm` operation, can and must occur in parallel. This enforces communication between the Thread processes and the Flags and Mutex processes.

All process equations shown in Appendix A are taken from the original paper [10].

|allow|
|---|
|store, load, lock, unlock, internal, improbable, enter_shared_call, enter_shared_return, leave_shared_call, leave_shared_return, enter_exclusive_call, enter_exclusive_return, leave_exclusive_call, leave_exclusive_return |
|, comm({ |
|store\( _f \) \{store\( _p \) \rightarrow store, load\( _f \) \{load\( _p \) \rightarrow load, lock\( _m \) \{lock\( _p \) \rightarrow lock, unlock\( _m \) \{unlock\( _p \) \rightarrow unlock |
|}) |
|Thread\( (p_1) \) || |
|: |
|Thread\( (p_N) \) || |
|Flags\( (\lambda f:F.\text{false}) \) || |
|Mutex\( (\text{false}) \) |
|
|} |
|Table 3: Parallel composition used to model the busy-forbidden protocol.
Flags(flags : F → □) =
\sum_{f,F,p} (\sum_{b:B} store_f(f,b,p) \cdot Flag(flags[f ↦ b])
+ load_f(f,flags(f),p) \cdot Flag(flags))

Mutex(locked : □) =
\sum_{p:P} (\text{locked} \rightarrow lock_m(p) \cdot Mutex(true)
\circ unlock_m(p) \cdot Mutex(false))

Table 4: Model of the components used during busy-forbidden.

EnterShared(p : P) =
enter_shared_call(p) \cdot
TryBothFlags(p) \cdot
enter_shared_return(p)

TryBothFlags(p : P) =
store_p(Busy(p),false,p) \cdot (load_p(Forbidden(p),true,p) \cdot
store_p(Busy(p),false,p) \cdot improbable \cdot TryBothFlags(p)
+ load_p(Forbidden(p),false,p)
)

Table 5: Model of the \texttt{enter} \_\texttt{shared} function shown in Table 1.

LeaveShared(p : P) =
leave_shared_call(p) \cdot
store_p(Busy(p),false,p) \cdot
leave_shared_return(p)

Table 6: Model of the \texttt{leave} \_\texttt{shared} shown in Table 1.

EnterExclusive(p : P) =
enter_exclusive_call(p)
lock_p(p) \cdot
SetAllForbiddenFlags(p,∅) \cdot
enter_exclusive_return(p)

SetAllForbiddenFlags(p : P, forbidden : Set(P)) =
(\forall p : P. p ∈ forbidden) \rightarrow \text{internal}
\circ \sum_{p',p : P} store_p(Forbidden(p'),true,p) \cdot (load_p(Busy(p'),false,p) \cdot
SetAllForbiddenFlags(p, forbidden ∪ \{p'\})
+ load_p(Busy(p'),true,p) \cdot
store_p(Forbidden(p'),false,p) \cdot improbable \cdot
SetAllForbiddenFlags(p, forbidden \setminus \{p'\})
+ store_p(Forbidden(p'),false,p) \cdot improbable \cdot
SetAllForbiddenFlags(p, forbidden \setminus \{p'\})
)

Table 7: Model of the \texttt{enter} \_\texttt{exclusive} function shown in Table 1.
\[
\text{LeaveExclusive}(p : P) = \\
\text{leave}\_\text{exclusive}\_\text{call}(p) \cdot \\
\text{AllowAllThreads}(p, \emptyset) \cdot \\
\text{unlock}_p(p) \cdot \\
\text{leave}\_\text{exclusive}\_\text{return}(p)
\]

\[
\text{AllowAllThreads}(p : P, \text{ allowed } : \text{Set}(P)) = \\
(\forall_{q, r : q ∈ \text{ allowed}}) \\
→ \text{internal} \\
◊ \sum_{p'} p \cdot ( \\
\text{store}_p(\text{Forbidden}(p'), \text{false}, p) \cdot \\
\text{AllowAllThreads}(p, \text{ allowed} \cup \{p'\}) \\
+ \text{store}_p(\text{Forbidden}(p'), \text{true}, p) \cdot \text{improbable} \\
\text{AllowAllThreads}(p, \text{ allowed} \setminus \{p'\}) \\
)
\]

Table 8: Model of the leave\_exclusive function shown in Table 1.

\[
\text{Thread}(p : P) = \\
\text{EnterShared}(p) \cdot \\
\text{LeaveShared}(p) \cdot \\
\text{Thread}(p) \\
+ \text{EnterExclusive}(p) \cdot \\
\text{LeaveExclusive}(p) \cdot \\
\text{Thread}(p)
\]

Table 9: Model of a thread \( p \) interacting with the protocol.

\[
\text{BF}(s : P \rightarrow S) = \\
\sum_{p, p} p(s(p) \approx \text{Free}) → \text{enter}\_\text{shared}\_\text{call}(p) \cdot \text{BF}(s[p \mapsto \text{ES}]) \\
+ \sum_{p, p} p(s(p) \approx \text{ES} \land \not\exists p' : p.s(p') ∈ \{\text{LOS, Exclusive}\}) → \text{improbable} \cdot \text{BF}(s) \\
+ \sum_{p, p} p(s(p) \approx \text{ES} \land \exists p' : p.s(p') ∈ \{\text{LOS, Exclusive}\}) → \tau \cdot \text{BF}(s[p \mapsto \text{LOE}]) \\
+ \sum_{p, p} p(s(p) \approx \text{LOE}) → \text{enter}\_\text{shared}\_\text{return}(p) \cdot \text{BF}(s[p \mapsto \text{Shared}]) \\
+ \sum_{p, p} p(s(p) \approx \text{Shared}) → \text{leave}\_\text{shared}\_\text{call}(p) \cdot \text{BF}(s[p \mapsto \text{LS}]) \\
+ \sum_{p, p} p(s(p) \approx \text{LS}) → \text{leave}\_\text{shared}\_\text{return}(p) \cdot \text{BF}(s[p \mapsto \text{Free}]) \\
+ \sum_{p, p} p(s(p) \approx \text{Free}) → \text{enter}\_\text{exclusive}\_\text{call}(p) \cdot \text{BF}(s[p \mapsto \text{EE}]) \\
+ \sum_{p, p} p(s(p) \approx \text{EE} \land \not\exists p' : p.s(p') ∈ \{\text{SAF, LOS, Exclusive}\}) → \tau \cdot \text{BF}(s[p \mapsto \text{SAF}]) \\
+ \sum_{p, p} p(s(p) \approx \text{SAF}) → \text{improbable} \cdot \text{BF}(s) \\
+ \sum_{p, p} p(s(p) \approx \text{SAF} \land \not\exists p' : p.s(p') ∈ \{\text{LOE, Shared}\}) → \tau \cdot \text{BF}(s[p \mapsto \text{LOE}]) \\
+ \sum_{p, p} p(s(p) \approx \text{LOE}) → \text{enter}\_\text{exclusive}\_\text{return}(p) \cdot \text{BF}(s[p \mapsto \text{Exclusive}]) \\
+ \sum_{p, p} p(s(p) \approx \text{Exclusive}) → \text{leave}\_\text{exclusive}\_\text{call}(p) \cdot \text{BF}(s[p \mapsto \text{LE}]) \\
+ \sum_{p, p} p(s(p) \approx \text{LE}) → \text{improbable} \cdot \text{BF}(s) \\
+ \sum_{p, p} p(s(p) \approx \text{LE}) → \tau \cdot \text{BF}(s[p \mapsto \text{OE}]) \\
+ \sum_{p, p} p(s(p) \approx \text{OE}) → \text{leave}\_\text{exclusive}\_\text{return}(p) \cdot \text{BF}(s[p \mapsto \text{Free}])
\]

Table 10: Process equation of the external behavior as shown in Figure 1.
B Linear process equations of the busy-forbidden protocol

Table 10 contains the process equation of the externally visible behaviour of the busy-forbidden protocol.

C Proofs of invariants and reachable states

In this section of the appendix, we give the explicit proofs of the invariants shown in Section 5. We also provide the proof of Theorem 4, in which we show that: Given a state \( d : D \) in which a thread is present in the SAF section and such that the focus condition holds and the transition from SAF to LOE is enabled in the specification. There must be some finite path of int actions from this state to a state \( d_{int} \) in which the aforementioned action to the LOE section is enabled.

**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 6

\[ \exists p:P. d_p \in M \iff \text{mtx}, \quad \forall p_x, p_y:P. d_{p_x}, d_{p_y} \in M \Rightarrow p_x = p_y \text{ where} \]

\[ M = \text{SAF} \cup \text{LOS} \cup \text{Exclusive} \cup \text{LE} \cup \{\text{OE}_1\} \]

*Proof.* This invariant holds in the initial state as all substates are initially set to Free and mtx is set to false.

Consider some state \( d:D \) in which the invariant holds and no thread is present in the given set of substates \( M \). A thread can only enter this set through the lock\( (p) \) action in EE. This is also the only action that results in mtx being set to true, thus the invariant will also hold in each subsequent state of \( d \) given that no thread is present in \( M \).

Now consider some state \( d:D \) in which the invariant holds and a (single) thread is present in the given set of substates MTX. A thread can only leave this set through the unlock\( (p) \) action in OE\(_1\). This is also the only action that results in mtx being set to false, thus the invariant will hold in all subsequent states of \( d \).

**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 6

\[ \forall p:P. d_p \in B \iff \text{busy}(p) \text{ where} \quad B = \text{LOE} \cup \text{Shared} \cup \{\text{ES}_2, \text{ES}_3, \text{LS}_1\}. \]

*Proof.* This invariant holds in the initial state as all substates are initially set to Free and busy initially maps each thread \( p \) to false.

Consider some state \( d:D \) in which the invariant holds. A thread \( p \) can only enter the set of substates \( B \) through the store\( (\text{Busy}(p), \text{true}, p) \) action in ES\(_1\). This is the only action that results in busy mapping \( p \) to true. Thus if there is no thread present in the set of substates \( B \), the invariant will also hold after each enabled action. Similarly, a thread \( p \) can only leave the set of substates \( B \) through the store\( (\text{Busy}(p), \text{false}, p) \) action in either ES\(_3\) or LS\(_1\), which are the only actions that result in busy mapping \( p \) to false. Thus if the invariant holds in some state \( d:D \), it also holds in all subsequent states.

**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 6

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

where \( F = \text{LOS} \cup \text{Exclusive} \cup \{\text{LE}_U|U \subset P \land p \in U\} \cup \{\text{SAF}_U|U \subset P \land p \in U\} \cup \{\text{SAF}_{p,U}|U \subset P\} \cup \{\text{SAF}_{p,U}^{\text{end}}|U \subset P\} \).
Proof. The invariant holds in the initial state as all substates are initially set to `Free` and `forbidden` initially maps each thread `p` to `false`.

Consider some state `d:D` in which the invariant holds. The set of substates `F` is a subset of `M`, given Invariant [1] and thus there can be at most 1 thread present in `F`. The only action that results in `forbidden` mapping `p` to `true` is the `store(Forbidden(p), true, q)` action that is enabled in the states `SAF_U` and `LE_U`. This is also the only action that results in some thread `q` entering `F`. Thus if there is no thread present in `F`, the invariant will also hold after each enabled action. Similarly, a thread `q` can only leave `F` through the `store(Forbidden(p), false, q)` action in `SAF_{p,U}`, `SAF_{undo}` and `LE_U`, which are the only actions that result in `forbidden` mapping `p` to `false`. Thus if the invariant holds in `d`, it also holds in all subsequent states.

**Theorem 4.** Given some state `d:D`, some thread `p_{SAF}:P`, and some data configuration `e_\tau:E_\tau` such that `FC(d)` and `e_\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_{\text{int}\rightarrow}d_{\text{int}}` and `c_\tau(d_{\text{int}}, e_\tau)` hold and `h(g_\tau(d_{\text{int}}, e_\tau))(p_{SAF}) = LOE`.

Proof. We define the state `aux_U(d) = (d'_1, \ldots, d'_N, busy', forbidden', mutex')`, for any given subset of thread `U \subset P`, such that for all `p:P` we have:

\[
\begin{align*}
d'_p &= ES_1 \text{ iff } d_p \in ES \land d_p \in U \\
d'_p &= ES_2 \text{ iff } d_p \in ES \land d_p \notin U \\
d'_p &= SAF_U \text{ iff } d_p \in SAF \\
d'_p &= d_p \text{ otherwise}
\end{align*}
\]

The values of `busy'`, `forbidden'`, and `mutex'` can be inferred from `d'_1, \ldots, d'_N` as per the invariants.

We show that `d_{\text{int}\rightarrow}aux_U(d)` holds for any given `U \subset P` using induction over `U`. For the base-case, `U = \emptyset`, we have `d = aux_\emptyset`. Now take any `U \subset P` such that `d =_{\text{int}} aux_U(d)` and any `p:P` such that `p \notin U` and `U \cup \{p\} \subset P`. We show that `aux_U(d) =_{\text{int}} aux_{U \cup \{p\}}(d)`. Take `p_{SAF}:P` such that `d_{p_{SAF}} = SAF_U`. If `d_p \in ES`, then the sequence of actions `store(Forbidden(p), true, p_{SAF}) \cdot load(Forbidden(p), true, p) \cdot store(Busy(p), false, p) \cdot load(Busy(p), false, p_{SAF})` takes us from `aux_U(d)` to `aux_{U \cup \{p\}}(d)`. If `p \notin ES`, then the sequence of actions `store( Forbidden(p), true, p_{SAF}) \cdot load(Busy(p), false, p_{SAF})` takes us from `aux_U(d)` to `aux_{U \cup \{p\}}(d)`. As the relation `\rightarrow` is transitive, it follows that `d_{\text{int}\rightarrow}aux_U(d)` for any given `U \subset P`.

We have thus shown that if we take `U = P \setminus \{p_{SAF}\}` such that `p_{SAF} \in SAF`, we have `d_{\text{int}\rightarrow}aux_U(d)`, from which we can take a single `int` action, i.e. `store(Forbidden(p_{SAF}), true, p_{SAF})`, after which the transition to the `LOE` section will be enabled.