Alma Mater Studiorum · University of Bologna

SCHOOL OF SCIENCE
Master's Degree in Computer Science

Implementation of Algorithms for Bisimulation Equivalence

Supervisor:
Prof.
ROBERTO GORRIERI

Author:
ANDREA BARONI

Session III
Academic Year 2020/2021

Acknowledgements

I would like to express my very great appreciation to Professor Roberto Gorrieri, for helping me in writing this thesis.

I wish to thank my parents for their support and encouragement throughout my years as a student at the University of Bologna.

Contents

1 Introduction **4**

2 Labeled transition systems and bisimulation **4**

3 Bisimulation as a fixed point **5**
3.1 Partially ordered set 5
3.2 Least upper bound 6
3.3 Greatest lower bound 6
3.4 Lattice 6
3.5 Monotonic functions and fixed points 6
3.6 How to compute fixed points 7
3.7 Computing bisimulation equivalence 7

4 Kannellakis and Smolka's algorithm **9**
4.1 Preliminaries 9
4.2 The algorithm 10

5 Valmari's algorithm **14**
5.1 Refinable data structure 14
5.2 Methods implemented for the refinable data structure 15
5.3 The algorithm 17
5.3.1 Update procedure 21
5.3.2 Main Procedure 23

6 Implementation **25**
6.1 Implementation of fixed point approach 25
6.1.1 Detailed steps 25
6.1.2 Time complexity 26
6.2 Implementation of Kannellakis and Smolka 27
6.2.1 Schema of the implementation 31
6.2.2 Detailed steps 31
6.2.3 Note on time complexity 32
6.3 Implementation of Valmari's algorithm 33

7 Results **33**
7.1 First test 34
7.2 Second test 36

8 BPP nets and team bisimilarity **37**
8.1 Definitions 37
8.2 Additive closure 40
8.3 Algorithms for checking the additive closure 41
8.3.1 First algorithm 41
8.3.2 Second algorithm 43
8.4 Team bisimulation on places 44
8.5 Team bisimilarity over markings 46

9 Team bisimilarity over places as a fixed point **46**
9.1 Implementation 49
9.1.1 Detailed steps 49
9.1.2 Time complexity 50

10 K&S's algorithm for team bisimilarity **51**
10.1 Implementation 52
10.2 Time complexity 53

11 Conclusions and future work **54**

1 Introduction

In this thesis, I describe three algorithms, and their implementation using the Scala programming language [OSV16], for computing strong bisimulation equivalence on finite labeled transition systems, LTSs for short. Moreover, I introduce the BPP nets, and a bisimulation-based, behavioral equivalence for BPP nets, called team bisimulation equivalence. I have also implemented two algorithms for computing the team bisimulation equivalence on BPP nets.

This thesis is organized as follow. Section 2 introduces the basic definition about labeled transition systems, and the bisimulation equivalence. Section 3 on the next page describes the first algorithm for computing the bisimulation equivalence on LTSs. In particular, it outlines the theory about the fixed points, and how to express the bisimulation equivalence as a fixed point of a monotone functional F. Section 4 on page 9 describes the second algorithm for computing the bisimulation equivalence on LTSs by means of successive refinements of an initial Partition. Section 5 on page 14 describes a data structure called *Refinable data structure* used for implementing the third algorithm. This is the quickest algorithm known for computing the bisimulation equivalence. Section 6 on page 25 describes the implementation that I have done through the Scala programming language of the three algorithms. In Section 7 on page 33 I do a comparison between the second algorithm and the third algorithm. Section 8 on page 37 describes the BPP nets and an equivalence relation on BPP nets called *team bisimulation equivalence*. Section 9 on page 46 describes how to compute the *team bisimulation equivalence* on BPP nets through the fixed point approach, also I outline the implementation through the Scala Programming Language. Section 10 on page 51 describes how to compute *team bisimulation equivalence* adapting the second algorithm for computing bisimulation equivalence on LTSs. Also I outline the implementation through the Scala Programming Language.

2 Labeled transition systems and bisimulation

The definition of LTS is the following:

Definition 2.1 (Labeled transition system)**.** A labeled transition system (LTS for short) is a triple $TS = (Q, A, \rightarrow)$ where:

- Q is the nonempty, countable set of states;
- A is the countable set of *labels* (or *actions*);
- $\rightarrow \subseteq Q \times A \times Q$ is the *transition relation.*

A finite labeled transition system is a labeled transition system with a finite set of states and finitely many transitions. Transition systems are introduced as a suitable semantic model of reactive systems, see [GV15] for more details. For example a coffee machine can be easily described by the LTS in Figure 1 on the next page, where

$$\begin{aligned} Q &= \{\, q1, q2 \,\}, \\ A &= \{\, \mathit{coin}, \mathit{coffee} \,\}, \\ \rightarrow &= \{\, (q1, \mathit{coin}, q2), (q2, \mathit{coffee}, q1) \,\}. \end{aligned}$$

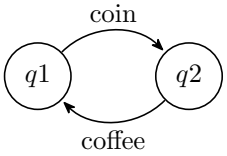


Figure 1: Coffee Machine

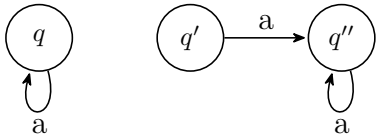


Figure 2: An LTS with three states

Definition 2.2 (Bisimulation)**.** Let $TS = (Q, A, \to)$ be a transition system. A *bisimulation* is a relation $R \subseteq Q \times Q$ such that if $(q_1, q_2) \in R$ then for all $\mu \in A$

- $\forall\, q_1'$ such that $q_1 \xrightarrow{\mu} q_1'$, $\exists q_2'$ such that $q_2 \xrightarrow{\mu} q_2'$ and $(q_1', q_2') \in R$
- $\forall\, q_2'$ such that $q_2 \xrightarrow{\mu} q_2'$, $\exists q_1'$ such that $q_1 \xrightarrow{\mu} q_1'$ and $(q_1', q_2') \in R$

Two states q and q' are *bisimilar* (or *bisimulation equivalent*), denoted $q \sim q'$, if there exists a bisimulation R such that $(q, q') \in R$.

For example the relation $R = \{\, (q, q'), (q, q'') \,\}$ is a bisimulation for the LTS in Figure 2. Indeed if we take the first pair (q, q') we can see that $q \xrightarrow{a} q$ and $q' \xrightarrow{a} q''$ and $(q, q'') \in R$, at the same time we see that $q' \xrightarrow{a} q''$ and q replies with $q \xrightarrow{a} q$ and $(q, q'') \in R$. Moreover for the second pair (q, q'') it is true that $q \xrightarrow{a} q$ and $q'' \xrightarrow{a} q''$ with $(q, q'') \in R$ (the case where q'' moves first is symmetric), so R is a bisimulation.

The goal of this article is to show three algorithms for computing $\sim$ defined as follow:

$$\sim = \bigcup \{\, R \in Q \times Q \mid R \text{ is a bisimulation} \,\} \tag{1}$$

Proposition 2.1. *$\sim$ is an equivalence relation.*

3 Bisimulation as a fixed point

It is possible to see the bisimulation equivalence $\sim$ as the greatest fixed point of a suitable function between relations. Before diving into the algorithm, I will explain some theory behind fixed points. The material in this section is taken from [Ace+07, Chapter 4].

3.1 Partially ordered set

A **partially ordered set** or **poset** is a pair $(\mathrm{D}, \leq)$, where D is a set and $\leq \subseteq \mathrm{D} \times \mathrm{D}$ is a relation:

- reflexive: $d \leq d \; \forall d \in D$
- antisymmetric : $d \leq e \wedge e \leq d \implies d = e \; \forall d, e \in D$
- transitive: $d \leq e \wedge e \leq f \implies d \leq f \; \forall d, e, f \in D$

A poset $(D, \leq)$ is totally ordered if: $(d \leq e) \vee (e \leq d) \, \forall d, e \in D$. As an example of poset we can take $(\mathbb{N}, \leq)$ namely the set of natural number with the usual ordering $\leq$. Also the power set together with inclusion relation $(2^S, \subseteq)$ is an example of poset. Let us now introduce two definitions.

3.2 Least upper bound

Let $(D, \leq)$ a poset and X a subset of D.

- $d \in D$ is an upper bound for X if and only if $x \leq d \; \forall x \in X$.
- d is sup (or Least upper bound) for X, usually written in its contracted form $\bigcup$X, if and only if:
  - d is upper bound for X, and
  - $d \leq d' \; \forall d' \in D$ that is upper bound for X.

3.3 Greatest lower bound

- $d \in D$ is a lower bound for X if and only if $d \leq x \; \forall x \in X$.
- d is inf (or greatest lower bound) for X, usually written in its contracted form $\bigcap$X, if and only if:
  - d is lower bound for X, and
  - $d' \leq d \; \forall d' \in D$ that is lower bound for X.

3.4 Lattice

A poset $(D, \leq)$ is a **lattice** iff $\forall d, e \in D$ there are both the sup $\bigcup \{ d, e \}$ and the inf $\bigcap \{ d, e \}$.

A poset is called a **complete lattice** iff there are both the sup $\bigcup X$ and the inf $\bigcap X$ for every subset X of D.

Remark 3.1. A complete lattice has a minimal element $\bot$ (bottom) given by $\bigcap$ D (greatest lower bound of D) and a maximal element $\top$ (top) given by $\bigcup$ D (least upper bound of D)

3.5 Monotonic functions and fixed points

Given a poset $(D, \leq)$, a function $f \colon D \to D$ is monotonic if and only if:

$$d \leq d' \implies f(d) \leq f(d') \, \forall d, d' \in D$$

An element d is a **fixed point** if and only if $d = f(d)$, a **post-fixed point** if and only if $d \leq f(d)$ and a **pre-fixed point** if and only if $f(d) \leq d$.

Theorem 3.1 (Knaster 1928 - Tarski 1955)**.** *Let $(D, \leq)$ be a complete lattice and $f\colon D \to D$ a monotonic function. The function f has a greatest fixed point Zmax and a least fixed point Zmin defined as follow:*

$$Zmax = \bigcup \{\, x \in D \mid x \leq f(x) \,\}$$
$$Zmin = \bigcap \{\, x \in D \mid f(x) \leq x \,\}$$

3.6 How to compute fixed points

Let $f\colon D \to D$ be a function on a set D. For every $n \in \mathbb{N}$, we define $f^n(d)$ for every $d \in D$ like the following:

$$f^0(d) = d$$
$$f^{n+1}(d) = f(f^n(d))$$

Theorem 3.2. *Let $(D, \leq)$ be a finite complete lattice and $f\colon D \to D$ a monotonic function. Then we get the least fixed point as:*

$$Zmin = f^m(\bot) \text{ for some } m \in \mathbb{N}$$

while the greatest fixed point as:

$$Zmax = f^m(\top) \text{ for some } m \in \mathbb{N}$$

where $\top$ and $\bot$ are both defined in Section 3.4.

3.7 Computing bisimulation equivalence

Now that we have seen some theory about fixed point, it is possible to think to $\sim$ as a greatest fixed point of a suitable function F that transforms binary relations R on states. We will use the algorithm outlined in section 3.6 to compute the relation $\sim$.

Remark 3.2. $2^{Q \times Q}$, that is the set of all binary relations on Q is a complete lattice (finite if Q is finite) with $\top = Q \times Q$

Let us now define F:

Definition 3.1. Given an LTS $TS = (Q, A, \to)$, the functional $F\colon \mathcal{P}(Q \times Q) \to \mathcal{P}(Q \times Q)$ (i.e., a transformer of binary relations over Q) is defined as follow. If $R \subseteq Q \times Q$, then $(q_1, q_2) \in F(R)$ **if and only if** for all $\mu \in A$:

- $\forall\, q_1'$ such that $q_1 \xrightarrow{\mu} q_1'$, $\exists q_2'$ such that $q_2 \xrightarrow{\mu} q_2'$ and $(q_1', q_2') \in R$
- $\forall\, q_2'$ such that $q_2 \xrightarrow{\mu} q_2'$, $\exists q_1'$ such that $q_1 \xrightarrow{\mu} q_1'$ and $(q_1', q_2') \in R$

Proposition 3.1. *For any LTS $TS = (Q, A, \to)$, we have that:*

1. *The functional F is monotone, i.e. if $R_1 \subseteq R_2$ then $F(R_1) \subseteq F(R_2)$.*
2. *Relation $R \subseteq Q \times Q$ is a bisimulation (see Definition 2.2 at page 5) if and only if $R \subseteq F(R)$.*

Theorem 3.3. *$\sim$ is the greatest fixed point of F.*

```
1 X := Q×Q
2 Y := F(X)
3 while X ≠ Y do {
4     X:=Y
5     Y:=F(X)
6 }
```

Listing 1: Pseudocode for computing bisimulation equivalence

Proof. F is monotone so for the Knaster-Tarski theorem (descibed in section 3.5) we have that $Zmax = \bigcup \{ R \mid R \subseteq F(R) \}$, but since R is a bisimulation if and only if $R \subseteq F(R)$ for the point 2 of Proposition 3.1 we have that:

$$Zmax = \bigcup \{ R \mid R \text{ is a bisimulation} \}$$

so we have proved that $Zmax = \sim$ where $\sim$ is defined in equation 1. □

From Theorem 3.3 on the preceding page, and the technique outlined in Section 3.6 on the previous page, we can derive the algorithm in Listing 1 that compute bisimulation equivalence; that is the greatest fixed point of F.

In order to prove the correctness of the algorithm in Listing 1 let us define $\sim$ by means of *stratified bisimulation relations*, see [AIS11] for more details.

Definition 3.2. The *stratified bisimulation relations* $\sim_k \subseteq Q \times Q$ for $k \in \mathbb{N}$ are defined as follow:

- $E \sim_0 F$ for all $E, F \in Q$
- $E \sim_{k+1} F$ iff for each $a \in A$: if $E \xrightarrow{a} E'$ then there is $F' \in Q$ such that $F \xrightarrow{a} F'$ and $E' \sim_k F'$; and if $F \xrightarrow{a} F'$ then there is $E' \in Pr$ such that $E \xrightarrow{a} E'$ and $E' \sim_k F'$.

Given a labelled transition system $\{ Q, A, \rightarrow \}$, let $\mathbf{next}(E, a)$ be:

$$\mathbf{next}(E, a) = \left\{ E' \in Q \mid E \xrightarrow{a} E' \right\}$$

for $E \in Q$ and $a \in A$. Let also $\mathbf{next}(E, *)$ be:

$$\mathbf{next}(E, *) = \bigcup_{a \in A} \mathbf{next}(E, a)$$

An LTS is *image-finite* if and only if the set **next**(E,a) is finite for every $E \in Q$ and $a \in A$. The following lemma is a standard one.

Lemma 3.1. *Assume that $(Q, A, \rightarrow)$ is an image-finite LTS and let $E, F \in Q$. Then $E \sim F$ if and only if $E \sim_k F$ for all $k \in \mathbb{N}$.*

Given a finite state labeled transition system $TS = (Q, A, \rightarrow)$, it is easy to see from the algorithm in Listing 1 that the i-th application of functional $F($ defined in Definition 3.1 on the previous page), $F^i(\dots F^1(F^0(Q \times Q)) \dots)$ in symbol F^i corresponds to $\sim_i$, where $\sim_i$ is defined in Definition 3.2, indeed we have:

$$\begin{aligned} \sim_0 &= F^0(Q \times Q) = Q \times Q \\ \sim_{k+1} &= F^{k+1}(Q \times Q) = F(F^k(Q \times Q)) = F(\sim_k) \end{aligned}$$

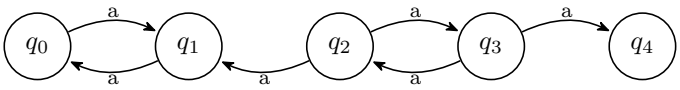


Figure 3: Example for explaining fixed point algorithm

Given a finite LTS $TS = (Q, A, \rightarrow)$, the algorithm in Listing 1 on the previous page, applies the functional F to an initial relation $R = Q \times Q$, until it reaches the greatest fixed point, that is F will eventually reach a $k \in \mathbb{N}$ such that

$$F^k(Q \times Q) = F^{k+1}(Q \times Q) = \sim_{k+1} = \sim_k = \sim$$

For checking the termination's condition of `while` in Listing 1 on the preceding page, it is enough to see that no couples have been removed from the application of the functional F. That is, if $X = F^k(Q)$ it is equal to $Y = F^{k+1}(Q) = F(X)$ for a $k \in \mathbb{N}$ then X is the greatest fixed point of the functional F. In fact the relation that is given in output from the application of the functional F is a non-increasing chain of sets, if you don't remove couples then you must stop.

As an example of application of the algorithm in Listing 1, consider the LTS in Figure 3. We have that:

$$\begin{aligned}
Q &= \{\, q_0, q_1, q_2, q_3, q_4 \,\} \\
F^0(Q \times Q) &= Q \times Q \\
F^1(Q \times Q) &= \{\, (q_0, q_1), (q_0, q_2), (q_0, q_3), (q_1, q_2), (q_1, q_3), (q_2, q_3) \,\} \cup \\
&\quad \cup \{\, (q_1, q_0), (q_2, q_0), (q_3, q_0), (q_2, q_1), (q_3, q_1), (q_3, q_2) \,\} \cup \\
&\quad \cup \{\, (q_0, q_0), (q_1, q_1), (q_2, q_2)(q_3, q_3), (q_4, q_4) \,\} \\
F^2(Q \times Q) &= \{\, (q_0, q_1), (q_0, q_2), (q_1, q_2) \,\} \cup \\
&\quad \cup \{\, (q_1, q_0), (q_2, q_0), (q_2, q_1) \,\} \cup \\
&\quad \cup \{\, (q_0, q_0), (q_1, q_1), (q_2, q_2)(q_3, q_3), (q_4, q_4) \,\} \\
F^3(Q \times Q) &= \{\, (q_0, q_1), (q_1, q_0) \,\} \cup \\
&\quad \cup \{\, (q_0, q_0), (q_1, q_1), (q_2, q_2)(q_3, q_3), (q_4, q_4) \,\}
\end{aligned}$$

Since the time complexity of computing the bisimulation equivalence through the fixed point approach relies heavily on the way the fixed point approach is implemented, the discussion about the complexity is done in the Section 6.1 on page 25. In the aforementioned Section I explain how I have implemented, by means of the programming language Scala, the algorithm for computing the bisimulation equivalence through the fixed point approach.

4 Kannellakis and Smolka's algorithm

The material for introducing the Kannellakis and Smolka's algorithm in this section is taken from [AIS11], from page 8 to page 13. The original article by Kannellakis and Smolka is available at [KS90].

4.1 Preliminaries

Definition 4.1. Let $TS = (Q, A, \rightarrow)$ be a finite labeled transition system. **A partition** is a set of mutually disjoint sets of elements of Q. Let π be a partition,

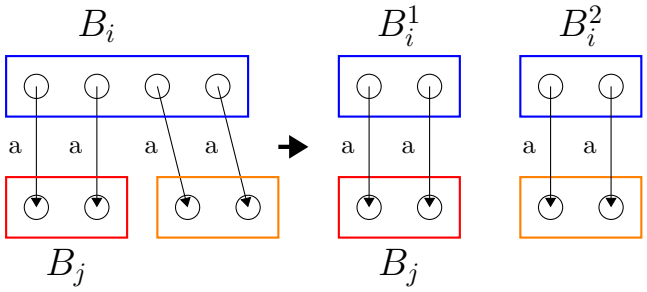


Figure 4: Splitter

in symbols, $\pi = \{ B_1, \ldots, B_n \}$ where each B_i is a set, called block, of elements of Q. We have that

- $B_i \cap B_j = \emptyset$ for each $1 \leq i < j \leq n$
- $\bigcup_{i=1}^{n} B_i = Q$

A block is a set of bisimilar states (or processes).

Suppose $P = \{ 1, 2, 3, 4, 5 \}$, we have that:

- $\{ \{ 1, 3 \}, \{ 4 \}, \{ 2, 5 \} \}$ is a partition of P.
- $\{ \{ 1, 3 \}, \{ 3, 4 \}, \{ 2, 5 \} \}$ is not a partition of P.

Definition 4.2 (Splitter)**.** A block B_j is a **splitter** for a block B_i if there is some states in B_i that afford an a-labeled transition that ends in B_j and other states in B_i that cannot do an a-labeled transition that ends in B_j. Formally a block B_j is a splitter for a block B_i if, given an action $a \in A$, we can divide B_i in two non empty set B_i^1 and B_i^2:

$$B_i^1 = \left\{ s \mid s \in B_i \text{ and } s \xrightarrow{a} s', \quad \text{for some } s' \in B_j \right\} \text{ and}$$
$$B_i^2 = B_i \setminus B_1^i$$

For example in the Figure 4 we have that block B_j, given an action $a \in A$, is a splitter for block B_i.

So we can rewrite

$$\pi = \{ B_1, \ldots, B_i, \ldots, B_n \} \text{ with}$$
$$\pi' = \{ B_1, \ldots, B_i^1, B_i^2, \ldots, B_n \}$$

The partition π' is a refinement of the partition π, indeed for every block B_1 in π' there is a block B_2 in π such that $B_1 \subseteq B_2$.

4.2 The algorithm

The algorithm by Kannellakis and Smolka starts by an initial partition composed of one element that is the set Q and iterates until no further refinement is possible. For example consider the labeled transition system depicted in Figure 5,

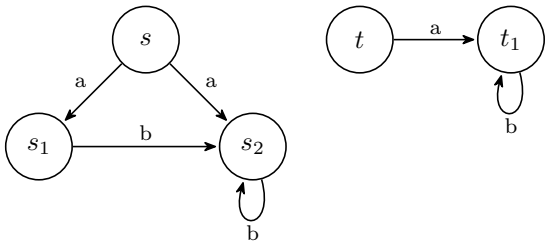


Figure 5: LTS for explaining the algorithm by Kannellakis and Smolka

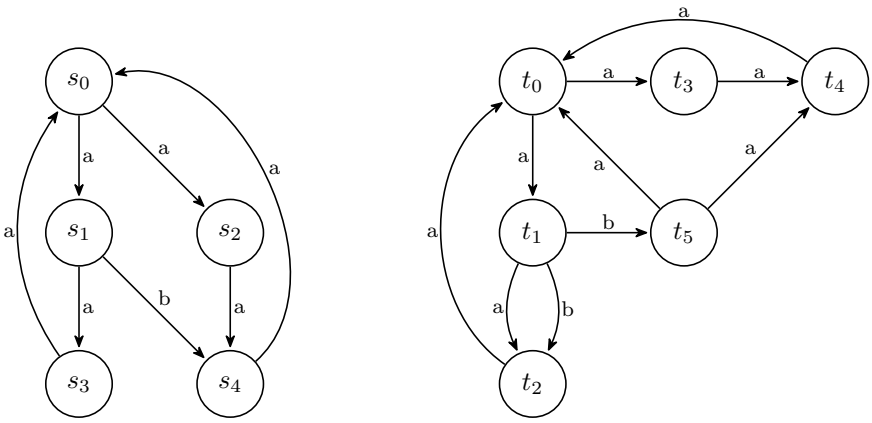


Figure 6: An LTS with 11 states and 16 transitions

the initial partition is $\pi_{init} = \{ Pr \}$ where $Pr = \{ s, s_1, s_2, t, t_1 \}$. The block Pr is a splitter for itself, indeed only some states in Pr afford an a-labeled transition, so we can replace Pr with two sets of states that is, the states that can do an a-labeled transition and the states that cannot. So Pr is replaced by two sets $Pr^1 = \{ s, t \}$ and $Pr^2 = \{ s_1, s_2, t \}$ and π_{init} becomes π'_{init} where $\pi'_{init} = \{ Pr^1, Pr^2 \}$. At this point we see that no other refinement is possible so we have finished. When we say that no other refinement is possible in a partition π it means that for every block $B_1 \in \pi, B_2 \in \pi$, $s_1 \in B_1$, $s_2 \in B_1$, $s'_1 \in B_2$, and $a \in A$ such that $s_1 \xrightarrow{a} s'_1$ there is an $s'_2 \in B_2$ such that $s_2 \xrightarrow{a} s'_2$. As another example for explaining Kannellakis and Smolka's algorithm consider the LTS in Figure 6. The example is taken from [AIS11]. Let the initial partition associated with this LTS be $\{ Q \}$ where

$$Q = \{ s_i, t_j \mid 0 \leq i \leq 4, 0 \leq j \leq 5 \}$$

The block Q is a splitter for itself. Indeed some states in Q afford b-labelled transitions while others do not. If we split Q by Q with respect to action b we obtain a new partition consisting of the blocks

$$\{ s_1, t_1 \} \text{and} \{ s_i, t_j \mid 0 \leq i \leq 4, 0 \leq j \leq 5 \, \text{with} \, i, j \neq 1 \}$$

Note now that the former block is a splitter for the latter one with respect to action a. Indeed only states s_0 and t_0 in that block afford a-labelled transitions that lead to a state in the block $\{ s_1, t_1 \}$. The resulting splitting yields the

partition

$$\{\{s_0, t_0\}, \{s_1, t_1\}, \{s_i, t_j \mid 2 \leq i \leq 4, 2 \leq j \leq 5\}\}$$

The above partition can be refined further. Indeed, some states in the third block have a-labelled transitions leading to states in the first block, but others do not. Therefore the first block is a splitter for the third one with respect to action a. The resulting splitting yields the partition

$$\{\{s_0, t_0\}, \{s_1, t_1\}, \{s_3, s_4, t_2, t_4, t_5\}, \{s_2, t_3\}\}$$

We continue by observing that the block $\{s_3, s_4, t_2, t_4, t_5\}$ is a splitter for itself with respect to action a. For example $t_5 \xrightarrow{a} t_4$ but the only a-labelled transition from s_4 is $s_4 \xrightarrow{a} s_0$ from s_3 is $s_3 \xrightarrow{a} s_0$, from t_2 $t_2 \xrightarrow{a} t_0$, from t_4 is $t_4 \xrightarrow{a} t_0$.The resulting splitting yields the partition

$$\{\{s_0, t_0\}, \{s_1, t_1\}, \{t_5\}, \{s_3, s_4, t_2, t_4\}, \{s_2, t_3\}\}$$

Now we have that t_1 by making action b can reach state t_5 that is in block $\{t_5\}$, whereas s_1 by making action b can reach only s_4 that is in block $\{s_3, s_4, t_2, t_4\}$, so we have that Set $\{t_5\}$, is a splitter for block $\{s_1, t_1\}$; the partition can be further refined in

$$\{\{s_0, t_0\}, \{t_1\}, \{s_1\}, \{t_5\}, \{s_3, s_4, t_2, t_4\}, \{s_2, t_3\}\}$$

Now we have that s_0 by making action a can reach s_1 that is in block $\{s_1\}$, whereas t_0 by making action a cannot reach the block $\{s_1\}$ so we have that the block $\{s_1\}$ is a splitter for the block $\{s_0, t_0\}$, the partition can be further refined in

$$\{\{s_0\}, \{t_0\}, \{t_1\}, \{s_1\}, \{t_5\}, \{s_3, s_4, t_2, t_4\}, \{s_2, t_3\}\}$$

We can continue by observing that s_3 and s_4 by making action a can reach the block $\{s_0\}$ but this is not true for the states t_2 and t_4, so the partition can be further refined in

$$\{\{s_0\}, \{t_0\}, \{t_1\}, \{s_1\}, \{t_5\}, \{s_3, s_4\}, \{t_2, t_4\}, \{s_2, t_3\}\}$$

Now we have that s_2 by making action a can reach the state s_4 that is in block $\{s_3, s_4\}$, whereas t_3 by making action a can reach only t_4 that is in block $\{t_2, t_4\}$. We have that the block $\{s_3, s_4\}$ is a splitter for the block $\{s_2, t_3\}$ and the partition becomes

$$\{\{s_0\}, \{t_0\}, \{t_1\}, \{s_1\}, \{t_5\}, \{s_3, s_4\}, \{t_2, t_4\}, \{s_2\}, \{t_3\}\}$$

Now we have finished because the partition cannot be further refined.

The pseudo-code for Kannellakis and Smolka's algorithm is given in Listing 3 on the following page. The algorithm uses the function $split(B, a, \pi)$ described in Listing 2 on the next page, which given a partition π, a block B in π and an action a, splits B with respect to each block in π and action a. For example if we take the LTS in Figure 6 on the preceding page, the call

$$\texttt{split}(\{s_1, t_1\}, b, \{\{s_0, t_0\}, \{s_1, t_1\}, \{t_5\}, \{s_3, s_4, t_2, t_4\}, \{s_2, t_3\}\})$$

```
function split(B,a,π){
    choose some state s ∈ B
    B1, B2 := ∅

    for each state t ∈ B do
        if s and t can reach the same set of blocks
        in π via a-labelled transitions then
            B1 := B1 ∪ { t }
        else
            B2 := B2 ∪ { t }

    if B2 is empty then
        return { B1 }
    else
        return { B1, B2 }
}
```

Listing 2: Pseudo Code for Split(B,a,π)

```
π := { Q }
changed := true
while changed do
    changed := false
    find := false
    for each block B ∈ π do
        if find then
            break
        for each action a do
            sort the a-labelled transitions from states in B
            if split(B,a,π) = { B1, B2 } ≠ { B } then{
                refine π by replacing B with B1 and B2
                changed := true
                find := true
                break
            }
```

Listing 3: The algorithm by Kannellakis and Smolka in pseudocode

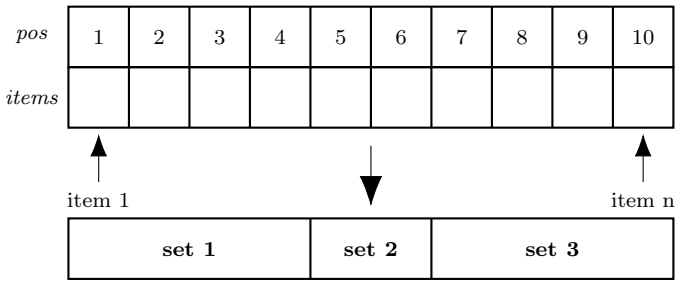


Figure 7: Partition

returns the pair $(\{ s_1 \}, \{ t_1 \})$ because the only block in the partition that can be reached from s_1 via b-labeled transition is $\{ s_3, s_4, t_2, t_4 \}$, whereas t_1 can also reach the block $\{ t_5 \}$. The complexity of the algorithm is $O(nm)$ where n is the number of states and m is the number of transitions, indeed the maximum number of blocks that we have to check is n because a block composed of one element cannot be further split. Deciding if a block B can be split or not costs $O(m)$ because we have to scan all the transitions that have the start state that belongs to B. So the complexity is $O(nm)$.

5 Valmari's algorithm

The Valmari's algorithm, described in [Val09], by the use of a suitable data structure is able to compute the bisimulation equivalence on a finite label transition system in $O(m \log n)$ time where m is the number of transitions and n is the number of states. In the following I will present this kind of data structure called *refinable data structure*. Valmari's algorithm is an adaptation of the algorithm proposed by Paige and Tarjan described in [PT87]. Indeed the algorithm proposed by Paige and Tarjan was defined on labeled transition systems that have the set of labels composed of only one action (see Definition 2.1 on page 4, for the definition of labeled transition system).

5.1 Refinable data structure

The refinable data structure maintains a partition $\{ A_1, \ldots, A_m \}$ of a set of $Items = \{ 1, \ldots, n \}$, where n is the size of the set *Items*, as the one in Figure 7; the items in our case can be the states or the transitions; that is we will use in the algorithm some refinable data structures whose items are states and some whose items are transitions. Both states and transitions are represented as natural numbers. Each set in Figure 7 may be 1-marked, 2-marked or unmarked as in the Figure 8 on the following page. The partition is refinable, meaning that is possible to replace each set A_i with two new disjoint subsets A_{i_1} and A_{i_2} providing that

- $A_{i_1} \cup A_{i_2} = A_i$,
- $A_{i_1}, A_{i_2} \neq \emptyset$,
- $A_{i_1} \cap A_{i_2} = \emptyset$.

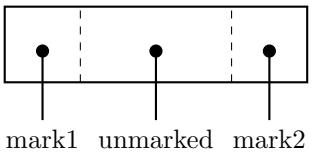


Figure 8: Set

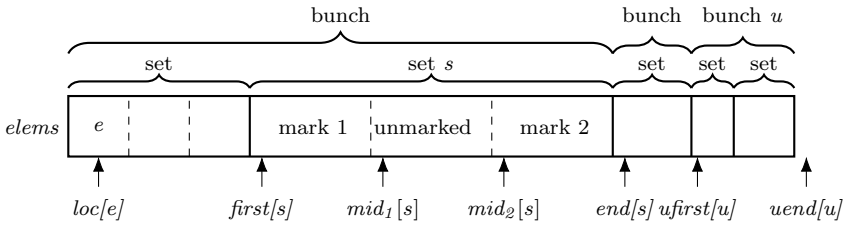


Figure 9: Refinable data structure

For deciding in which subset the items have to go, the refinable data structure uses the mark-1 and mark-2. Given a set A_i, we indicate with A_i^1 the elements $e \in A_i$ that are 1-marked, with A_i^2 the elements $e \in A_i$ that are 2-marked and, with $A_i \setminus (A_i^1 \cup A_i^2)$ the unmarked elements. Also we have that

$$\begin{aligned} A_i^1 \cap A_i^2 &= \emptyset \\ A_i^1 \cap (A_i \setminus (A_i^1 \cup A_i^2)) &= \emptyset \\ A_i^2 \cap (A_i \setminus (A_i^1 \cup A_i^2)) &= \emptyset \end{aligned}$$

Some instances of the data structure uses *bunches* that are a partition P of the set $\{ A_1, \ldots, A_n \}$. A bunch $U_u = \{ A_{u1}, \ldots, A_{ug} \}$ is a subset of the partition P. The refinable data structure, showed in its completeness in Figure 9, comes with some methods that are listed below.

5.2 Methods implemented for the refinable data structure

Here, I will show the methods that are implemented for the refinable data structure, taken from [Val09], from page 127 to page 130.

`Size(s)` Returns the number of elements in the set with index s, that is A_s.

`Set(e)` Returns the index of the set that element e belongs to, that is, the s such that $e \in A_s$.

`Mark1(e), Mark2(e)`
Marks the element e for splitting, at a later time, by means of the methods `Split1(s)` and `Split2(s)` the set A_s that contains e. `Mark1(e)` adds e to A_s^1 and `Mark2(e)` to A_s^2, unless e is already in $A_s^1 \cup A_s^2$. The set A_s^1 is the set of all elements $e_1 \in A_s$ that are 1-marked. The set A_s^2 is the set of all elements $e_2 \in A_s$ that are 2-marked. `Mark1(e)` and `Mark2(e)` do nothing if the element `e` is already in the set A_s^1 or A_s^2, that is if $e \in A_s^1 \cup A_s^2$.

`Split1(s)`

If $A_s^1 = \emptyset$, that is if there is no 1-marked elements inside A_s **returns zero**, if $A_s = A_s^1 \cup A_s^2$, that is if there is no *unmarked* elements inside A_s, then `Split1(s)` unmarks all 1-marked elements in A_s and **returns zero**, otherwise, it updates A_s with $A_s := A_s - A_s^1$, that is remove all 1-marked elements inside A_s creates a new set $A_z := A_s^1$, A_z now contains all the 1-marked elements that was formerly in A_s. Next, puts A_z into the same bunch of A_s, and **returns** $\boldsymbol{z}$. In the end, $A_z^1 = A_z^2 = A_s^1 = \emptyset$, that is A_z does not contain any 1-marked elements ($A_z^1 = \emptyset$) or 2-marked elements ($A_z^2 = \emptyset$), also A_s does not contain any 1-marked elements ($A_s^1 = \emptyset$) because the 1-marked elements that was formerly in A_s are now in the new set A_z. The 2-marked elements inside A_s, that is the set A_s^2, has not changed.

`Split2(s)`

If $A_s^2 = \emptyset$, that is if there is no 2-marked elements inside A_s **returns zero**, if $A_s = A_s^1 \cup A_s^2$, that is if there is no *unmarked* elements inside A_s, then `Split2(s)` unmarks all 2-marked elements in A_s and **returns zero**, otherwise, it updates A_s with $A_s := A_s - A_s^2$, that is remove all 2-marked elements inside A_s creates a new set $A_z := A_s^2$, A_z now contains all the 2-marked elements that was formerly in A_s. Next, puts A_z into the same bunch of A_s, and **returns** $\boldsymbol{z}$. In the end, $A_z^1 = A_z^2 = A_s^2 = \emptyset$, that is A_z does not contain any 1-marked elements ($A_z^1 = \emptyset$) or 2-marked elements ($A_z^2 = \emptyset$), also A_s does not contain any 2-marked elements ($A_s^2 = \emptyset$) because the 2-marked elements that was formerly in A_s are now in the new set A_z. The 1-marked elements inside A_s, that is the set A_s^1, has not changed..

`No_marks(s)`

Returns True if and only if $A_s^1 = A_s^2 = \emptyset$, that is if A_s does not contain 1-marked elements ($A_s^1 = \emptyset$), nor 2-marked elements ($A_s^2 = \emptyset$).

`First(s), Next(e)`

Since the set A_s is represented as a vector, as we can see in Figure 7, page 14.

$$A_s = \{\, elems[i], elems[i+1], \ldots, elems[j] \,\}$$

`First(s)` returns the first element of the set A_s that in our case is *elems[i]*.

Assuming that $e = elems[i]$, `Next(e)` returns $elems[i+1]$. If e i the last element of the set A_s, in our example *elems[j]*, `Next(e)` returns 0.

`Bunch(s)`

Returns the index of the bunch that set s belongs to.

`Bunch_first(u),Bunch_next(e)`

Let $U_u = \{\, A_{u1}, \ldots, A_{ug} \,\}$ be a bunch. With these operations, the elements of $A_{u1} \cup A_{u2} \cup \cdots \cup A_{ug}$ can be scanned, similarly to how `First(s)` and `Next(e)` scans a set in the partition.

`Has_many(u)`

Returns False if and only if bunch U_u consists of precisely one set.

`Extract_set(u)`
: Let $U_u = \{ A_{u_1}, \ldots, A_{u_g} \}$ be a bunch. If $g = 1$ then this operation returns zero without changing anything. Otherwise, it selects some i, introduces a new bunch $\{ A_{u_i} \}$, removes A_{u_i} from U_i, and returns u_i. The chosen i is such that if U_u has a unique biggest set, then it is not A_{u_i}.

`Left_neighbour(e), Right_neighbour(e)`
: If the partition consists of one set, then both of these return zero. Otherwise, at least one of them returns an element that is not currently in the same set as e, but was in the same set until the most recent splitting of the set. The other one may return zero or an element. I will explain why these methods are used in section where i explain the algorithm.

The algorithm uses also the variables `sets` and `bunches` that record the numbers of sets and bunches respectively. The implementation in pseudo code of these methods, taken from [Val09], on pages 129 and 130, is reported in Listing 4 and Listing 5. Moreover the algorithm uses the following arrays taken from [Val09] on pages 129 and 130:

`elems`
: Contains $1, 2, \ldots, items$ in such an order that elements that belong to the same set are one after another. It is also the case that the sets that belong to the same bunch are one after another in *elems*.

`first, end`
: Indicate the segment in elems where the elements of a set are stored, That is, $A_s = \{ elems[f], elems[f+1], \ldots, elems[l-1] \}$, where $f = first[s]$ and $l = end[s]$.

`mid1, mid2`
: Let f and l be as above, and let $m_1 = mid1[s]$ and $m_2 = mid2[s]$. Then $A_s^1 = \{ elems[f], \ldots, elems[m_1 - 1] \}$, the unmarked elements are $elems[m_1], \ldots, elems[m_2 - 1]$.

  Also we have $A_s^2 = \{ elems[m_2], \ldots, elems[l-1] \}$.

`loc`
: Tells the location of each element in *elems*, that is, $elems[loc[e]] = e$.

`sidx`
: The index of the set that e belongs to is $sidx[e]$. That is $e \in A_{sidx[e]}$

`uidx`
: The index of the bunch that A_s belongs to is $uidx[s]$. That is, $A_s \in U_{uidx[s]}$.

`ufirst, uend`
: We have that U_u is $\{ elems[f], elems[f+1], \ldots, elems[l-1] \}$, where $f = ufirst[u]$ and $l = uend[u]$

5.3 The algorithm

In this section I will present the pseudo code as outlined in [Val09] from page 131 to page 137. The algorithm assumes that states and labels are represented as numbers. In symbols the states and labels are represented respectively as:

```
1  Size(s)
2      return end[s] - first[s]
3
4  Set(e)
5      return sidx[e]
6
7  First(s)
8      return elems[first[s]]
9      /*Certainly exists, because the A_i are non-empty */
10
11 Next(e)
12     if (loc[e] + 1 ≥ end[first[s]] then
13         return 0
14     else
15         return elems[loc[e] + 1]
16
17 Mark1(e)
18     s:= sidx[e]; l:= loc[e]; m:=mid1[s]
19     if m ≤ l < mid2[s] then
20         mid1[s] := m + 1
21         elems[l] := elems[m]; loc[elems[l]] := l;
22         elems[m] := e; loc[e] := m
23
24 Mark2(e)
25     s:= sidx[e]; l:= loc[e]; m:=mid2[s] − 1
26     if mid1[s] ≤ l < m then
27         mid2[s] := m
28         elems[l] := elems[m]; loc[elems[l]] := l;
29         elems[m] := e; loc[e] := m
30
31 Split1(s)
32     if mid1[s] = mid2[s] then mid1[s] := first[s]
33     if mid1[s] := first[s]
34     else
35         sets := sets + 1; uidx[sets] := uidx[s]
36         first[sets]:= first[s]; end[sets]:=mid1[s]; first[s]:=mid1[s]
37         mid1[sets] := first[sets]; mid2[sets] := end[sets]
38         for l:= first[sets] to end[sets] - 1 do
39             sidx[elems[l]]:= sets
40         return sets
41
42 Split2(s)
43     if mid1[s] = mid2[s] then mid2[s] := end[s]
44     if mid2[s] := end[s] then return 0
45     else
46         sets := sets + 1; uidx[sets] := uidx[s]
47         first[sets]:=mid2[s]; end[sets]:=end[s]; end[s]:=mid2[s]
48         mid1[sets] := first[sets]; mid2[sets] := end[sets]
49         for l:= first[sets] to end[sets] - 1 do
50             sidx[elems[l]]:= sets
51         return sets
52
53 No_marks(s)
54     if mid1[s] = first[s] ∧ mid2[s] = end[s] then
55         return True
56     else
57         False
```

Listing 4: Main features of the refinable partition data structure

```
1  Bunch(s)
2      return uidx[s]
3
4  Bunch_first(u)
5      return elems[ufirst[u]]
6
7  Bunch_next(e)
8      if loc[e] + 1 ≥ uend[uidx[sidx[e]]] then
9          return 1
10     else
11         return elems[loc[e] + 1]
12
13 Has_many(u)
14     if end[sidx[elems[ufirst[u]]]] ≠ uend[u] then
15         return True
16     else
17         return False
18
19 Extract_set(u)
20     s1 := sidx[elems[ufirst[u]]]; s2 := sidx[elems[uend[u]-1]]
21     if s1 = s2 then
22         return 0
23     else
24         bunches := bunches + 1
25         if Size(s1) ≤ Size(s2 then
26             ufirst[u] := end[s1]
27         else
28             uend[u] := first[s2]; s1 := s2
29         ufirst[bunches] := first[s1]; uend[bunches]:= end[s1]
30         uidx[s1]:= bunches
31         return s1
32
33 Left_neighbour(e)
34     l:= first[sidx[e]];
35     if l > 1 then
36         return elems[l−1]
37     else
38         return 0
39
40 Right_neighbour(e)
41     l:= end[sidx[e]]
42     if l≤items then
43         return elems[l]
44     else
45         return 0
```

Listing 5: Bunch- and neighbour-features of the refinable partition data structure

$S = \{ 1, 2, \ldots, n \}$, $L = \{ 1, 2, \ldots, \alpha \}$. For the transition relation we have that $\Delta \subseteq S \times L \times S$ and $m = |\Delta|$. The algorithm also takes an initial partition $I = \{ S_1, \ldots, S_k \}$. The initial partition can be given in input to the algorithm or not. If the partition is not given in input, we have that $k = 1$ and the initial partition is equal to Q, where Q is the set of all states of the LTS. If the partition is given in input to the algorithm we may choose to divide the set of states Q of the LTS in two disjoint subsets S_1 and S_2 such that:

1. $S_1 \cup S_2 = Q$,
2. $S_1 \cap S_2 = \emptyset$.

The set S_1 is the set of all states that can do at least one transition, while the set S_2 is the set of deadlock states. The input of the algorithm consists of :

n **:** The number of states;

α **:** The number of labels;

Δ **:** The transition relation

$\{ S_1, \ldots, S_k \}$ The initial partition

In addition we have that

$$\Delta_{a,B} = \Delta \cap (S \times \{ a \} \times B)$$
$$\Delta_{s,a,B} = \Delta \cap (\{ s \} \times \{ a \} \times B)$$

Transitions are represented as three array *tail* , *label* , *head*. Each transitions (s, a, s') has an index t in the range $1, \ldots, m$ such that:

$$tail[t] = s, label[t] = a, \text{ and, } head[t] = s'$$

It is also assumed that the indices of the transitions that share the same head state s are available as:

$$\texttt{In\_transitions[s]} = \{ (s_1, a, s_2) \in \Delta \mid s_2 = s \}$$

The algorithm uses the following data structures taken from [Val09] on page 132.

`Blocks` This is a refinable partition data structure on the states $\{ 1, \ldots, n \}$. Its sets are the blocks and when the algorithm ends each blocks represent the states that are bisimilar.

`Splitters`
This is a refinable partition data structure on transition relation $\rightarrow$ that is on $\{ 1, \ldots, m \}$ where m is the number of transitions. Each set in `Splitters` contains the indices of transitions that can do the same action and end in a state that belongs to the same block. Formally

$$\texttt{Splitters} = \{ \Delta_{a,B} \mid a \in L \wedge B \in \texttt{Blocks} \wedge \Delta_{a,B} \neq \emptyset \} \quad \text{where}$$
$$\Delta_{a,B} = \{ (s, l, s') \mid l = a \wedge s' \in B \}$$

The bunch feature of `Splitters` will be used.

```
1  Update(b, b')
2  if Blocks.Size(b) ≤ Blocks.Size(b') then
3      s := Blocks.First(b)
4  else
5      s := Blocks.First(b')
6  while s ≠ 0 do
7      for t ∈ In_transitions[s] do
8          p := Splitters.Set(t); o := Outsets.Set(t)
9          if Splitters.No_marks(p) then
10             Touched_Splitters.Add(p)
11         if Outsets.No_marks(o) then
12             Touched_Outsets.Add(p)
13         Splitters.Mark1(t); Outsets.Mark1(t)
14     s := Blocks.Next(s)
15 while ¬ Touched_Splitters.Empty do
16     p := Touched_Splitters.Remove
17     u := Splitters.Bunch(p); if Has_many(u) then u := 0
18     p' := Splitters.Split1(p);
19     if u≠0 ∧ p'≠ 0 then
20         Unready_Bunches.Add(u)
21 while ¬ Touched_Outsets.Empty do
22     o:= Touched_Outsets.Remove;
23     o' := Outsets.Split1(o)
```

Listing 6: Update

`Outsets` This is a refinable data structure like `Splitters` but finer, that is transitions that are in the same set share the initial state. `Outsets` $= \{ \Delta_{s,a,B} \mid s \in S \wedge a \in L \wedge B \in Blocks \wedge \Delta_{s,a,B} \neq \emptyset \}$

`Unready_Bunches`
This is an initially empty stack. It contains the indices of the bunches of `Splitters` that consists of two or more sets.

`Touched_Blocks`
This is an initially empty stack that contains the sets in the refinable data structure `Block` that have been marked (with mark-1 or mark-2).

`Touched_Splitters, Touched_Outsets`
These are initially empty stacks that contains the transition in Splitters and `Outsets` that have been marked. It is necessary to introduce these stacks because when a set in `Blocks` is split it is necessary to update these two.

5.3.1 Update procedure

Before discussing the main procedure in Listing 7, it is important to understand the update subroutine in Listing 6. Whenever a set has been split in refinable data structure `Blocks`, it is necessary to update the refinable data structure `Splitters` and `Outsets` accordingly.

For example if a set B_i in the refinable data structure `Blocks`, has been split in B_i^1 and B_i^2 we have to update `Splitters`; in particular those transitions of type Δ_{a,B_i}, where $a \in Labels$ must be updated in Δ_{a,B_i^1} and Δ_{a,B_i^2}. The same procedure discussed above must be repeated for `Outsets`. The parameters b and

```
1  Main_part
2  initialize Blocks to { S }
3  initialize Splitters to = { Δ_{a,B} | a ∈ L ∧ B ∈ Blocks ∧ Δ_{a,B} ≠ ∅ }
4  make every set of Splitters a singleton bunch
5  initialize Outsets to = { Δ_{s,a,B} | a ∈ L ∧ B ∈ Blocks ∧ Δ_{s,a,B} ≠ ∅ }
6  for i := 2 to k do
7      for s ∈ S_i do Blocks.Mark1(s)
8      b := Blocks.Split1(1); Update(1, b)
9  for u := 1 to Splitters.bunches do
10     t := Splitters.Bunch_first(u)
11     while t ≠ 0 do
12         s := tail[t]; b := Blocks.Set(s)
13         if Blocks.No_marks(b) then Touched_Blocks.Add(b)
14         Blocks.Mark1(s); t := Splitters.Bunch_next(t)
15     while ¬Touched_Blocks.Empty do
16         b := Touched_Blocks.Remove
17         b'=Blocks.Split1(b); if b' ≠ 0 then Update(b, b')
18 while ¬Unready_Bunches.Empty do
19     u := Unready_Bunches.Remove; p := Splitters.Extract_set(u)
20     if Splitters.Has many(u) then Unready_Bunches.Add(u)
21     t := Splitters.First(p)
22     while t≠0 do
23         if t = Outsets.First( Outsets.Set(t) ) then
24             s := tail[t]; b := Blocks.Set(s)
25             if Blocks.No marks(b) then Touched_Blocks.Add(b)
26             t1 := Outsets.Left_neighbour(t)
27             t2 := Outsets.Right_neighbour(t)
28             if t_1 > 0 ∧ tail[t_1] = s ∧
29             Splitters.Bunch(Splitters.Set(t_1)) = u
30             ∨ t2 > 0 ∧ tail[t_2] = s ∧
31             Splitters.Bunch(Splitters.Set(t_2)) = u
32             then Blocks.Mark1(s) else Blocks.Mark2(s)
33         t := Splitters.Next(t)
34     while ≠ Touched_Blocks.Empty do
35         b := Touched_Blocks.Remove
36         b' := Blocks.Split1(b); if b' ≠ 0 then Update(b, b')
37         b' := Blocks.Split2(b); if b' ≠ 0 then Update(b, b')
```

Listing 7: Main

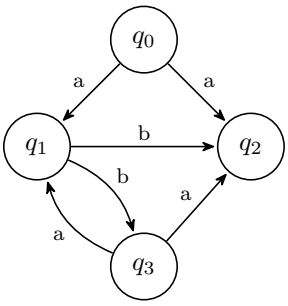


Figure 10: A Lts

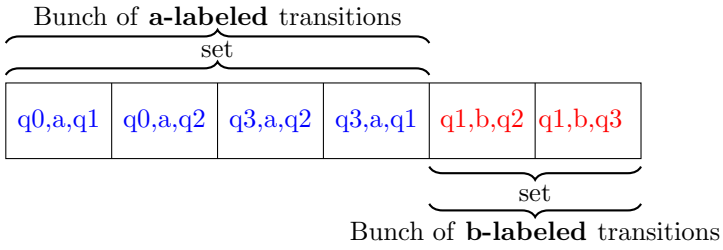


Figure 11: Splitter example

b', given in input to the Update subroutine, represent the indices of the spltted sets. The splitted sets are B_i^1 and B_i^2.

The `Update` procedure starts by scanning the indices of the transitions that end in the smallest set between B_i^1 and B_i^2 by means of `In_Transitions[s]`, that records the transitions that end in s; then Update marks the transitions that belong to the set `In_Transitions[s]`, and proceed with the splitting operation. The process of updating `Splitters` it is a bit more involved because we have to take into account the stack `Unready_Bunches`. For deciding which set has been marked the update procedure uses `Touched_Splitters` and `Touched_Outsets` respectively.

5.3.2 Main Procedure

The main procedure, that is showed in Listing 7 on the previous page, consists of two parts. The first one is between line 9 and line 17 included; its duty is to split the refinable data structure `Blocks` into different set, where each set represents the states that can do an a-labeled transition. Just for a reminder the refinable data structure `Blocks` at the end of the algorithm contains different sets, where each set represents states that are bisimilar.

For detecting the states that can do an a-labeled transition the algorithm scans one at a time the bunches of `Splitters`. Each bunch in `Splitters` is grouped initially by label. That is, if we have only two labels in an LTS like the one in Figure 10, the `Splitters` will be initially like the one in Figure 11. Next for each bunch `Bu` $\in$ `Splitters`, each transition `t` $\in$ `Bu`, will be scanned one at a time, and for each `t` the start state `s` = `tail`[`t`], line 12, Listing 7 on the preceding page, will be marked in the refinable data structure `Blocks`. Next,

after all the transitions `t` $\in$ `Bu`, have been scanned `Blocks` is split, according to the marked states. After each splitting of `Blocks`, `Splitters` and `Outsets` are updated. When all bunches in `Splitters` have been scanned, the second part of the algorithm processes the bunches of `Splitters` that contains more than one set. The refinable data structure `Splitters` contains sets of transitions. For each bunch B that contains two or more sets, the algorithm extracts a Set S (of transitions), makes a new bunch that contains only S, (the bunch B now does not contain S) and for each transition $(s1, a, s2) \in S$ checks if *s1* appears as a tail in other transitions but in the same bunch where before was S.

Just for a reminder a transition t is represented as a triple

$$(\texttt{tail[t]}, \texttt{label[t]}, \texttt{head[t]})$$

meaning that there is a transition t that starting from state `tail[t]` and by doing an action `label[t]`, ends in `head[t]`.

For a summary of the second part: given a bunch B in `Splitters` that contains two or more sets one have to do the following steps:

1. Extract a set S from B and create a new bunch that contain only S and remove S from B.

2. For each transition $t \in S$ check if the tail state of t appears also elsewhere as a tail state of another transition $t' \in B$, B that now does not contain S, if yes mark t with mark1 else mark t with mark2.

3. Split the marked states in Blocks and next update the sets of transitions `Splitters` and `Outsets`.

4. continue until there is no more Bunch in `Splitters` that contains two or more sets.

Thanks to three tricks, the algorithm runs in $O(m \log n)$ times where m is the number of transitions an n is the numbers of states. The tricks are the following:

1. Every time in the second part of the algorithm we extract a set (line 19 in Listing 7 on page 22), from bunches of `Splitters` who contains more than two sets we choose the smallest from the first set and last one set. In this way when a transition is used (for deciding if it must be mark-1 or matk-2) it belongs to a set whose size is at most the half of the previous one. Because all transitions in a set of a `Splitters` have the same label there can be at most n^2 of them (each state has a transition to all other states). Thus each transition can be used at most $\log_2 n^2$.

2. Update defined in Listing 6 on page 21, takes two parameters b and b' that are the indices of the set that has been splitted (B_i^1 and B_i^2) and chooses the smallest from these two. So when a state is used it belongs to a set whose size is at most half the size in the previous time. Thus each state can be used at most $\log_2 n$ times for splitting

3. The third tricks is that the set of labels are never scanned so we can omit to take into account the set of labels.

6 Implementation

I have implemented all the three algorithms outlined above through the Scala Programming language [OSV16]. Full code is available at [Bar21]. The labeled transition system given in input is specified through a textual form. For example the LTS in Figure 10 on page 23, can be given in input to the algorithms in this way:

```
states=q0,q1,q2,q3
relation=q0,a,q1;q0,a,q2;q1,b,q2,q1,b,q3
relation=q3,a,q1;q3,a,q2
```

The lines that start with the word `states` define states, while the lines that start with the word `relation` defines the transition relation.

6.1 Implementation of fixed point approach

In this Section I explain how I have implemented the fixed point approach for computing the bisimulation equivalence described in Section 3 on page 5.

Given a finite LTS $(Q, A, \rightarrow)$ where Q is the finite set of states, A is the finite set of actions and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation composed of m transitions, the implementation computes the greatest fixed point of functional F defined in Definition 3.1 on page 7 through the method explained in Section 3.6 on page 7.

The implementation simply, starts with the relation $R = Q \times Q$, where R is the Cartesian product of the set Q with itself. Then iterates until -under certain conditions- no other couples $(q_1, q_2) \in R$ can be removed.

I have implemented the relation $R = Q \times Q$ -and the subsets of R- as a List of couples $(q_1, q_2) \in R$ where $q_1 \in Q$ and $q_2 \in Q$. The LTS in this algorithm is implemented as a class that has one, fundamental, field: a vector v that takes in input a state $q \in Q$ and an action $a \in A$ and returns a vector of transitions (q, a, q') where $q' \in Q$. The vector of transitions that is returned from the vector v, -vector v that takes in input a state $q \in Q$ and an action $a \in A$- are all the transitions that start from q and by doing an action a end in q'. The vector v takes in input a state q and an action a as numbers so the LTS has two additional fields that are two maps. The first map takes in input a state q and gives in output a number associated to the state q, the second map takes in input an action a and gives in output a number associated to the action a.

6.1.1 Detailed steps

For computing the bisimulation equivalence the algorithm initially sets R to $Q \times Q$. R is the List of couples (q_1, q_2) -with $q_1 \in Q$ and $q_2 \in Q$- of length n^2 where $n = |Q|$.

For each couple $(q_1, q_2) \in R$ the algorithm does the following: for each label $a \in A$:

1. Builds a list `l1` of transitions (q_1, a, q_1') where $q_1' \in Q$. Each tuple (q_1, a, q_1') represents the transition that starting from q_1 and by doing a fixed action a ends in q_1'. The list `l1` is built by means of vector v described above. v takes in input q_1 and a, and returns `l1`.

2. Builds a list `l2` of transitions (q_2, a, q_2') where $q_2' \in Q$. Each tuple (q_2, a, q_2') represents the transition that starting from q_2 and by doing an action a ends in q_2'. The list `l2` is built similarly to the list `l1` described in the point 1 of this list.

3. For each $(q_1, a, q_1') \in$ `l1` the algorithm checks if exist a transition $t = (q_2, a, q_2') \in$ `l2` and $(q_1', q_2') \in R$. If t exists return true else false.

4. Similarly to the previous point, but considering first the state q_2; for each $(q_2, a, q_2') \in$ `l2` the algorithm checks if exist a transition $t = (q_1, a, q_1') \in$ `l1` and $(q_1', q_2') \in R$. If t exists return true else false.

Then if for all $a \in A$ both the checks in point 3 and 4 of the list above returns true, then the couple (q_1, q_2) can stay in the relation R, otherwise it will be removed.

When there is no more couples to be removed from R the algorithm ends and returns the relation R, that contains the bisimulation equivalence $\sim$.

For checking if a couple (q_1, q_2) belongs to R or not in $O(1)$ time, the algorithm uses a vector m. The vector m takes in input a couple (q_1, q_2) and returns true if $(q_1, q_2) \in R$, false otherwise.

When the algorithm finishes to scan the relation R, divides R in two relations: R_1 and R_2. R_1 is the set of couples that **satisfies** the point 3 and 4 of the list above, R_2 is the set of couples that **does not satisfies** the point 3 and 4 of the list above. The couples $(q_1, q_2) \in R_2$ will be marked as false in the vector m described in the previous paragraph.

6.1.2 Time complexity

For defining the time complexity of the algorithm described in this Section (Section 6.1) we have to do the following considerations:

1. Initially the length of the relation $R = Q \times Q$ is equal to n^2 because $n = |Q|$. In the worst case, the algorithm, at every step removes one couple $(q_1, q_2) \in R$ at a time so this loop -that we call `loop1`- costs in the worst case $O(n^2)$.

2. At every step of the `loop1` we have to iterate over all the couples of the relation R, in order to check all the couples $(q_1, q_2) \in R$ for deciding if (q_1, q_2) can stay in R or has to be removed. This loop -that we call `loop2`- also costs in the worst case $O(n^2)$.

3. For deciding if a couple (q_1, q_2) can stay in R we have to build the lists `l1` and `l2` described in Section 6.1.1 for every $a \in A$. Next, for every tuple $(q_1, a, q_1') \in$ `l1` we have to search if exists in `l2` a suitable tuple $(q_2, a, q_2') \in$ `l2` such that $(q_1', q_2') \in R$. That is, if given an action $a \in A$, for every q_1' such that $q_1 \xrightarrow{a} q_1'$ exists a state q_2' such that $q_2 \xrightarrow{a} q_2'$ and $(q_1', q_2') \in R$. If we say that the length of the list `l1` of transitions starting from q_1 is i and i can be at most m where m is the number of transitions, for every couple $(q_1, q_2) \in$ `l1` we have to iterate over all the couples of the list `l2`. This costs $O(ij)$ where j is the length of the list `l2` of transitions starting from q_2.

So if $i \leq m$ and $j \leq m$ where i and j are respectively the length of `l1` and `l2`, deciding if a couple (q_1, q_2) can stay in R costs $O(m^2)$, I have omitted the case when q_2 moves first, that is the case when for every transition that belongs to `l2` we have to search if exists a suitable transition in the list `l1`, but it is symmetric to the case when q_1 moves first, hence costs $O(m^2)$.

Considering that the length of `l1` can be at most m is a pessimistic reasoning because `l1` represents the transitions that -given an action $a \in A$- start from q_1 and by doing an action a end in a state $q' \in Q$. The length of `l1` is certainly less than m. Same for the list `l2`.

In the random LTSs that I have built for doing the tests, and also in other LTSs we may consider that the length of `l1` and `l2` be at most $\frac{m}{n}$ because the transitions that start from a random state q -given an action a- if the transitions are set randomly are seldom m. If the transitions that start from a state q are m it means that all the transitions in the LTS start from q. So if $i \leq \frac{m}{n}$ and $j \leq \frac{m}{n}$, where i and j are the length of `l1` and `l2` respectively we have that deciding if a couple (q_1, q_2) can stay in R costs $O((\frac{m}{n})^2)$.

For what we have said above the time complexity of the implementation that I have done for computing the bisimulation equivalence through the fixed point approach is:

1. $O(n^2 n^2 2(\frac{m}{n})^2) = O(n^2 m^2)$ if we consider, the length of `l1` an `l2` be at most $\frac{m}{n}$.
2. Otherwise if we consider the length of `l1` an `l2` be at most $O(m)$ the time complexity is $O(n^2 n^2 m^2)$.

The Scala code is available in Listing 8 on the next page.

6.2 Implementation of Kannellakis and Smolka

For the purpose of this Section we can think of a partition as a list of blocks, and of a block as a list of states.

I have implemented -for this implementation- the LTS by means of a vector v, that takes in input a state q and an action a and gives in output a vector of transitions of the form (q, a, q') with $q' \in Q$. The vector v formally:

$$v : Q \times A \to \text{Vector}[(Q, A, Q)]$$

takes in input a state q and an action a and gives in output a vector of transitions (q, a, q'). The transitions (q, a, q') start from q and by doing an action a end in q'. Each block is implemented as a list of states, and a partition as a list of blocks. The algorithm also uses a vector `blockOfNode` of length n, where n is the number of states. The vector `blockOfNode` takes in input a state q and returns the index of the block B -that contains q- in the partition. That is given a partition $\pi = \{ B_1, \ldots, B_i, \ldots, B_n \}$ and a state $q \in B_i$, `blockOfNode`(q) returns i. The program also uses a variable `numBlocks` that records the number of Blocks in the partition.

```
class FixedPointFinal(val l: Lts) {

  def fixedPointBisim(): List[(Node, Node)] = {

    val start = l.nodes.flatMap(x => l.nodes.map(y => (x, y)))
    val matrix= (0 until l.numNodes)
      .map(_ => true)
      .toVector
      .map(_ => (0 until l.numNodes)
        .map(_ => true).toVector)

    def check(p: Node, q: Node,
              matrix: Vector[Vector[Boolean]])
    : Boolean = {
      def checkAction(action:Int) ={
        val indexP = l.indexNodes getOrElse(p.name,-1)
        val indexQ = l.indexNodes getOrElse(q.name,-1)
        val v1 = l.vectorTrans(indexP)(action)
        val v2 = l.vectorTrans(indexQ)(action)
        v1.forall(x => v2.exists(y => matrix(x._3)(y._3)) )
      }
      l.numIndex.forall(x => checkAction(x))
    }
    @tailrec
    def updateMatrix(matrix: Vector[Vector[Boolean]],
    toMark:List[(Node,Node)]):Vector[Vector[Boolean]] = {

      toMark match {
        case ::(head, next) =>
          val i = l.indexNodes getOrElse(head._1.name, -1)
          val j = l.indexNodes getOrElse(head._2.name, -1)
          val newLines = matrix(i).updated(j,false)
          updateMatrix(matrix.updated(i,newLines),next)
        case Nil => matrix
      }
    }
    @tailrec
    def iterate(rel: List[(Node, Node)],
                matr:Vector[Vector[Boolean]]):List[(Node, Node)]
    = {
      val (guess1,toBeMarked) = rel.partition(x => {
        check(x._1, x._2, matr) &&
          check(x._2, x._1, matr)
      })
      if (toBeMarked.isEmpty)
        rel
      else {
        val newMatrix= updateMatrix(matr,toBeMarked)
        iterate(guess1,newMatrix)
      }

    }
    iterate(start,matrix)
  }
}
```

Listing 8: Scala fixed point

```
class KS5(val l: Lts) {
  @tailrec
  final def iterateBlock(scanned: List[List[Node]],
                         toScan: List[List[Node]],
                         blockOfNode: Vector[Int],
                         numBlocks: Int
                        )
    : (List[List[Node]], Vector[Int], Int, Boolean) = {

    toScan match {

      case ::(head, next) =>

        val ris = split(head, blockOfNode, numBlocks)

        val (b1, b2, new_blockOfNode, new_numBlocks) = ris

        if (b2.isEmpty) {

          iterateBlock(head :: scanned,
          next, blockOfNode, numBlocks)

        }

        else {
          val two_blocks = b1 :: b2 :: next

          (two_blocks ::: scanned,
          new_blockOfNode, new_numBlocks, true)

        }
      case Nil => (scanned, blockOfNode, numBlocks, false)
    }
  }
  @tailrec
  final def iter(part: List[List[Node]],
                 blockOfNode: Vector[Int],
                 numBlocks: Int
                ): List[List[Node]] = {

    val ris_itB = iterateBlock(Nil, part, blockOfNode, numBlocks)

    val (new_part, new_blockOfNode, new_numBlocks, flag) = ris_itB

    if (flag) {
      iter(new_part, new_blockOfNode, new_numBlocks)
    }
    else
      part
  }
}
```

Listing 9: Partial code for Kannelakis and Smolka's algorithm

```
class KS5(val l: Lts) {
  val tr: Vector[Vector[Vector[(Int, Int, Int)]]] = l.vectorTrans
  def split(ln: List[Node],
            blockOfNode: Vector[Int],
            numBlocks: Int
           ): (List[Node], List[Node], Vector[Int], Int) = {
    def check(vec1:Set[Int], t: Node, action: Int): Boolean = {
      val numt = l.indexNodes.getOrElse(t.name, -1)
      val vec2 = tr(numt)(action).map(x => blockOfNode(x._3))
      vec1 == vec2.toSet
    }
    @tailrec
    def iterAction(actions: List[Int])
    : (List[Node], List[Node], Vector[Int], Int) = {
      actions match {
        case ::(cons_a, tail_a) =>
          val s = ln.head
          val nums = l.indexNodes.getOrElse(s.name, -1)
          val vec1 = tr(nums)(cons_a)
            .map(x => blockOfNode(x._3))
            .toSet
          val (b1, b2) = ln.partition(t => check(vec1, t, cons_a))
          if (b2.isEmpty) {
            iterAction(tail_a)
          }
          else {
            @tailrec
            def UpdateVector(list: List[Node],
                             vec: Vector[Int],
                             index: Int)
            : Vector[Int] = {
              list match {
                case ::(head, next) =>
                UpdateVector(next,
                  vec.updated(
                    l.indexNodes.getOrElse(head.name, -1),
                    index
                  ),
                  index)
                case Nil => vec
              }
            }

            val new_blockOfNode = UpdateVector(b2,
              blockOfNode, numBlocks + 1)
            (b1, b2, new_blockOfNode, numBlocks + 1)
          }
        case Nil => (ln, Nil, blockOfNode, numBlocks)
      }
    }
    iterAction(l.numIndex)
  }
}
```

Listing 10: Split code from Kannelakis and Smolka

6.2.1 Schema of the implementation

The algorithm starts with a partition P that contains a unique block B. B contains all states of the LTS. Next the algorithm scans one at a time each block B in the partition P and does the following: for each $a \in A$:

1. Create two empty set B_1 and B_2
2. Select a state $s \in B$
3. For each state $t \in B$, if s and t can reach the same set of blocks in the partition P via a-labelled transitions then add t to the set B_1 otherwise add t to the set B_2.
4. If B_2 is not empty replace B with B_1 and B_2 in the partition P. If B_2 is empty go ahead with the next label -indexed with a-. If there is no more label to check go ahead with the next block and repeat the procedure in this list starting over to scan all the labels. If there is no more block to check stop and give in output the partition P that now contains the bisimulation equivalence $\sim$.

6.2.2 Detailed steps

For implementing the schema outlined above, I have created the programs showed in Listing 10 on the preceding page and in Listing 9 on page 29. The program in Listing 9 has the burden to scan all the blocks B in the partition P in order to check if the block B can be split in two non-empty set:

1. B_1: the set of states that **can reach** the same set of blocks via a-labeled transitions.
2. B_2: the set of states that **cannot reach** the same set of block via a-labeled transitions.

If there is no more blocks that can be split the algorithm stops and returns the bisimulation equivalence contained in P.

For checking if a block B -given a partition P- can be split in two blocks B_1 and B_2, I have created the program in Listing 10 on the preceding page. In particular the function split takes in input:

1. A block B.
2. A vector `blockOfNode` that given in input a state p returns the index i such that $p \in B_i$.
3. A variable `numBlocks` that keeps track of the number of blocks in the partition.

The function split for each action a does the following steps with the block B that takes in input:

1. Select a state s that belongs to the Block B
2. Create a vector *vec1*. The vector *vec1* contains the transitions (s, a, s') with $s' \in Q$ that start from s and by doing an action a end in s'. The vector *vec1* is created by means of the vector v described in the initial part of the Section 6.2.

3. Transform the vector *vec1* that contains the transitions (s, a, s') with $s' \in Q$ in a vector of indexes. The indexes are all the i such that $s' \in B_i$.

4. For each state $t \in B$ create a vector *vec2* following the same procedure for building the vector *vec1*.

5. For each state $t \in B$, compare the vector *vec1* with the vector *vec2*. If *vec1* is equal to *vec2*, that is s and t can reach the same blocks by doing an a-action add t to B_1, otherwise add t to B_2. If B_2 is not empty replace the block B in the partition P with B_1 and B_2.

6.2.3 Note on time complexity

In the LTS that I have used for this implementation, the list of transitions are specified through a vector v that takes in input a state $q \in Q$ and an action $a \in A$ and gives in output a vector of transitions (q, a, q') with $q' \in Q$. The meaning of (q, a, q') is that there is a transition that starts from a state q and by doing an action a ends in a state q'.

The implementation in this Section does not maintain the vector v sorted. That is for all states $q \in Q$ and actions $a \in A$ the vector $vec_{q,a}$ of transitions (q, a, q') that v gives in output when (v) takes in input q and a is not sorted by the indexes of the blocks the states q' belong. In other words v remains the same during the execution of the algorithm.

For this reason, for checking if two states (q_1, q_2) -given an action a- can reach the same blocks (when I say blocks, I mean indexes of blocks), we have to do the following:

1. Build the vector $vec1$ composed of tuples (q_1, a, q_1') of three items, with $q_1' \in Q$ and a and q_1 fixed. The meaning of (q_1, a, q_1') is that there is a transition that starts from a state q_1 and by doing an action a ends in a state q_1'. The vector $vec1$ is created by means of the vector v that takes in input q_1 and a and returns $vec1$.

2. Transform the vector $vec1$ in a vector of indexes by means of the vector `blockOfNode`, that takes in input a state s and returns the index i such that $s \in B_i$. The vector `blockOfNode` is applied one at a time to all the q_1' such that $q_1 \xrightarrow{a} q_1'$.

3. Build the vector $vec2$ in the same way the vector $vec1$ is built by giving q_2 and a to v.

For checking if $vec1$ it is equal to $vec2$ we have to sort $vec1$ and $vec2$ and then check if $vec1$ is equal to $vec2$. All the operation cost $O(j \log j)$ where j is the maximum length between $vec1$ and $vec2$. Indeed once $vec1$ and $vec2$ are sorted it is easy in linear time to check if $vec1$ is equal to $vec2$.

For this reason, deciding if a block B can be split in two block B_1 and B_2 does not cost $O(m)$ and for this reason the time complexity of the algorithm of Kannellakis and Smolka is not $O(mn)$.

Let α, be the maximum number of the vectors *veci* that we have to sort, and β the maximum length among the vectors *veci*. The time complexity of the algorithm of Kannellakis and Smolka is $O(n \cdot (m + \alpha \cdot \beta \log \beta))$.

I have chosen to implement the Kannelakis and Smolka algorithm without keeping the vector of transition sorted -despite the time complexity seems to be higher than $O(nm)$- because I have seen -in many tests- that if I keep the vector of transition sorted, the execution time slow down a lot. Indeed I have implemented a second version of the Kannellakis and Smolka's algorithm where I keep the vector of transitions sorted after a splitting of a Block happened. The cost of keeping the vector v of transitions sorted slow down a lot the execution time. Also I noticed that the execution of sorting the indexes of blocks reachable from a state q and an action a is irrelevant compared to the cost of keeping the vector v of transitions sorted.

6.3 Implementation of Valmari's algorithm

The implementation of Valmari's algorithm consists more or less of 1100 lines of code so it is impossible to present the full code here, that however can be found in [Bar21]. The algorithm assumes that the states and the labels of the labeled transition system given in input is specified through numbers; for this reason is necessary to process the LTS that is given in textual form for converting it in numeric form, this necessarily introduces an overhead. All the code that I have implemented for the Valmari's algorithm is based on the detailed pseudo code that is described in the article of Antti Valmari [Val09]. The Scala programming language is used mainly as a functional language, as a result the variables cannot be modified once they have been initialized, hence the loops like while are forbidden and must be substituted by means of recursive functions in particular through **tail recursive function** that a compiled time are as efficient as the while loops.

Given an LTS $(Q, A, \rightarrow)$, the time complexity of the implementation of Valmari's algorithm is $O(m \log n)$, where m is the number of transitions and n is the number of states. In the calculation of the time complexity, I have not included the time needed for translating the LTS from the textual form to the numeric form. Also, I have not included the time needed for the initialization of the refinable partition data structures `Splitters` and `Outsets` described in Section 5.3 on page 17. The choice of not including the time needed for the translation and the initialization comes from the fact that, these actions are not closely related to the algorithm.

7 Results

In this Section I show the results of the tests that I have done in order to see how the algorithms of Kannellakis&Smolka and Valmari perform in real cases.

As I stated in the previous Sections I have implemented the algorithms of Kannellakis&Smolka and Valmari through the Scala programming language.

In order to do the tests I have created a certain number of random LTSs. For building the LTSs I have designed a program -in Scala- that takes in input 4 parameters:

1. The number of states, that is the size of Q where Q is the set of states.
2. The size of the set A, where A is the set of labels or actions.

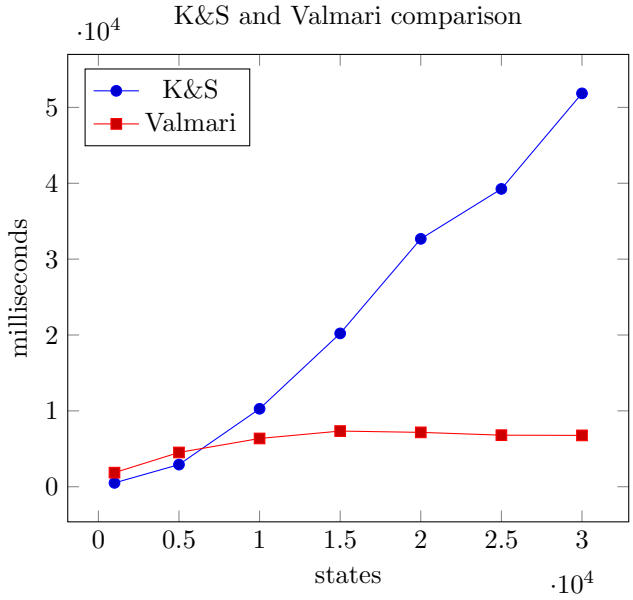


Figure 12: LTSs with 100 000 transitions and labels taken randomly from a set of length 10. States vary on x-axis

3. The number of transitions that we want in our LTSs. A transition is represented as (q, a, q'), where q and q' are taken randomly from the set of states Q, and a is taken randomly from the set of labels or actions A.

4. The number of LTSs that we want to build.

After we have provided the four parameter, the program returns n LTSs where n is specified by the fourth item of the previous list. The LTSs given in output have the characteristics specified by the the first three items.

The program used for creating the LTSs also takes care of removing duplicate transitions, that is let $T = \{ (q, a, q') \mid q, q' \in Q, a \in A \}$ the set of transitions, for each $(q_1, a, q_2), (q_1', a', q_2') \in T$, it is always the case that: $q_1 \neq q_1'$ or $a \neq a'$ or $q_2 \neq q_2'$. For this reason the effective number of transitions may be slightly less than the desired number.

I have carried out two tests where I have measured the time execution in millisecond of the algorithms of Kannellakis&Smolka and Valmari.

Both the tests have been conducted on a laptop with the following characteristics:

OS Windows 10 Home

CPU AMD A9-9420 RADEON R5, 5 COMPUTE CORES 2C+3G 3.00 GHz

RAM 8,00 GB

7.1 First test

The goal of the first test is to measure the time execution of the algorithm of Kannellakis&Smolka and Valmari when we keep fixed the number of transitions

States	Time KS (ms)	Time Valmari (ms)
1000	509.15	1860.4
5000	2918.8	4518.35
10 000	10 274.6	6363.45
15 000	20 210.75	7339.2
20 000	32 674.1	7167.2
25 000	39 255.45	6797.45
30 000	51 847.45	6768.2

Table 1: Results of the test done on LTSs with 100 000 transitions and label taken randomly from a set of length 10. States vary on the first column.

Class Number	Number of states	Number of samples
1	1000	20
2	5000	20
3	10 000	20
4	15 000	20
5	20 000	20
6	25 000	20
7	30 000	20

Table 2: Description of the classes for the first test

and we change the number of states. For doing the test I have built seven classes of LTSs. Each class has a number of transitions fixed to 100 000 and states that vary. For each class the number of states and the number of samples for each class, are specified by means of the Table 2. For each class I have built 20 LTSs, and I have taken the arithmetic mean of time execution in milliseconds of the 20 samples. The results are shown in Figure 12 on the previous page and available in tabular format in Table 1.

As stated by Valmari [Val09] it is difficult to get full control of the activities that is going on in a modern computer. As a consequence, the measurements contain some noise, hence the results should be considered as typical, not as the absolute truth. For example as we can see in Table 1 the time execution of the Valmari's algorithm contains an anomaly: the time execution does not grow when states are grater than 20 000. This may be caused by this fact: when the number of states grow the Valmari's alghorithm removes the unreachable states early in the algorithm and this affect the time execution. Also the time execution depends on the size of the result, that is on the number of classes found. The smaller it is, the less splitting of blocks hence minor time execution.

Despite of what we have said above we can see in Figure 12 on the preceding page that when we keep fixed the number of transitions and let vary the number of states, the time execution of the Kannellakis&Smolka's algorithm grows linearly in the number of states while the Valmari's algorithm grows slower (more or less in a logarithmic scale) .

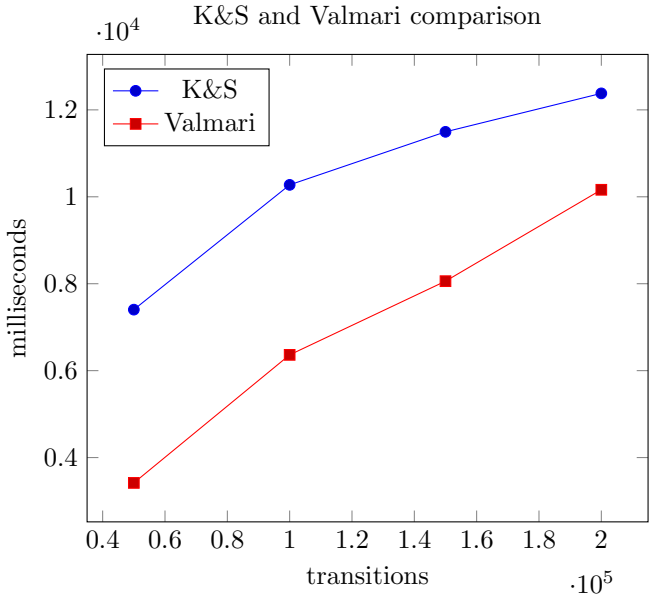


Figure 13: LTSs with 10 000 states and labels taken randomly from a set of length 10. Transitions vary on x-axis

Transitions	Time KS (ms)	Time Valmari (ms)
50 000	7403.7	3417.35
100 000	10 274.6	6363.45
150 000	11 494.85	8058.85
200 000	12 379.6	10 161.65

Table 3: Results of the test done on LTSs with 10 000 states and label taken randomly from a set of length 10. Transitions vary on the first column.

7.2 Second test

The goal of the second test is to measure the time execution of the algorithm of Kannellakis&Smolka and Valmari when we keep fixed the number of states and we change the number of transitions. For doing the tests I have built four classes of LTSs. Each class has a number of states fixed to 10 000 and transitions that vary. For each class the number of transitions and the number of samples for each class are specified by means of the Table 4 on the next page. As in the previous test for each class I have built 20 LTSs, and I have taken the arithmetic mean of the time execution in milliseconds of the 20 samples. The results are shown in Figure 13 and available in tabular form in Table 3.

As we can see when we keep fixed the number of states and vary the number of transitions, the time execution of both the algorithms grows linearly.

Class Number	Number of transitions	Number of samples
1	50 000	20
2	100 000	20
3	150 000	20
4	200 000	20

Table 4: Description of the classes for the second test.

8 BPP nets and team bisimilarity

In this Section, I describe the BPP nets, a subclass of finite Place/Transition Petri nets, and a bisimulation-based, behavioral equivalence, called *team bisimilarity*. A BPP net is a simple type of finite Place/Transition Petri net whose transitions have singleton pre-set. The description done in this section is taken from [Gor21b]. For a full description of Petri Nets, and subclasses of finite Place/Transition Petri nets, as well as the main behavioral equivalences see [Gor17].

8.1 Definitions

Definition 8.1 (*Multiset*)**.** Let $\mathbb{N}$ be the set of natural numbers. Given a finite set S, a *multiset* over S is a function $m\colon S \to \mathbb{N}$. The *support* set $dom(m)$ of m is $\{\, s \in S \mid m(s) \neq 0 \,\}$. The set of all multisets over S, denoted by $\mathscr{M}(S)$, is ranged over by m. We write $s \in m$ if $m(s) > 0$. The *multiplicity* of s in m is given by the number $m(s)$. The *size* of m, denoted by $|m|$ is the number $\sum_{s\in S} m(s)$. A multiset m such that $dom(m) = \emptyset$ is called *empty* and is denoted by θ. We write $m \subseteq m'$ if $m(s) \leq m'(s)$ for all $s \in S$.

Multiset union $\_ \oplus \_$ is defined as follows: $(m \oplus m')(s) = m(s) + m(s')$; the operation $\oplus$ is commutative, associative and has θ as neutral element. *Multiset difference* $\_ \ominus \_$ is defined as follows: $(m_1 \ominus m_2)(s) = \max\{\, m_1(s) - m_2(s), 0 \,\}$. The *scalar product* of a number j with m is the multiset $j \cdot m$ defined as $(j \cdot m)(s) = j \cdot (m(s))$.

By s_i we also denote the multiset with s_i as only element. Hence, a multiset m over $S = \{\, s_1, \ldots, s_n \,\}$ can be represented as $k_1 \cdot s_1 \oplus k_2 \cdot s_2 \oplus \cdots \oplus k_n \cdot s_n$, where $k_j = m(s_j) \geq 0$ for $j = 1, \ldots, n$.

Definition 8.2 (*BPP net*)**.** A labeled BPP net is a tuple $N = (S, A, T)$ where

- S is the finite set of *places*, ranged over by s (possibly indexed),
- A is the finite set of *labels*, ranged over by l (possibly indexed), and
- $T \subseteq S \times A \times \mathscr{M}(S)$ is the finite set of *transitions*, ranged over by t (possibly indexed).

Given a transition $t = (s, l, m)$, we use the notation:

- ${}^\bullet t$ to denote its *pre-set* s (which is a single place) of tokens to be consumed;
- $l(t)$ for its *label* l, and
- $t^\bullet$ to denote its *post-set* s (which is a multiset, possibly even empty) of tokens to be produced.

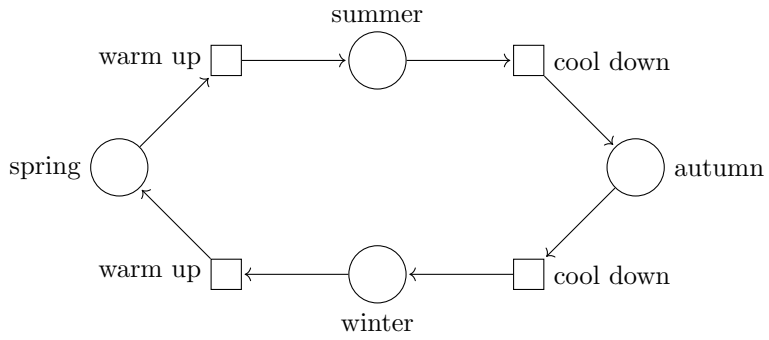


Figure 14: Four seasons

Hence, transitions t can be also represented as ${}^\bullet t \xrightarrow{l(t)} t^\bullet$. In a *BPP net* for every transition $t \in T$ we have that: $|{}^\bullet t| = 1$, that is every transition has exactly one input place.

Graphically, a place is represented by a little circle, a transition by a little box, which is connected by a directed arc from the place in its pre-set and to the places in its post-set (if any); the out-going arcs may be labeled with a number to denote the number of tokens produced by the transition (if the number is omitted, then the default value is 1).

For example in Figure 14 we have a *BPP net* that represents the four seasons of the year where

$$\begin{aligned} S &= \{ \text{ spring, summer, autumn, winter } \} \\ A &= \{ \text{ warm up, cool down } \} \\ T &= \{ \text{ (spring, warm up, summer),(summer, cool down, autumn) } \} \cup \\ &\quad \cup \{ \text{ (autumn, cool down, winter),(winter, warm up, spring) } \} \end{aligned}$$

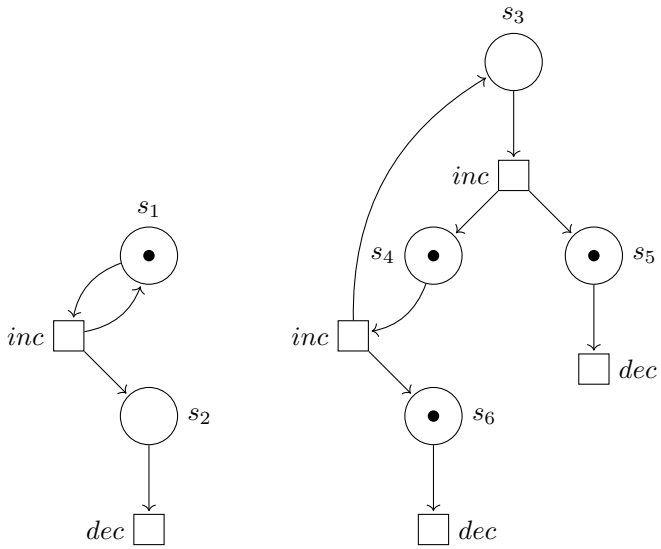
Definition 8.3 (*Marking, BPP net system, firing sequence, reachable place, dynamically reduced*)**.** A multiset over S is called a *marking*. Given a marking m and a place s, we say that the place s contains $m(s)$ *tokens*, graphically represented by $m(s)$ bullets inside place s. A *BPP net system* $N(m_0)$ is a tuple (S, A, t, m_0), where (S, A, T) is a BPP net and m_0 is a marking over S, called the *initial marking*. We also say that $N(m_0)$ is a *marked* net.

A transition t is *enabled* at marking m, denoted by $m[t\rangle$, if ${}^\bullet t \subseteq m$. The execution (or *firing*) of t enabled at m produces the marking $m' = (m \ominus {}^\bullet t) \oplus t^\bullet$. This is written $m[t\rangle m'$. This procedure is called the *token game*.

A *firing sequence* starting at m is defined inductively as follows:

- $m[\varepsilon\rangle m$ is a firing sequence (where ε denotes the empty sequence of transitions) and
- if $m[\sigma\rangle m'$ is a firing sequence and $m'[t\rangle m''$, then $m[\sigma t\rangle m''$ is a firing sequence.

If $\sigma = t_1 \dots t_n$ for $(n \geq 0)$ and $m[\sigma\rangle m'$ is a firing sequence, then there exist $m_1, \dots, m_{n+1}$ such that $m = m_1[t_1\rangle m_2[t_2\rangle \dots m_n[t_n\rangle m_{n+1} = m'$ and $\sigma =$



(a) semi-counter (b) variant

Figure 15: The net representing a semi-counter in (a), and a variant in (b)

$t_1 \dots t_n$ is called a *transition sequence* starting at m and ending at m'. The set of *reachable markings* from m is

$$[m\rangle = \{\, m' \mid \exists \sigma . m[\sigma\rangle m' \,\}\,.$$

Note that the reachable markings can be countably infinite. The set of *reachable places* from s is

$$reach(s) = \bigcup_{m \in [s\rangle} dom(m)$$

Note that $reach(s)$ is always a finite set, even if $[s\rangle$ is infinite. A BPP net system $N(m_0) = (S, A, T, m_0)$ is *dynamically reduced* if

$$\forall s \in S \, \exists m \in [m_0\rangle . m(s) \geq 1$$

and also,

$$\forall t \in T \, \exists m, m' \in [m_0\rangle \quad \text{such that} \quad m[t\rangle m'$$

Example 8.1. By using the drawing convention for Petri nets mentioned above, Figure 15a shows the simplest BPP net representing a semi-counter, that is, a counter which cannot test for zero. The number represented by this semi-counter is given by the number of tokens which are present in place s_2, that is, in the place ready to perform the action *dec*. Figure 15a represents a semi-counter holding number 0; note also that the number of tokens which can be accumulated in s_2 is unbounded. Indeed, the set of reachable markings for a BPP net can be countably infinite. In Figure 15b a variant semi-counter is outlined, which holds number 2 (that is two tokens are ready to perform action *dec*).

8.2 Additive closure

In this Section I introduce a relation $R^{\oplus} \subseteq \mathscr{M}(S) \times \mathscr{M}(S)$, where $\mathscr{M}(S)$ is defined in Definition 8.1 on page 37, that will be used for defining the *Team bisimulation*.

Definition 8.4 (*Additive closure*)**.** Given a BPP net $N = (S, A, T)$ and a place relation $R \subseteq S \times S$, we define a *marking relation* $R^{\oplus} \subseteq \mathscr{M}(S) \times \mathscr{M}(S)$, called the *additive closure* of R, as the least relation induced by the following axiom and rule.

$$\frac{}{(\theta, \theta) \in R^{\oplus}} \qquad \frac{(s_1, s_2) \in R \quad (m_1, m_2) \in R^{\oplus}}{(s_1 \oplus m_1, s_2 \oplus m_2) \in R^{\oplus}}$$

Note that, by definition, two markings are related by $R^{\oplus}$ only if they have the same size; in fact, the axiom states that the empty marking is related to itself, while the rule, assuming by induction that m_1 and m_2 have the same size, ensures that $s_1 \oplus m_1$ and $s_2 \oplus m_2$ have the same size. Note also that there may be several proofs of $(m_1, m_2) \in R^{\oplus}$ depending on the chosen order of the elements of the two markings and on the definition of R. For instance, if $R = \{ (s_1, s_3), (s_1, s_4), (s_2, s_3), (s_2, s_4) \}$ then $(s_1 \oplus s_2, s_3 \oplus s_4) \in R^{\oplus}$ can be proved by means of the pairs (s_1, s_3) and (s_2, s_4), as well as by means of $(s_1, s_4), (s_2, s_3)$. An alternative way to define that two markings m_1 and m_2 are related by $R^{\oplus}$ is to state that m_1 can be represented as $s_1 \oplus s_2 \oplus \cdots \oplus s_k$, m_2 can be represented as $s'_1 \oplus s'_2 \oplus \cdots \oplus s'_k$ and $(s_i, s'_i) \in R$ for $i = 1, \ldots, k$.

Now I list some properties of the additive closure $R^{\oplus}$.

Proposition 8.1. *For each BPP net $N = (S, A, T)$ and each place relation $R \subseteq S \times R$, if $(m_1, m_2) \in R^{\oplus}$ then $|m_1| = |m_2|$*

Proposition 8.2. *For each BPP net $N = (S, A, T)$ and each place relation $R \subseteq S \times S$ the following hold:*

1. *If R is is reflexive, then $R^{\oplus}$ is reflexive.*
2. *If R is symmetric, then $R^{\oplus}$ is symmetric.*
3. *If R is transitive, then $R^{\oplus}$ is transitive.*
4. *If $R_1 \subseteq R_2$, then $R_1^{\oplus} \subseteq R_2^{\oplus}$, that is the additive closure is monotone.*

A consequence of the proposition above is that if R is an equivalence relation, then its additive closure $R^{\oplus}$ is also an equivalence relation.

I present in the following Proposition a theorem that its necessary for implementing one of the algorithm for computing $R^{\oplus}$.

Proposition 8.3 (Additivity/subtractivity)**.** *Given a BPP net $N = (S, A, T)$ and a place relation R, the following hold:*

1. *If $(m_1, m_2) \in R^{\oplus}$ and $(m'_1, m'_2) \in R^{\oplus}$ then $(m_1 \oplus m'_1, m_2 \oplus m'_2) \in R^{\oplus}$.*
2. *If R is an equivalence relation, $(m_1 \oplus m'_1, m_2 \oplus m'_2) \in R^{\oplus}$ and $(m_1, m_2) \in R^{\oplus}$ then $(m'_1, m'_2) \in R^{\oplus}$.*

Example 8.2. The requirement that R is an equivalence relation is strictly necessary for Proposition 8.3. As a counterexample, consider

$$R = \{ (s_1, s_3), (s_1, s_4), (s_2, s_4) \}$$

We have that $(s_1 \oplus s_2, s_3 \oplus s_4) \in R^{\oplus}$ and $(s_1, s_4) \in R^{\oplus}$, but $(s_2, s_3) \notin R^{\oplus}$.

Some useful properties of additively closed place relations are the following. The proofs of these properties can be found on [Gor21b, Section 3.1].

Proposition 8.4. *For each BPP net $N = (S, A, T)$ and for each family of place relations $R_i \subseteq S \times S \, (i \in I)$, the following hold:*

1. $\emptyset^{\oplus} = \{ (\theta, \theta) \}$*, that is, the additive closure of the empty place relation is a singleton marking relation, relating the empty marking to itself.*
2. $(\mathscr{I}_S)^{\emptyset} = \mathscr{I}_M$*, that is the additive closure of the identity relation on places $\mathscr{I}_S = \{ (s, s) \mid s \in S \}$ is the identity relation on markings $\mathscr{I}_M = \{ (m, m) \mid m \in \mathscr{M}(S) \}$.*
3. $(R^{\oplus})^{-1} = (R^{-1})^{\oplus}$*, that is, the inverse of an additively closed relation R is the additive closure of its inverse R^{-1}.*
4. $(R_1 \circ R_2)^{\oplus} = (R_1^{\oplus}) \circ (R_2^{\oplus})$*, that is, the additive closure of the composition of two place relations is the compositions of their additive closures.*
5. $\bigcup_{i \in I} (R_i^{\oplus}) \subseteq (\bigcup_{i \in I} R_i)^{\oplus}$*, that is, the union of additively closed relations is included into the additive closure of their union.*

8.3 Algorithms for checking the additive closure

Given a BPP net (S, A, T), in this section I present two algorithms for checking if two marking m_1 and m_2 are related by $R^{\oplus} \subseteq \mathscr{M}(S) \times \mathscr{M}(S)$, under the condition that $R \subseteq S \times S$ is an equivalence relation.

8.3.1 First algorithm

The first algorithm is described in [Gor21b]. The description that I do in the following is taken from [Gor21a] and [Gor21b]. The algorithm, establishes whether an $R-$preserving bijection between the two markings exists, by first implementing the equivalence relation R as an adjacency matrix A of size n (the entry $A[s, s']$ is marked 1 if $(s, s') \in R$, 0 otherwise), and then by checking whether for each place/token s in m_1 there exists a place/token s' in m_2 such that the entry $A[s, s']$ is is marked 1. The complexity of this algorithm is not very high: first, the generation of the adjacency matrix takes $O(n^2)$ time, and then checking whether $(m_1, m_2) \in R^{\oplus}$ takes $O(k^2)$ time, if k is the size of m_1 and m_2. Note that if we want to perform additional team equivalence checks on the same net, we can reuse the already computed matrix A, so that the new checks will take only $O(k^2)$ time from the second check on. It is important to mention that this algorithm is correct only if R is an equivalence relation, so that $R^{\oplus}$ is is subtractive. In fact, assuming that $(m_1, m_2) \in R^{\oplus}$, when we match one place, say s_1 in m_1 with one place, say s_2 in m_2 such that $(s_1, s_2) \in R$, then

```
1  Let N = (S, A, T) be a BPP net.
2
3  Let R ⊆ S × S be a place relation, which is an equivalence.
4
5  Let A be the adjacency matrix generated as follows:
6  A[s, s'] = 1 if (s, s') ∈ R ; otherwise A[s, s'] = 0.
7
8  Let m_1 = k_1 · s_11 ⊕ k_2 · s_12 ⊕ ··· ⊕ k_{j_1} · s_{1 j_1} such that:
9  k_i > 0 for i = 1, ... j_1 and Σ_{i=1}^{j_1} k_i = k.
10 Let M_1 be an array of length j_1 such that:
11 M_1[j] = k_j, for j = 1, ..., j_1.
12
13 Let m_2 = h_1 · s_21 ⊕ h_2 · s_22 ⊕ ··· ⊕ h_{j_2} · s_{2 j_2} such that:
14 h_i > 0 for i = 1, ... j_2 and Σ_{i=1}^{j_2} h_i = k.
15 Let M_2 be an array of length j_2 such that:
16 M_2[j] = h_j, for j = 1, ..., j_2.
17
18 Let P be the set of currently matched R-related places,
19 initialized to ∅
20 for i = 1 to j_1 do
21     for j = 1 to M_1[i] do
22         h = 1
23         b = true
24         while (h ≤ j_2 and b) do
25             if M_2[h] ≠ 0 and A[s_1i, s_2h] == 1 then
26                 add (s_1i, s_2h) to P
27                 M_2[h] = M_2[h] − 1
28                 b = false
29             else
30                 h = h + 1
31             end if
32         end while
33         if h > j_2 then
34             return false
35         end if
36     end for
37 end for
38 return P
```

Listing 11: Checking the Additive Closure of an Equivalence Place Relation

```
1  Let N = (S, A, T) be a BPP net, with S = { s_1, ..., s_n }
2
3  Let m_1 and m_2 be two markings on S.
4
5  Let R ⊆ S × S be an equivalence place relation.
6
7  Let P = { B_1, ..., B_l }, 1 ≤ l ≤ n be the partition of S,
8  in the equivalence classes (called blocks) of R where:
9
10     B_i ∩ B_j = ∅ for i ≠ j,
11
12     ⋃_{i=1}^{l} B_i = S,
13
14     ∀s, s' ∈ B_i  (s, s') ∈ R for i = 1, ..., l and, finally
15
16     ∀s ∈ B_i, ∀s' ∈ B_j if i ≠ j, then (s, s') ∉ R.
17
18 Let count_1, count_2 be two integer variables
19 for all blocks in P do
20     count_1, count_2 = 0
21     for all places s in the current block do
22         count_1 = count_1 + m_1(s)
23         count_2 = count_2 + m_2(s)
24     end for
25     if not count_1 == count_2 then
26         return false
27     end if
28 end for
29 return true
```

Listing 12: Algorithm for checking whether $(m_1, m_2) \in R^{\oplus}$

we need that also $(m_1 \ominus s_1, m_2 \ominus s_2) \in R^{\oplus}$ (cf. Example 8.2). The pseudo-code is available in Listing 11 on the previous page.

Of course, two markings m_1 and m_2 are not team bisimilar if they have different size, or if the Algorithm described in this Section fails by singling out a place s in the residual of m_1 (that is, in the portion of m_1 which has not been scanned yet) which has no matching team bisimilar place in (the residual of) m_2.

8.3.2 Second algorithm

The second algorithm is described in [Gor21a]. The algorithm described in [Gor21a] is a slight generalization of the algorithm proposed originally in [Lib19]. The description that I do in the following is taken from [Gor21a].

The algorithm checks whether $(m_1, m_2) \in R^{\oplus}$ simply by checking if, for each equivalence class of R, the number of places/tokens of m_1 in that class equals the number of places/tokens of m_2 in the same class. In this way, we are sure that there is an R-preserving bijective mapping between the two markings. The complexity of this new algorithm is $O(n)$, because we have essentially to scan all the (equivalence classes and then the) places (in these classes), and this complexity holds already for the first check. Therefore, this new algorithm is better than the one described in Section 8.3.1 on page 41, while it may be less performant than the original one, from the second check onwards, only if the

markings are small compared to the size of the net: more precisely, if $k < \sqrt{n}$. The pseudo-code is available in Listing 12 on the previous page. The reason why this second algorithm usually outperforms the one described in Section 8.3.1 is that, by exploiting the partition of S induced by R, there is no need to build any auxiliary data structure for representing R.

Two markings m_1 and m_2 are not team bisimilar, if for some equivalence class B of $\sim$, the number of all the tokens in the places of m_1 belonging to B is different from the number of all the tokens in the places of m_2 belonging to B.

8.4 Team bisimulation on places

Now that we have defined the additive closure $R^{\oplus}$ we are ready to define the *Team bisimulation on places*.

Definition 8.5 (*Team bisimulation*)**.** Let $N = (S, A, T)$ be a BPP net. A *team bisimulation* is a place relation $R \subseteq S \times S$ such that if $(s_1, s_2) \in R$ then for all $l \in A$

- $\forall m_1$ such that $s_1 \xrightarrow{l} m_1$, $\exists m_2$ such that $s_2 \xrightarrow{l} m_2$ and $(m_1, m_2) \in R^{\oplus}$,
- $\forall m_2$ such that $s_2 \xrightarrow{l} m_2$, $\exists m_1$ such that $s_1 \xrightarrow{l} m_1$ and $(m_1, m_2) \in R^{\oplus}$.

Two places s and s' are *team bisimilar* (or *team bisimulation equivalent*), denoted $s \sim s'$, if there exists a team bisimulation R such that $(s, s') \in R$

Example 8.3. If we consider the BPP nets in Figure 15 on page 39, it is easy to see that relation:

$$R = \{ (s_1, s_3), (s_1, s_4), (s_2, s_5), (s_2, s_6) \}$$

is a team bisimulation. In fact, the pair (s_1, s_3) is a team bisimulation pair because, to transition $s_1 \xrightarrow{inc} s_1 \oplus s_2$, s_3 can respond with $s_3 \xrightarrow{inc} s_4 \oplus s_5$, and $(s_1 \oplus s_2, s_4 \oplus s_5) \in R^{\oplus}$; symmetrically, if s_3 moves first. Also the pair (s_1, s_4) is a team bisimulation pair because, to transition $s_1 \xrightarrow{inc} s_1 \oplus s_2$, s_4 can respond with $s_4 \xrightarrow{inc} s_3 \oplus s_6$ and $(s_1 \oplus s_2, s_3 \oplus s_6) \in R^{\oplus}$; symmetrically, if s_4 moves first. Also the pair (s_2, s_5) is a team bisimulation pair: to transition $s_2 \xrightarrow{dec} \theta$, s_5 responds with $s_5 \xrightarrow{dec} \theta$, and $(\theta, \theta) \in R^{\oplus}$. Similarly for the pair (s_2, s_6). Hence, relation R is a team bisimulation, indeed.

Example 8.4. Consider the nets in Figure 16 on the next page. It is easy to realize that relation

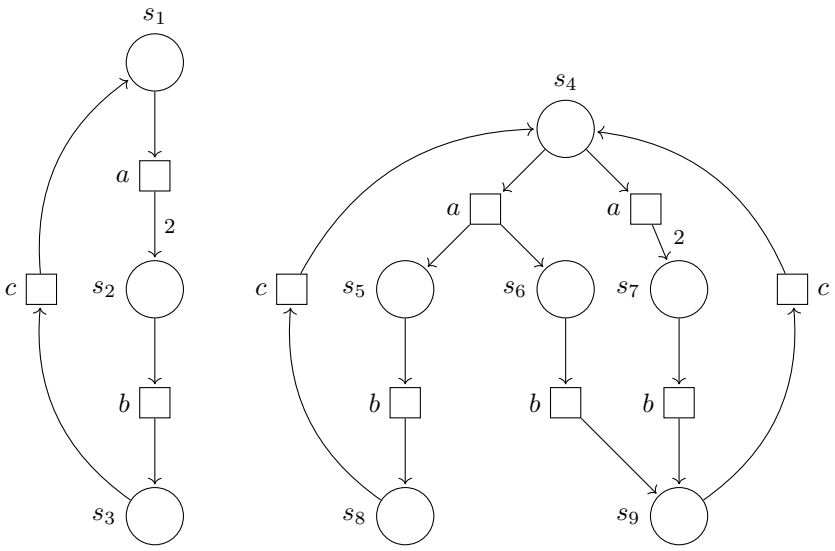
$$R = \{ (s_1, s_4), (s_2, s_5), (s_2, s_6), (s_2, s_7), (s_3, s_8), (s_3, s_9) \}$$

is a team bisimulation.

Now I list some properties of team bisimulation relations. The proofs of these properties can be found on [Gor21b, Section 3.2].

Proposition 8.5. *For each BPP net $N = (S, A, T)$, the following hold:*

1. *The identity relation $\mathscr{I}_S = \{ (s, s) \mid s \in S \}$ is a team bisimulation;*



(a) A BPP net (b) Another BPP net

Figure 16: Two team bisimilar BPP nets

2. *the inverse relation* $R^{-1} = \{ (s', s) \mid (s, s') \in R \}$ *of a team bisimulation* R *is a team bisimulation;*

3. *the relational composition*

$$R_1 \circ R_2 = \{ (s, s'') \mid \exists s'.(s, s') \in R_1 \wedge (s', s'') \in R_2 \}$$

of two team bisimulations R_1 *and* R_2 *is a team bisimulation;*

4. *the union* $\bigcup_{i \in I} R_i$ *of team bisimulations* R_i *is a team bisimulation.*

As stated in Definition 8.5 on the preceding page, given two places s and s' we have that $s \sim s'$, if there exists a team bisimulation containing the pair (s, s'). This means that $\sim$ is the union of all team bisimulations, that is,

$$\sim = \bigcup \{ R \subseteq S \times S \mid R \text{ is a team bisimulation} \}. \tag{2}$$

By Proposition 8.5, point 4, $\sim$ is also a team bisimulation, hence the largest such relation.

Proposition 8.6. *For each BPP net* $N = (S, A, T)$*, relation* $\sim \subseteq S \times S$ *is the largest team bisimulation relation.*

A team bisimulation relation need not be reflexive, symmetric, or transitive. Nonetheless, the largest team bisimulation relation $\sim$ is an equivalence relation. As a matter of fact, as the identity relation $\mathscr{I}_S$ is a team bisimulation by Proposition 8.5, point 1, we have that $\mathscr{I}_S \subseteq \sim$, and so $\sim$ is reflexive. Symmetry derives from the following argument. For any $(s, s') \in \sim$, there exists a team bisimulation R such that $(s, s') \in R$; by Proposition 8.5, point 2, relation R^{-1} is a team bisimulation containing the pair (s', s); hence, $(s', s) \in \sim$ because

$R^{-1} \subseteq \sim$. Transitivity also holds for $\sim$. Assume $(s, s') \in \sim$ and $(s', s'') \in \sim$; hence, there exist two team bisimulations R_1 and R_2 such that $(s, s') \in R_1$ and $(s', s'') \in R_2$; by Proposition 8.5, point 3, relation $R_1 \circ R_2$ is a team bisimulation containing the pair (s, s''); hence, $(s, s'') \in \sim$, because $R_1 \circ R_2 \subseteq \sim$. Summing up, we have the following.

Proposition 8.7. *For each BPP net $N = (S, A, T)$, relation $\sim \subseteq S \times S$ is an equivalence relation.*

8.5 Team bisimilarity over markings

Starting from team bisimilarity $\sim$, which has been computed over the places of an *unmarked* BPP net, we can extend team bisimulation equivalence over its markings in a distributed way: m_1 is team bisimulation equivalent to m_2 if they are related by the additive closure of $\sim$, that is, if $(m_1, m_2) \in \sim^{\oplus}$, usually denoted by $m_1 \sim^{\oplus} m_2$.

If team bisimilarity $\sim$ is implemented as a matrix A such that

$$A[s, s'] = \begin{cases} 1 & \text{if } s \sim s' \\ 0 & \text{if } s \not\sim s' \end{cases}$$

then for checking if $(m_1, m_2) \in \sim^{\oplus}$ we can use the algorithm outlined in Section 8.3.1 on page 41. Otherwise if $\sim$ is implemented by equivalence classes we can use the algorithm outlined in Section 8.3.2 on page 43.

In the following I list some properties of $\sim^{\oplus}$.

Proposition 8.8. *For each BPP net $N = (S, A, T)$, if $m_1 \sim^{\oplus} m_2$, then $|m_1| = |m_2|$*

Proposition 8.9. *For each BPP net $N = (S, A, T)$, relation $\sim^{\oplus} \subseteq \mathscr{M}(S) \times \mathscr{M}(S)$ is an equivalence relation.*

Example 8.5. If we take the semi-counter depicted in Figure 15 on page 39, the marking $s_1 \oplus 2 \cdot s_2$ is team bisimilar to the following markings of the net in (b): $s_3 \oplus 2 \cdot s_5$, or $s_3 \oplus s_5 \oplus s_6$, or $s_3 \oplus 2 \cdot s_6$, or $s_4 \oplus 2 \cdot s_5$, or $s_4 \oplus s_5 \oplus s_6$, or $s_4 \oplus 2 \cdot s_6$.

Example 8.6. If we take the two BPP nets depicted in Figure 16 on the preceding page, it is clear that, for instance, $s_1 \oplus 3 \cdot s_2$ is team bisimilar to any marking obtained with one token on place s_4 and three tokens distributed over the places s_5, s_6 and s_7; for example $s_1 \oplus 3 \cdot s_2 \sim^{\oplus} s_4 \oplus 2 \cdot s_5 \oplus s_7$ or $s_1 \oplus 3 \cdot s_2 \sim^{\oplus} s_4 \oplus s_6 \oplus 2 \cdot s_7$.

9 Team bisimilarity over places as a fixed point

In this Section I describe Team bisimilarity over places as a fixed point. The description done in this Section is taken from [Gor21b, Section 3.4].

The *Team bisimulation equivalence over places* can be characterized nicely as the greatest fixed point of a suitable monotone relation transformer, essentially by extending the characterization developed for ordinary bisimulation over LTSs done in Section 3 on page 5.

Even if the discussion done in this Section is similar to the one done in Section 3 on page 5 for LTSs, for the sake of clarity I will report it anyway.

Definition 9.1. Given a BPP net $N = (S, A, T)$, the functional $F : \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ (i.e., a transformer of binary relations over S) is defined as follows. If $R \subseteq S \times S$, then $(s_1, s_2) \in F(R)$ if and only if for all $l \in A$

- $\forall m_1$ such that $s_1 \xrightarrow{l} m_1$, $\exists m_2$ such that $s_2 \xrightarrow{l} m_2$ and $(m_1, m_2) \in R^{\oplus}$
- $\forall m_2$ such that $s_2 \xrightarrow{l} m_2$, $\exists m_1$ such that $s_1 \xrightarrow{l} m_1$ and $(m_1, m_2) \in R^{\oplus}$

As we can see the Definition 9.1 is similar to the one described in Definition 3.1 on page 7. The only difference is that in the case of BPP net the two markings m_1 and m_2 must belong to $R^{\oplus}$.

As in the case of LTSs we have that

Proposition 9.1. *For each BPP net $N = (S, A, T)$, we have that:*

1. *The functional F is monotone, that is, if $R_1 \subseteq R_2$ then $F(R_1) \subseteq F(R_2)$.*
2. *A relation $R \subseteq S \times S$ is a team bisimulation if and only if $R \subseteq F(R)$.*

As stated in Section 3.5 on page 6 a *fixed point* for F is a relation R such that $R = F(R)$. Knaster–Tarski's fixed point theorem, outlined in Theorem 3.1 on page 7 and described in [Ace+07, p. 80], ensures that the greatest fixed point of the monotone functional F is

$$\bigcup \{\, R \subseteq S \times S \mid R \subseteq F(R) \,\}$$

It is possible to show that this greatest fixed point is $\sim$. $\sim$ is defined in Equation 2 on page 45. As stated in Section 3.5 on page 6, a *post-fixed point* of F is a relation R such that $R \subseteq F(R)$. By Proposition 9.1, point 2, we know that the team bisimulations are the post-fixed points of F. As we can see in Equation 2 on page 45 team bisimilarity $\sim$ is the union of all the team bisimulations. Hence, we conclude that $\sim$ is the greatest fixed point of F, that is

$$\sim = \bigcup \{\, R \subseteq S \times S \mid R \subseteq F(R) \,\}$$

The following theorem provides a direct proof of this fact.

Theorem 9.1. *Team bisimilarity $\sim$ is the greatest fixed point of F.*

Proof. We first prove that $\sim$ is a fixed point, that is, $\sim = F(\sim)$, by proving that $\sim \subseteq F(\sim)$ and that $F(\sim) \subseteq \sim$. Since $\sim$ is a team bisimulation, $\sim \subseteq F(\sim)$ by Proposition 9.1, point 2. As F is monotonic, by Proposition 9.1, point 1, we have that $F(\sim) \subseteq F(F(\sim))$, that is, also $F(\sim)$ is a post-fixed point of F i.e., a team bisimulation. Since we know that $\sim$ is the union of all team bisimulation relations (as well as the greatest post-fixed point of F), it follows that $F(\sim) \subseteq \sim$.

Now we want to show that $\sim$ is the greatest fixed point. Assume T is another fixed point of F, i.e. $T = F(T)$. Then, in particular, we have that $T \subseteq F(T)$, i.e., T is a team bisimulation by Proposition 9.1, point 2, hence $T \subseteq \sim$ □

There is a natural iterative way of approximating $\sim$ by means of a descending (actually, initially descending, and then constant from a certain point onwards) chain of relations indexed on the natural numbers. We will see that there is a strict relation between this chain of relations and the functional F above.

Definition 9.2. Given a BPP net $N = (S, A, T)$, for each natural $i \in \mathbb{N}$, we define the binary relation $\sim_i$ over S as follows:

- $\sim_0 = S \times S$.
- $s_1 \sim_{i+1} s_2$ if and only if for all $l \in A$
  - $\forall m_1$ such that $s_1 \xrightarrow{l} m_1$, $\exists m_2$ such that $s_2 \xrightarrow{l} m_2$ and $m_1 \sim_i^{\oplus} m_2$
  - $\forall m_2$ such that $s_2 \xrightarrow{l} m_2$, $\exists m_1$ such that $s_1 \xrightarrow{l} m_1$ and $m_1 \sim_i^{\oplus} m_2$

We denote by $\sim_\omega$ the relation $\bigcup_{i \in \mathbb{N}} \sim_i$.

Intuitively, $s_1 \sim_i s_2$ if and only if the two places are team bisimilar up to paths of length at most i. Hence, all the places are in the relation $\sim_0$.

Proposition 9.2. *For each $i \in \mathbb{N}$ we have that:*

- *relation $\sim_i$ is an equivalence relation,*
- $\sim_i = F^i(S \times S)$
- $\sim_{i+1} \subseteq \sim_i$

Moreover, $\sim_\omega = \bigcap_{i \in \mathbb{N}} \sim_i$ is an equivalence relation.

Hence, we have a non-increasing chain of equivalence relations,

$$\sim_0 = F^0(S \times S) \supseteq \sim_1 = F^1(S \times S) \supseteq \cdots \supseteq \sim_i = F^i(S \times S) \supseteq \cdots \supseteq \sim_\omega$$

with relation $\sim_\omega$ as its limit. Interestingly, this limit coincides with team bisimilarity $\sim$, as proved below. Some auxiliary lemmata are needed.

Lemma 9.1. *For each BPP net $N = (S, A, T)$, it holds that there exists an index k such that $\sim_k = \sim_{k+1} = \cdots = \sim_\omega$, i.e., the chain is initially decreasing, but becomes constant from index k onwards.*

Proof. Since the BPP net is finite, the initial relation $\sim_0 = S \times S$ is finite as well. Therefore, it is not possible that $\sim_i = F^i(S \times S) \supset \sim_{i+1}$ for all $i \in \mathbb{N}$. This means that there exists an index k such that $\sim_k = F^k(S \times S) = F(F^k(S \times S)) = \sim_{k+1}$. Hence $\sim_k = \sim_j$ for each $j > k$, and so $\sim_k = \sim_\omega$. □

Theorem 9.2. *For each BPP net $N = (S, A, T)$, it holds that $\sim = \sim_\omega$.*

Proof. We prove first that $\sim \subseteq \sim_i$ for all i by induction on i. Indeed, $\sim \subseteq \sim_0$ (the universal relation); moreover, assuming $\sim \subseteq_i$, by monotonicity of F and the fact that $\sim$ is a fixed point for F, we get $\sim = F(\sim) \subseteq F(\sim_i) = \sim_{i+1}$. Hence $\sim \subseteq \sim_\omega$.

Now we prove that $\sim_\omega \subseteq \sim$, by showing that relation $\sim_\omega$ is a team bisimulation. Indeed by Lemma 9.1, we know that $\sim_\omega = \sim_k$ for some $k \in \mathbb{N}$. As $\sim_{k+1} = F(\sim_k) = \sim_k$, we have that $\sim_k$, thanks to Definition 9.2, satisfies Definition 9.1 on the preceding page, so that, by Preposition 9.1 on the previous page, point 2, $\sim_k$ is a team bisimulation. □

The characterization of $\sim$ as the limit of the non-increasing chain of relations $\sim_i$ offers an easy algorithm to compute team bisimilarity $\sim$ over BPP nets; just start from the universal relation $R_0 = S \times S$ and then iteratively apply functional F; when $R_{i+1} = F(R_i) = R_i$, then stop and take R_i as the team bisimilarity

relation. The aforementioned approach is the same followed for computing the bisimulation equivalence on finite labeled transition system, outlined in Listing 1 on page 8. Of course, this algorithm always terminates by the argument in Lemma 9.1 on the previous page: since S is finite, we are sure that an index k exists such that $R_{k+1} = F(R_k) = R_k$. As I said before the algorithm is the same as the one in Listing 1 on page 8.

9.1 Implementation

For computing the *team bisimulation equivalence* through the fixed point approach on BPP nets, I have followed the same procedure for computing the *bisimulation equivalence* on labeled transition systems, through the fixed point approach, outlined in Section 6.1 on page 25. The only change that I have done is to take into account the additive closure $R^{\oplus}$. In the following, I will describe the algorithm outlined in Section 6.1 on page 25 with the changes done for taking into account the additive closure $R^{\oplus}$.

Given a BPP net $N = (S, A, T)$, the implementation simply, starts with the relation $R = S \times S$, where R is the Cartesian product of the state S with itself. Then iterates until -under certain conditions- no other couples $(s_1, s_2) \in R$ can be removed. The conditions that the couples $(s_1, s_2) \in R$ must satisfy are the ones defined in Definition 8.5 on page 44.

I have implemented the relation $R = S \times S$ -and the subsets of R- as a List of couples $(s_1, s_2) \in R$ where $s_1 \in S$ and $s_2 \in S$. The BPP net in this algorithm is implemented as a class that has one, fundamental, field: a vector v that takes in input a place $s \in S$ and an action $a \in A$ and returns the list of multisets that are reachable from the place s, when s does the action a. The vector v takes in input a place s and an action a as numbers so the class used for implementing the BPP nets, has two additional fields that are two maps. The first map takes in input a place $s \in S$ and gives in output a number associated to the state s, the second map takes in input an action $a \in A$, and gives in output a number associated to the action a.

9.1.1 Detailed steps

For computing the *team bisimulation equivalence* the algorithm initially sets R to $S \times S$. R is, at the beginning, the list of couples (s_1, s_2) -with $s_1 \in S$ and $s_2 \in S$- of length n^2 where $n = |S|$.

For each couple $(s_1, s_2) \in R$ the algorithm does the following: for each label $a \in A$:

1. Builds a list `l1` of multisets. The list `l1` contains all the multisets that the place s_1 reaches by doing an action a. The list `l1` is built by means of vector v described above. The vector v takes in input the place s_1 and the action a and returns `l1`

2. Builds a list `l2` of multisets. The list `l2` contains all the multisets that the place s_2 reaches by doing an action a. The list `l2` is built by means of vector v described above. The vector v takes in input the place s_2 and the action a and returns `l2`

3. For each multiset $m_1 \in \texttt{l1}$, the algorithm checks if exists a multiset $m_2 \in \texttt{l2}$, such that $(m_1, m_2) \in R^{\oplus}$. If $(m_1, m_2) \in R^{\oplus}$ return true else false.

4. Similarly to the previous point, for each multiset $m_2 \in \texttt{l2}$, the algorithm checks if exists a multiset $m_1 \in \texttt{l1}$, such that $(m_1, m_2) \in R^{\oplus}$. If $(m_1, m_2) \in R^{\oplus}$ return true else false.

Then if for all $a \in A$ both the checks in point 3 and 4 of the list above returns true, the the couple (s_1, s_2) can stay in the relation R, otherwise it will be removed.

When there is no more couples to be removed from R the algorithm ends and returns the relation R, that contains the team bisimulation equivalence $\sim$.

Given two multisets m_1 and m_2, for checking if $(m_1, m_2) \in R^{\oplus}$, I have implemented the Algorithm described in Section 8.3.1 on page 41.

For checking if a couple (s_1, s_2) belongs to R or not in $O(1)$ time, as required by the Algorithm described in Section 8.3.1 on page 41, I have implemented a matrix m. The matrix m takes in input a couple of places (s_1, s_2), with $s_1 \in S$ and $s_2 \in S$, and returns true if $(s_1, s_2) \in R$, false otherwise. The matrix m corresponds to the matrix A in the pseudo-code available in Listing 11 on page 42.

When the algorithm finishes to scan the relation R, divides the relation R in two relations: R_1 and R_2. R_1 is the set of couples that **satisfies** the point 3 and 4 of the list above, R_2 is the set of couples that **does not satisfies** the point 3 and 4 of the list above. The couples $(s_1, s_2) \in R_2$ will be marked as false in the matrix m described in the previous paragraph.

I have implemented the steps described above through the Scala programming language. The code is available at [Bar21].

9.1.2 Time complexity

The discussion about the time complexity of computing the *team bisimulation equivalence* through the fixed point approach on BPP nets, is similar the discussion done for computing the bisimulation equivalence on LTSs through the fixed point approach, done in Section 6.1.2 on page 26. The only difference is that we have to take into account the cost for checking if $(m_1, m_2) \in R^{\oplus}$, where m_1 and m_2 are two multisets and R is an equivalence relation.

The steps are the following:

1. Initially the length of the relation $R = S \times S$ is equal to n^2 because $n = |S|$. In the worst case, the algorithm, at every step removes one couple $(s_1, s_2) \in R$ at a time so this loop -that we call `loop1`- costs in the worst case $O(n^2)$.

2. At every step of `loop1` we we have to iterate over all the couples (s_1, s_2) of the relation R, for deciding if (s_1, s_2) can stay in R or not. This loop -that we call `loop2`- also costs in the worst case $O(n^2)$.

3. For deciding if a couple (s_1, s_2) can stay in R, we have to build the lists `l1` and `l2` described in Section 9.1.1 on the preceding page, for every $a \in A$. Next, for every multiset $m_1 \in \texttt{l1}$, we have to search if exists in `l2` a multiset $m_2 \in \texttt{l2}$ such that $(m_1, m_2) \in R^{\oplus}$. That is, given an

action $a \in A$, for every multiset m'_1 such that $s_1 \xrightarrow{a} m'_1$ exists a multiset m'_2 such that $s_2 \xrightarrow{a} m'_2$ and $(m_1, m_2) \in R^{\oplus}$. The cost of checking if $(m_1, m_2) \in R^{\oplus}$ is $O(k^2)$ where k is the size of m_1 and m_2 because we have employed the Algorithm outlined in Section 8.3.1 on page 41 for checking if $(m_1, m_2) \in R^{\oplus}$. If we say that the length of the list `l1` of multisets is i and i can be at most m, where m is the number of transitions, for every multiset $m_1 \in$ `l1` we have to iterate over all the multiset $m_2 \in$ `l2`. This costs $O(i \cdot j \cdot p^2)$ where j is the length of the list of multiset `l2` and p is the least number such that $|t^{\bullet}| \leq p$ for all the transition $t \in T$.

So if $i \leq m$ and $j \leq m$ where i and j are respectively the length of `l1` and `l2`, and p is the least number such that $|t^{\bullet}| \leq p$ for all the transition $t \in T$, deciding if a couple of places (s_1, s_2) can stay in R costs $O(m^2 \cdot p^2)$, because for every multiset m_1 that belongs to `l1` we have to iterate over all the multiset m_2 that belongs to `l2` and then check if $(m_1, m_2) \in R^{\oplus}$. I have omitted the case when s_2 moves first, that is the case when for every multisets m_2 that belongs to `l2` we have to search if exists a suitable multisets m_1 that belongs to `l1` and then cheking if $(m_1, m_2) \in R^{\oplus}$, but it is symmetric to the case when s_1 moves first, hence costs $O(m^2 \cdot p^2)$.

Considering that the length of `l1` can be at most m where m is the number of transitions f the BPP net is a pessimistic reasoning because `l1` represents the transitions that -given a place $s_1 \in S$ and an action $a \in A$- starts from s_1 and by doing an action a end in a multiset $m' \in \mathscr{M}(S)$. The length of `l1` is certainly less than m. Same for the list `l2`.

In a random BPP net we may consider that the length of `l1` and `l2` be at most $\frac{m}{n}$, where n is the number of places of the BPP net, because the transitions that start from a random state q -given an action a- if the transitions are set randomly are seldom m. If the transitions that start from a state q are m it means that all the transitions in the BPP net start from q. So if $i \leq \frac{m}{n}$ and $j \leq \frac{m}{n}$, where i and j are the length of `l1` and `l2` respectively we have that the cost of deciding if a couple (s_1, s_2) can stay in R is $O((\frac{m}{n})^2 \cdot p^2)$, where p is the least number such that $|t^{\bullet}| \leq p$ for all the transition $t \in T$. The factor p^2 is the cost of checking if $(m_1, m_2) \in R^{\oplus}$, through the algorithm described in Section 8.3.1 on page 41.

For what we have said above the time complexity of the implementation that I have done for computing the time bisimulation equivalence through the fixed point approach is:

1. $O(n^2 \cdot n^2 \cdot 2(\frac{m}{n})^2 \cdot p^2) = O(n^2 \cdot m^2 \cdot p^2)$ if we consider, the length of `l1` and `l2` be at most $\frac{m}{n}$.
2. Otherwise if we consider the length of `l1` and `l2` be at most $O(m)$ the time complexity is $O(n^2 \cdot n^2 \cdot m^2 \cdot p^2)$.

10 K&S's algorithm for team bisimilarity

In this section, I explain the steps followed in order to implement the Kannellakis and Smolka's algorithm for computing team bisimilarity over places through the Scala programming language.

For the purpose of this Section we can think of a partition as a list of blocks, and of a block as a list of places.

I have implemented a BPP net by means of a vector v, that takes in input a place s and an action a and gives in output the list of multisets reachable from s, when s does an action a. That is, when v takes in input a place s and an action a gives in output all the multiset $m \in \mathscr{M}(S)$ such that $(s, a, m) \in T$, where T is the set of transition of the BPP net on which we have to compute the team bisimulation equivalence. The algorithm also uses a vector `indexes` of length n, where n is the number of places of the BPP net. The vector `indexes` takes in input a place p and return the index of the block B, that contains p in the partition. That is, given a partition $\pi = \{ B_1, \ldots, B_i, \ldots, B_n \}$ and a place $p \in B_i$, `indexes`(p) returns i. The program also uses a variable `numBlocks` that records the number of Blocks in the partition. In addition the BPP net has two fields that i call $\texttt{map}_1$ and $\texttt{map}_2$; $\texttt{map}_1$ maps each place $s \in S$ to an ID, $\texttt{map}_2$ maps each label $a \in A$ to an ID. The IDs are represented as natural numbers. These map are necessary because the vectors v and `numBlocks` takes their input as a natural number.

10.1 Implementation

Given a BPP net (S, A, T), the algorithm starts with a partition π that contains a unique block B. B contains all places of the BPP net, that is B is equal to S. Next the algorithm scans one at a time each block B in the partition π and does the following: for each $a \in A$:

1. Create two empty set B_1 and B_2
2. Select a place $s \in B$ and by means of the vector v get the list of all multisets reachable from s when s does an action a, this list is called `l1`. The vector v takes in input s and a and gives in output `l1`.
3. For each multiset m_i that belongs to `l1` create a vector $\texttt{vec}_\texttt{i}$ of length $\texttt{numBlocks} + 1$ with all elements of $\texttt{vec}_\texttt{i}$ set to 0, that is

   $$\forall k \,\text{such that}\, 0 \leq k \leq \texttt{numBlocks} + 1 \,\text{we have that}\, \texttt{vec}_\texttt{i}(\texttt{k}) = 0$$

   The vector $\texttt{vec}_\texttt{i}$ has length $\texttt{numBlocks} + 1$ because the vectors in Scala starts at index 0. The element $\texttt{vec}_\texttt{i}(j)$ contains the number of places/tokens of m_i that belong to the block j. In my implementation the blocks are numbered starting from 1, this is the reason why $\texttt{vec}_\texttt{i}$ has length $\texttt{numBlocks} + 1$. After the vector $\texttt{vec}_\texttt{i}$ has been created, update $\texttt{vec}_\texttt{i}$ in the following way:

   $$\texttt{for each p} \in \texttt{dom}(\texttt{m}_\texttt{i})\texttt{:}\ \texttt{vec}_\texttt{i}(\texttt{indexes}(\texttt{p})) = \texttt{vec}_\texttt{i}(\texttt{indexes}(\texttt{p})) + \texttt{m}_\texttt{i}(\texttt{p})$$

   Given a place $p \in S$, `index`$(\texttt{p})$ returns the number i of the block such that $p \in B_i$.

   When all the $\texttt{vec}_\texttt{i}$ has been built from all the multisets $m_i \in \texttt{l1}$, put all the $\texttt{vec}_\texttt{i}$ in a list called $\texttt{markBlock}_\texttt{s}$. The list $\texttt{markBlock}_\texttt{s}$ is a list of vector of the same length, that is for all i and j such that $\texttt{vec}_\texttt{i}, \texttt{vec}_\texttt{j} \in \texttt{markBlock}_\texttt{s}$, we have that $|\texttt{vec}_\texttt{i}| = |\texttt{vec}_\texttt{j}| = \texttt{numBlocks} + 1$. After $\texttt{markBlock}_\texttt{s}$ has been

created, sort $\texttt{markBlock}_\texttt{s}$ by lexicographic order. That is, given two vector $\texttt{vec}_\texttt{i} = e_{i_1}, \ldots, e_{i_k}$ and $\texttt{vec}_\texttt{j} = e_{j_1}, \ldots, e_{j_k}$, let l the first position where $\texttt{vec}_\texttt{i}$ and $\texttt{vec}_\texttt{j}$ differ, we say that $\texttt{vec}_\texttt{i} < \texttt{vec}_\texttt{j}$ if ond only if: $e_{i_l} < e_{j_l}$. After the list $\texttt{markBlock}_\texttt{s}$ has been sorted, remove the duplicates from it.

4. For each place $p \in B$ create $\texttt{markBlock}_\texttt{p}$ in the same way as $\texttt{markBlock}_\texttt{s}$ has been created in the two previous points.

5. For all $p \in B$, if $\texttt{markBlock}_\texttt{s}$ is equal to $\texttt{markBlock}_\texttt{p}$, add p to the set B_1, otherwise add p to the set B_2.

6. If B_2 is not empty replace B with B_1 and B_2 in the partition P, and update `numBlocks` with $\texttt{numBlocks} + 1$ and for all the places that belong to B_2, update the vector `indexes` in this way:

$$\forall \,\text{place}\, s \in B_2 \; \texttt{indexes(s)} = \texttt{numBlocks}$$

If B_2 is empty go ahead with the next label -indexed with a-. If there is no more label to check go ahead with the next block and repeat the procedure in this list starting over to scan all the labels. If there is no more block to check stop and give in output the partition π that now contains the team bisimulation equivalence $\sim$.

The code is available at [Bar21].

10.2 Time complexity

For defining the time complexity of computing the team bisimulation equivalence on a BPP net (S, A, T) through the implementation of Kannellakis and Smolka's algorithm defined in Section 10.1 on the previous page, we have to do the following considerations:

1. The number of blocks that can be created is at most n, where n is the number of places, because a block composed of one place cannot be further split.

2. When we look for a block to be split, in the worst case we have to look for all the blocks in the partition, so we have to build the list of vectors $\texttt{markBlock}_\texttt{s}$, defined in point 3 of the previous list, for each place $s \in S$. In order to build the vectors $\texttt{markBlock}_\texttt{s}$ for each place $s \in S$ we have to scan all transitions, and this takes $O(m)$ and then for each multiset $m1$, we have to build a vector, like the vectors $\texttt{vec}_\texttt{i}$ in point 3 of the previous list, that records the number of places/tokens of $m1$ that belongs to each block of the partition and this takes $O(n)$, because the number of places/tokens in a multiset can be at most n. Next, we have to sort the vectors $\texttt{markBlock}_\texttt{s}$ and deletes the duplicates, this depends on the number of the vectors $\texttt{markBlock}_\texttt{s}$ and on the length of each vector $\texttt{markBlock}_\texttt{s}$. Moreover, in order to decide if $vec_i < vec_j$, where vec_i and vec_j are two vectors that belongs to $\texttt{markBlock}_\texttt{s}$ we have to scan vec_i and vec_j and this takes $O(n)$, because the number of blocks can be at most n, where n is the number of places. Let α the number of vectors $\texttt{markBlock}_\texttt{s}$ and β the maximum length among the vectors $\texttt{markBlock}_\texttt{s}$. We have that the time complexity of computing the team bisimulation equivalence through the implementation in this section is $O(n \cdot ((m \cdot n) + \alpha \cdot \beta \log \beta \cdot n))$.

11 Conclusions and future work

In this thesis, I have implemented three algorithms for computing bisimulation equivalence on labeled transition system and two algorithms for computing *team bisimulation equivalence* on BPP nets. The language that I have used is Scala [OSV16].

As future work, there is the possibility to lower the time complexity of computing the team bisimulation, by adapting the algorithm described in Section 5 on page 14.

References

[PT87] Robert Paige and Robert E. Tarjan. "Three Partition Refinement Algorithms". In: *SIAM Journal on Computing* 16.6 (1987), pp. 973–989. DOI: 10.1137/0216062. eprint: https://doi.org/10.1137/0216062. URL: https://doi.org/10.1137/0216062.

[KS90] Paris C. Kanellakis and Scott A. Smolka. "CCS expressions, finite state processes, and three problems of equivalence". In: *Information and Computation* 86.1 (1990), pp. 43–68. ISSN: 0890-5401. DOI: 10/dn276s. URL: https://www.sciencedirect.com/science/article/pii/089054019090025D.

[Ace+07] Luca Aceto et al. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007. DOI: 10/bf3c4z. URL: https://doi.org/10.1017/cbo9780511814105.

[Val09] Antti Valmari. "Bisimilarity Minimization in O(m logn) Time". In: June 2009, pp. 123–142. ISBN: 978-3-642-02423-8. DOI: 10.1007/978-3-642-02424-5_9.

[AIS11] Luca Aceto, Anna Ingolfsdottir, and Jiri Srba. "The Algorithmics of Bisimilarity". English. In: *Advanced Topics in Bisimulation and Coinduction*. Vol. 52. Cambridge Tracts in Theoretical Computer Science 52. United Kingdom: Cambridge University Press, 2011, pp. 100–172. ISBN: 9781107004979. URL: https://www.ru.is/faculty/luca/PAPERS/algobisimchapter.pdf.

[GV15] Roberto Gorrieri and Cristian Versari. *Introduction to Concurrency Theory - Transition Systems and CCS*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2015. ISBN: 978-3-319-21490-0. DOI: 10.1007/978-3-319-21491-7. URL: https://doi.org/10.1007/978-3-319-21491-7.

[OSV16] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, Third Edition. A comprehensive step-by-step guide*. artima, 2016. URL: https://www.artima.com/shop/programming_in_scala_3ed.

[Gor17] Roberto Gorrieri. *Process Algebras for Petri Nets - The Alphabetization of Distributed Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2017. ISBN: 978-3-319-55558-4. DOI: 10.1007/978-3-319-55559-1. URL: https://doi.org/10.1007/978-3-319-55559-1.

[Lib19] Alessandro Liberato. "A Study on Bisimulation Equivalence and Team Equivalence". MA thesis. University of Bologna, Computer Science [LM-DM270], 2019. URL: http://amslaurea.unibo.it/19128/.

[Bar21] Andrea Baroni. *Internship repository*. 2021. URL: https://github.com/AndreaBaroni92/Tirocinio.

[Gor21a] Roberto Gorrieri. "Causal Semantics for BPP Nets with Silent Moves". In: *Fundam. Informaticae* 180.3 (2021), pp. 179–249. DOI: 10.3233/FI-2021-2039. URL: https://doi.org/10.3233/FI-2021-2039.

[Gor21b] Roberto Gorrieri. "Team bisimilarity, and its associated modal logic, for BPP nets". In: *Acta Informatica* 58.5 (2021), pp. 529–569. DOI: 10.1007/s00236-020-00377-4. URL: https://doi.org/10.1007/s00236-020-00377-4.