Aalborg University

Caal 2.0

Recursive HML, Distinguishing Formulae, Equivalence Collapses and Parallel Fixed-Point Computations

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Software Engineering
Selma Lagerløfs Vej 300
Telephone +45 9940 9940
+45 9940 9798
http://www.cs.aau.dk

Synopsis:

This report documents the continued development of Caal, a tool to aid in the education of semantics and verification.

We extend HML to fully support multiple variables, with the limitation that they cannot be mutual recursive. We describe the complete reduction of HML to dependency graphs, implement games for HML formulae, describe the rules and prove the relation between game winner, satisfiability and fixed-point assignments to dependency graphs.

As an alternative to playing games, we implement the ability to generate distinguishing formulae for two non-bisimilar processes. We show that finding a distinguishing formulae is a simple task, however finding the smallest one is not, but offer a greedy algorithm.

Processes can sometimes become too complex and unwieldy. We describe equivalence collapsing, and how it can help collapse bisimilar process to simplify the visualisation of processes.

Finally we tried to adapt Caal to run in parallel to see if we could speed up the computation of fixed-points by running on multiple processes in parallel.

Title:
Caal

Subject:
Semantics and Verification

Project period:
Spring semester 2015

Project group:
DES103F15

Attendees:
Jacob K. Wortmann
Simon Reedtz Olesen
Søren Enevoldsen

Supervisors:
Jiří Srba
Kim Guldstrand Larsen

Finished: 09-06-2015

Number of pages: 108

Appendix pages: 0

Signatures

Jacob K. Wortmann

Simon Reedtz Olesen

Søren Enevoldsen

Preface

This report is a Master Thesis written by Jacob K. Wortmann, Simon Reedtz Olesen and Søren Enevoldsen, a group of Software Engineering students from the Department of Computer Science at Aalborg University, during the Spring of 2015. The report documents further developments to Caal.

We would like to thank our supervisors Jiří Srba and Kim Guldstrand Larsen for their assistance and feedback given throughout the project.

Abstract

This report documents the addition of new features to Caal. Previous work on Caal resulted in a web application used for teaching students about reactive systems. The tool supports description of processes in the process algebra Calculus of Communicating Systems. It is possible to verify the presence or absence of various preorders and equivalence between processes, and the tool offers to play against the user in bisimulation games to prove to the user the validity of a result. The tool also supports model checking of recursive Hennessy-Milner formulae. All computations are done by reducing problems to dependency graphs, and interpreting the computed fixed-point assignments, which are computed by an 'on-the-fly' algorithm.

We extend Caal with full syntax and semantics of HML formulae and offer reductions from HML formulae to dependency graphs. To complement the bisimulation games, we define the HML game and show that satisfiability is related to which player has a universal winning strategy. We also show that universal winning strategy corresponds to fixed-point assignments on the dependency graphs, and in doing so also shows the relationship between satisfiability and fixed-point assignments.

As an alternative to playing bisimulation games, we show it is possible to compute a distinguishing formula for two non-bisimilar processes. Finding the simplest possible distinguishing formula turns out difficult and we conjecture the decidability problem of whether a simple formula exists is NP-hard. Instead we present a greedy algorithm simplifies some formulae.

In Caal it is possible to view a visual representation of processes. Processes can become quite complex and difficult to understand. Equivalence collapses may simplify the state system and allow the user to focus on particular aspects of a process. We present the theory and an algorithm to collapse processes based on merging equivalent processes into equivalence classes.

As an experiment we attempted to reuse the code in Caal for a parallel algorithm. We document our design, pseudocode, and test result. The results were unsatisfactorily, but we identify possible reasons for this.

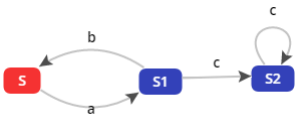
Contents

Introduction 1

This report documents the continuation of our work on Caal, first documented in [14]. Caal is a tool intended for education in semantics and verification of processes described in the process algebra Calculus of Communicating Systems (CCS) [13]. At Aalborg University students can choose to follow a semantics and verification course as part of their education. The course teaches students the theory of reactive systems and how to model, analyse, and understand them. Caal has a graphical interface with an editor that supports the syntax of CCS, an explorer which enables the user to interact with CCS processes. Figure 1.1a and Figure 1.1b shows the definition of an CCS process in the editor and the process visualised by the explorer in Caal.

```
1  S = a.S1;
2  S1 = b.S + S2;
3  S2 = c.S2;
```

(a) CCS editor.



(b) CCS Labelled Transition System.

Figure 1.1

Apart from supporting visualisation of CCS processes, Caal also supports checking for various preorders and equivalences, including strong and weak bisimilarity, and offers model checking for formulae described using recursive Hennessy-Milner logic (HML) [8; 11]. It includes equivalence games where the user is able to play against the computer in a way to help convince them of the validity of a equivalence result. To describe the theory behind, and the implementation of these features, we make use of Dependency Graph (DG)s [12]. For computing the fixed-points of the DG we use a local "on-the-fly" algorithm.

Caal also supports generation of distinguishing formulae, as an alternative to playing games. Distinguishing formulae are another way of convincing the user that two processes are not bisimilar. For two non-bisimilar processes, a distinguishing formula has the property that one process satisfies the formula and the other does not. We show that finding *a* distinguishing formula for two non-bisimilar processes is rather simple, but the problem of finding the simplest one is not.

The original report did not fully describe the semantics of recursive HML in Caal. We amend this by offering the full semantics for recursive HML and by describing the complete reduction from HML to dependency graphs for verification purposes. Since games are a useful approach to gain deeper understanding of the transition systems, we implement games for HML formulae, describe the rules, and prove the relation between game winner and satisfiability.

For complex transition systems the graphical overview of a process and its transitions can quickly become unwieldy and confusing. In some cases

the complexity of the graph can be reduced by collapsing processes that are bisimilar. If a process can take a transition to two bisimilar processes, we can simplify the visualisation of the two bisimilar processes, by collapsing them into one process, without changing the behaviour.

Caal is meant to be an educational tool running on the omnipresent browser platform. Its primary design goal was not designed to fully utilise the computer's processing power. Nevertheless we investigate whether Caal can be adapted to run in parallel on multiple processes. This would make the tool usable for larger state systems that are not practical to verify in the browser. We attempt to adapt the software and try to compute fixed-points, the central framework for all computations in Caal, using multiple processes running in parallel.

1.1 Related Work

This sections talks about some of the other model checking tools available and how they are similar or different from Caal.

The Edinburgh Concurrency Workbench (CWB) [18] is a tool for analysing reactive systems, however there is no longer in active development. The latest binaries are from 1999 and are not supported on Mac OS. The users interact with CWB using a command-line interface. CWB has been the tool of choice at Aalborg University through a number of years. However with the lack of support it has become difficult to acquire and install.

CWB supports μ-calculus for model checking. As of version 7.1 they have implemented support for games, however only strong bisimulation game is available.

Concurrency Workbench of the New Century (CWB-NC) [4] is a tool developed at Stony Brook University. Unlike CWB they claim to support both a command-line interface and a graphical interface however it has been impossible to test as the last update is from year 2000 and the download links do not work.

CWB-NC supports μ-calculus for model checking and their system design allows users to add new equivalences and preorders themselves. There has no been no systematic studying of their performance.

Both CWB and CWB-NC are command-line based tools with little or none graphical interface. This also includes no editor. The users must write their CCS programs in an external editor without syntax checking and load the file using the command-line.

Both tools have support for equivalence checking of CCS and model checking using μ-calculus. Neither of them make a claim to be using "on-the-fly" algorithms for the verification process.

The most notable comparison of the two tools and Caal is our graphical interface, abstracting away from the command-line and letting the users visualise their processes. Caal includes an editor with full support for the syntax and semantics of CCS and implements games for both strong and weak bisimulation and model checking game for recursive HML formulae.

1.2 Bibliographical Remarks

The definitions and theorems in Chapter 2 are the same as used in our previous work [14], with the exception of Example 2.8 and Example 2.19.

Both the notion of dependency graphs and the local minimum fixed-point algorithm described in Section 2.2, was introduced by Liu and Smolka in [12]. Our definitions, which differ in the descriptions of pre-fixed-points and post-fixed-points, and the algorithm, which contains a correction [9] to a minor omission in the original, are taken from previous work [14].

Hennessy and Milner introduced HML in [8]. In previous work we extended the syntax from [2] to include multiple recursively defined variables. The syntax presented in Chapter 3 is the same, but in this report the semantics are reworked, and completed for multiple variables.

The definition and rules of both bisimulation and HML games are based on the book Reactive Systems [2], but the reduction to dependency graphs for both bisimulation and HML games are from our previous work [14]. Just like the semantics, the reduction of HML to dependency graphs has been completed to support multiple variables.

The Caal Tutorial in Chapter 8 is written in cooperation with project members from our previous work, and it is partly based on the text from [3]. Our main contributions to this chapter is Section 8.1, the section about model checking in Section 8.4 and the section about HML game in Section 8.5.

Preliminaries

2

CAAL is a multi semester project that continues from previous work. It is necessary to introduce some previous theory in order for this report to be understandable. The first part covers the language CCS, and Labelled Transition System (LTS) along with bisimulation relations. Following this, we introduce dependency graphs and assignments which forms the basis for implementation. The chapter ends by describing algorithms to compute fixed-points.

2.1 Syntax and Semantics for CCS

We have a countable finite collection of channel names $\mathcal{A}$, and the set of complementary names $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$. The label set is defined as $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$. Then the set of actions is defined by

$$\texttt{Act} = \mathcal{L} \cup \{\tau\}.$$

Assume a finite collection of process names or process constants $\mathcal{K}$. The collection of CCS expressions $\mathcal{P}$ is given by the grammar:

$$P, Q ::= K \;\Big|\; \alpha.P \;\Big|\; P + Q \;\Big|\; P \mid Q \;\Big|\; P[f] \;\Big|\; P \setminus L \;\Big|\; 0$$

where K is a process name in $\mathcal{K}$, α is an action in $\texttt{Act}$, L is a set of labels from $\mathcal{A}$, and $f : \texttt{Act} \longrightarrow \texttt{Act}$ is a function satisfying the constraints:

$$f(\tau) = \tau,$$
$$f(\overline{a}) = \overline{f(a)} \text{ for each label } a.$$

By convention we have that $\overline{\tau} = \tau$.

The behaviour of each process name is given by a defining equation. There is one definition for each $K \in \mathcal{K}$:

$$K \stackrel{\text{def}}{=} P,$$

where $P \in \mathcal{P}$ and the constant K may appear in P.

With relabelling it is possible to write $[b_1/a_1, ..., b_n/a_n]$, where $n \geq 1, a_i \in \mathcal{A}, b_i \in \mathcal{A} \cup \{\tau\}$, for each $i \in \{1, ..., n\}$ and a_i are distinct channel names. The relabelling maps each a_i to b_i, and each $\overline{a_i}$ to $\overline{b_i}$.

CCS uses the convention that the operators have decreasing binding strength in the following order

1. Restriction and relabelling (the tightest binding)
2. Action prefixing
3. Parallel composition and summation

The Structural Operational Semantics (SOS) rules are given in Table 2.1 where $\alpha \in \texttt{Act}$ and $a \in \mathcal{L}$.

$$\text{ACT}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM1}\ \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

$$\text{COM1}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{SUM2}\ \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

$$\text{COM2}\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad \text{REL}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{COM3}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \overline{\alpha} \notin L$$

$$\text{CON}\ \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K \stackrel{\text{def}}{=} P$$

Table 2.1: SOS rules for CCS.

Given an CCS process and the SOS rules we can describe the behaviour of a process using a LTS [10]. An LTS consists of a number of states, a set of actions, and relations.

Definition 2.1 — *Labelled Transition System*

An LTS is a triple $(\texttt{Proc}, \texttt{Act}, \rightarrow)$ where `Proc` is a set of states, `Act` is a set of actions, and $\rightarrow \subseteq \texttt{Proc} \times \texttt{Act} \times \texttt{Proc}$ is the transition relation.

Given one or more processes, it might be interesting to see if two states have the same behaviour, more specifically whether or not they can match each other's transitions. For this we introduce the notion of bisimulation. We introduce both strong and weak bisimulation. Two states are said to bisimulate each other if there exists a bisimulation relation between them.

Definition 2.2 — *Strong Bisimulation*

A binary relation $\mathcal{R}$ over the set of states of an LTS is a bisimulation if and only if whenever $s_1 \mathcal{R} s_2$ and α is an action:

If $s_1 \xrightarrow{\alpha} s_1'$ then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $s_1' \mathcal{R} s_2'$ and
If $s_2 \xrightarrow{\alpha} s_2'$ then there is a transition $s_1 \xrightarrow{\alpha} s_1'$ such that $s_1' \mathcal{R} s_2'$.

Two states s and s' are *bisimilar* written, $s \sim s'$, if and only if there is a bisimulation that relates them. From now on the relation $\sim$ will be referred to as *strong bisimilarity.*

All τ-actions in process behaviours are supposed to be internal and thus unobservable. Consider the following example $acc.\tau.0$ and $acc.0$, since the τ-action is supposed to be unobservable we would expect these two processes to be observable equivalent. Unfortunately, the processes $acc.\tau.0$ and $acc.0$ are not strongly bisimilar. Therefore we need a notion of equivalence which preserves all the good properties of strong bisimulation abstracting from τ-actions in the behaviour of a process. In order to define a notion that allows us to abstract from the unobservable τ-actions in process behaviours, we first must define another transition relation between processes.

Definition 2.3

Let P and Q be two states in an LTS. For each action α, we write $P \stackrel{\alpha}{\Rightarrow} Q$ if and only if either

- $\alpha \neq \tau$ and there are processes P' and Q' such that
$$P(\stackrel{\tau}{\rightarrow})^* P' \stackrel{\alpha}{\rightarrow} Q'(\stackrel{\tau}{\rightarrow})^* Q$$
- or $\alpha = \tau$ and $P(\stackrel{\tau}{\rightarrow})^* Q$,

where we write $(\stackrel{\tau}{\rightarrow})^*$ for the reflexive and transitive closure of the relation $\stackrel{\tau}{\rightarrow}$.

With $\Rightarrow$ defined, we are able to define the equivalence notion weak bisimulation.

Definition 2.4 — *Weak Bisimulation*

A binary relation R over the set of states of an LTS is a weak bisimulation if and only if whenever $s_1 R s_2$ and α is an action (including τ):

if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\Rightarrow} s_2'$ such that $s_1' R s_2'$;
if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\Rightarrow} s_1'$ such that $s_1' R s_2'$.

Two states s and s' are weakly bisimilar, written $s \approx s'$, if and only if there is a weak bisimulation that relates them. From now on the relation $\approx$ will be referred to as weak bisimilarity.

CCS processes can consists of different constants, but we want to ensure that every process is guarded within the scope of some action prefix.

Definition 2.5 — *Guarded Process Constant*

A process constant K is *guarded* in CCS expression P, if every occurrence of K in P is within the scope of some action prefixing.

We say that if a process is not guarded, it is unguarded and we can prove that any process that is guarded has finitely many derivation trees.

Lemma 2.6 — *[14]*

Any CCS expression P where every process constant K is guarded in P can only have finitely many derivation trees, all of them finite.

Any prcoess outside the scope of an action prefix is *unguarded*. We provide a more formal definition in terms of a *reference graph*

Definition 2.7 — *Reference Graph*

A reference graph is a directed graph described by a tuple (V, E), where the vertices $V = \mathcal{K}$ are constants and $E \subseteq V \times V$. For any CCS definition $K \stackrel{\text{def}}{=} P$ we have $(K, K') \in E$ if and only if K' occurs in P unguarded (not in the scope of action prefixing).

Example 2.8

Consider the following process definitions.

$$P \stackrel{\text{def}}{=} Q + a.R$$
$$Q \stackrel{\text{def}}{=} R + b.Q + P$$
$$R \stackrel{\text{def}}{=} P + c.R + R$$

Figure 2.1 shows the corresponding reference graph.

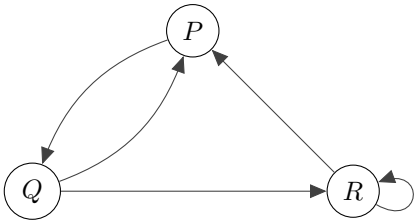


Figure 2.1: Cycle representing unguarded recursion.

Process P does not have an edge to R since R occurs in the scope of action prefixing in $a.R$. The graph describes that to derive the transitions for the process constant P, it is necessary to derive the transitions for the process constant Q, which itself depends on R and P. To derive the transitions for R is it necessary to derive the transitions both for P and R itself. Cycles in the reference graph correspond to unguarded recursion.

Definition 2.9 — *Weakly Guarded CCS Expression*

A CCS program is *weakly guarded* if and only if its reference graph has no cycles.

A program fulfilling this definition of guardedness satisfies the following property:

Lemma 2.10 — *[14]*

All weakly guarded CCS expressions have a finite number of derivations trees all of finite size.

2.2 Dependency Graphs

This section introduce the notion of DG and algorithms for computing fixed-points for dependency graphs [12]. DGs are used as the basis for verifications in CAAL.

In order to compute fixed-points on DGs, we introduce the notion of assignments on DGs and how the complete lattice that exists between assignments ensures the existence of both minimum and maximum fixed-points by applying the Knaster-Tarski Theorem [19]. For the computation of the fixed-points we present a local "on-the-fly" algorithm.

Definition 2.11

A DG is a pair (V, E) where V is a finite set of vertices and $E \subseteq V \times \mathcal{P}(V)$ is a finite set of hyperedges. A hyperedge is a pair (v, T) where v is the source and $T \subseteq V$ is the target set.

Example 1. $G = (V, E)$ is a dependency graph where $V = \{u, w, x, y\}, E = \{(y, \emptyset), (x, \{y\}), (u, \{x, y\}), (u, \{w\}), (w, \{w\})\}$. This can be seen in Figure 2.2.

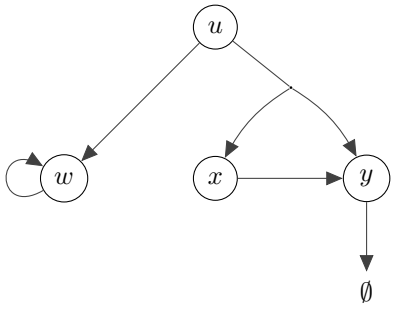


Figure 2.2: Simple dependency graph.

Using dependency graphs we are able to compute minimum and maximum fixed-points. In order to compute the fixed-points it is useful to introduce the notion of *assignments*.

Definition 2.12

An *assignment* A on a dependency graph G is defined as a function mapping each vertex to either 1 or 0: $A : V \to \{0, 1\}$.

There is also the notion of the pre-fixed-point assignments:

Definition 2.13

A *pre-fixed-point assignment* A of dependency graph G is an assignment where for all $v \in V$, if there is an $(v, T) \in E$ such that for all $v' \in T$ it is the case that $A(v') = 1$, then $A(v) = 1$.

And post-fixed-point assignments:

Definition 2.14

A *post-fixed-point assignment* A of dependency graph G is an assignment where for any $v \in V$, if it is the case that for all $(v, T) \in E$ there is a $v' \in T$ such that $A(v') = 0$, then $A(v) = 0$.

The partial order $\sqsubseteq$ between assignments is defined as $A \sqsubseteq A'$ if for all $v \in V$, $A(v) \leq A(v')$ and forms a complete lattice. Then by the Knaster-Tarski fixed-point theorem, there exists a unique minimum pre-fixed-point assignment, A_{min}, and a unique maximum post-fixed-point assignment, A_{max} [19].

We define F as a function mapping assignments to assignments.

Definition 2.15

Let A be an assignment on a dependency graph $G = (V, E)$, then $F(A)$ is also an assignment for G, where $F(A)(v') = 1$ if and only if there exists a hyperedge $(v', T) \in E$ such that it is the case for all $v \in T$ that $A(v) = 1$.

Repeated applications of F starting with $\bot$ (assignment mapping all vertices to 0) approximates the minimum-fixed-point $F(F(\ldots F(\bot)))$. For convenience, we let $F^0(A) = A$, and $F^{n+1}(A) = F(F^n(A))$. When $F^{n+1}(\bot) = F^n(\bot)$, then $F^n(\bot) = A_{min}$.

Example 2.16

As an example we compute the minimum fixed-point for the example in Figure 2.2.

1. Initially all vertices are marked 0. However, for the hyperedge $(y, \emptyset)$, it is trivial that for all $y' \in \emptyset$, it is the case that $A_{min}(y') = 1$. Thus the assignment must have $A_{min}(y) = 1$.

2. Since $A_{min}(y) = 1$, and due to the hyperedge $(x, y) \in E$, it must be that $A_{min}(x) = 1$.

3. $A_{min}(x) = 1$ and $A_{min}(y) = 1$, then due to $(u, \{x, y\}) \in E$, it is the case that $A_{min}(u) = 1$.

4. $A_{min}(w) = 0$ since it is not case that for any $(w, T) \in E$, for all $v' \in T$, $A_{min}(v') = 1$.

Computing Fixed-Point Assignments

We now introduce a local algorithm for computing minimum pre-fixed-point assignments on a DG. The algorithm was first introduced by Xinxin Liu and Scott A. Smolka [12].

Local Algorithm

Algorithm 2.1 shows the local Liu-Smolka algorithm. The algorithm takes a dependency graph $G = (V, E)$ and a vertex $v \in V$ as its input and computes the minimum pre-fixed-point assignment 'on the fly' $A^G_{min}(v)$.

The algorithm computes the minimum pre-fixed-point assignment $A^G_{min}(v)$ for a single vertex which means instead of constructing the whole DG, only the necessary parts have to be constructed during model checking. This can reduce both the time required for model checking but also the memory usage.

The algorithm maintains a set W of hyperedges waiting to be processed, and for each $v \in V$ records in $D(v)$ the hyperedges which were processed under the assumption that $A(v) = 0$.

W initially contains the set of hyperedges which have v_0 as the source, and will be expanded as needed. The symbol $\perp$ indicates that a vertex has not been investigated yet.

A hyperedge $e = (v, T)$ is selected and removed from W in each iteration of the while-loop. There are three cases:

- If $A(v') = 1$ for every vertex v' in the target set T of e, then also $A(v) = 1$. Each hyperedge $e' \in D(v)$ must be re-processed since the assumption that $A(v) = 0$ is no longer true. To accomplish this, $D(v)$ is added to W.

- There exists a $v' \in T$ such that $A(v') = 0$. To allow e to be re-processed if $A(v')$ becomes 1 at a later point, e is added to $D(v')$.

- When $A(v') = \perp$ for every $v' \in T$. In this case e is added to $D(v')$ and every hyperedge which has v' as the source is added to W.

The local algorithm terminates when $W = \emptyset$.

Table 2.2 shows the internal state of the local algorithm before the i'th iteration of the while-loop when executed on the dependency graph shown in Figure 2.2 and the vertex x.

Algorithm 2.1 Liu-Smolka Local Algorithm.

Input: A dependency graph $G = (V, E)$ and a vertex $v_0 \in V$.
Output: The minimum pre-fixed-point assignment $A_{min}(v_0)$ for v_0.

```
1: for all v ∈ V do
2:     A(v) ← ⊥
3: A(v_0) ← 0
4: D(v_0) ← ∅
5: W ← succ(v_0)
6: while W ≠ ∅ do
7:     e ← (v, T) ∈ W
8:     W ← W \ {e}
9:     if ∀v' ∈ T. A(v') = 1 then
10:        A(v) ← 1
11:        W ← W ∪ D(v)
12:    else if ∃v' ∈ T. A(v') = 0 then
13:        D(v') ← D(v') ∪ {e}
14:    else
15:        A(v') ← 0
16:        D(v') ← {e}
17:        W ← W ∪ succ(v')
18: return A(v_0)
```

i	W	$A(x)$	$A(y)$	$D(x)$	$D(y)$
1	$\{(x,\{y\})\}$	0	$\bot$	$\emptyset$	$\emptyset$
2	$\{(y,\emptyset)\}$	0	0	$\emptyset$	$(x,\{y\})$
3	$\{(x,\{y\})\}$	0	1	$\emptyset$	$(x,\{y\})$
4	$\emptyset$	1	1	$\emptyset$	$(x,\{y\})$

Table 2.2: Execution of Algorithm 2.1 on Figure 2.2.

We will now define the level of a vertex $v \in V$ on a dependency graph $G = (V, E)$. This definition will aid us in finding the distinguishing formula later in the report.

Definition 2.17

The level of vertex $v \in V$ is $\text{level}(v) = n$ if $F^n(v) = 1$ and $F^{n-1}(v) = 0$. By convention if the vertex's assignment never changes, then $\text{level}(v) = \infty$.

Strong Bisimilarity Reduction

We present reductions from equivalences to problems of computing the minimum pre-fixed-point assignment on a dependency graph, which can be done using the algorithms presented in Section 2.2.

Suppose we have an LTS $T = (\texttt{Proc}, \texttt{Act}, \rightarrow)$ and two states $s, t \in \texttt{Proc}$ and that we want to determine if s and t are strongly bisimilar. We construct a dependency graph $G^T_\sim = (V, E)$ where $V \subseteq \texttt{Proc} \times \texttt{Proc}$. The dependency graph is constructed with (s, t) as the root and the hyperedges are generated

by Algorithm 2.2. The function $succ(s, \alpha)$ returns a set of states $\{s' \mid s \xrightarrow{\alpha} s'\}$.

Algorithm 2.2 Successor generator for $G^T_\sim = (V, E)$.

Input: An LTS $T = (\texttt{Proc}, \texttt{Act}, \rightarrow)$ and two states $s, t \in \texttt{Proc}$.
Output: All hyperedges starting from (s, t).

1: $E \leftarrow \emptyset$
2: **for all** $\alpha \in \texttt{Act}$ **do**
3: $\quad S \leftarrow \text{succ}(s, \alpha)$
4: $\quad T \leftarrow \text{succ}(t, \alpha)$
5: $\quad$ **for all** $s' \in S$ **do** ▷ Hyperedge constructed triggered by $s \xrightarrow{\alpha} s'$.
6: $\quad\quad E \leftarrow E \cup ((s, t), \{(s', t') \mid t' \in T\})$
7: $\quad$ **for all** $t' \in T$ **do** ▷ Hyperedge construction triggered by $t \xrightarrow{\alpha} t'$.
8: $\quad\quad E \leftarrow E \cup ((s, t), \{(s', t') \mid s' \in S\})$
9: **return** E

We say the hyperedge added to E in line 6, was *triggered* by $s \xrightarrow{\alpha} s'$. Similarly for line 8, the construction of the hyperedge added to E was *triggered* by $t \xrightarrow{\alpha} t'$.

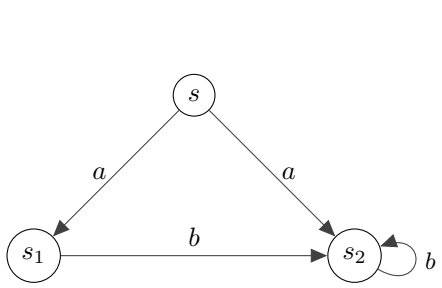
To construct a dependency graph for weak bisimilarity reduction use the weak transition relation instead of the strong. The constructed dependency graph has an interesting property: any vertex in the dependency graph, a process pair (s, t), are bisimilar if and only if the minimum fixed-point for the vertex is 0.

Theorem 2.18 — *[14]*

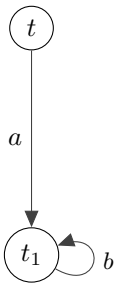
We have that $A_{min}((s, t)) = 0$ if and only if $s \sim t$.

Example 2.19

Let LTS *Left* be defined by the Figure 2.3a and LTS *Right* be defined by Figure 2.3b.



(a) LTS *Left*



(b) LTS *Right*

To find out whether the processes s, t are bisimilar, we construct a bisimulation DG $G = (V, E)$, and compute its the minimum pre-fixed-

point assignment, using the algorithm defined in Algorithm 2.1 and the $succ(s, \alpha)$ defined in Algorithm 2.2.

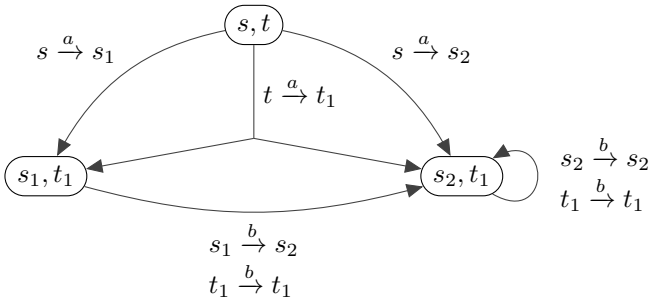


Figure 2.4: Bisimulation dependency graph for process s and t.

The vertices in Figure 2.4 represents process pairs, and are marked either 0 or 1 according to the minimum pre-fixed-point A_{min}. For a single bordered vertex $v = (s, t) \in V$, it is the case that $A_{min}(v) = 0$, and for a double bordered vertex, it is the case that $A_{min}(v) = 1$, note that there are no double bordered vertices in this example. We have established earlier that if $A_{min}(v) = 0$ then $s \sim t$, and if $A_{min}(v) = 1$ then $s \not\sim t$. The edges are labelled with the transitions that triggered the construction of the hyperedge. The hyperedge originating in (s, t), with both (s_1, t_1) and (s_2, t_1) as target vertices, was triggered because $t \xrightarrow{a} t_1$ and s could match with both $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$. In this example the root vertex (s, t) has the assignment $A_{min}((s, t)) = 0$, because it was not possible to promote any vertices $v \in V$ to 1. For a vertex $v = (s, t) \in V$ to be promoted to the assigment $A_{min}(v) = 1$, then there must exists either a transition

$$s \xrightarrow{\alpha} s' \text{ such that } t \not\xrightarrow{\alpha}$$

or

$$t \xrightarrow{\alpha} t' \text{ such that } s \not\xrightarrow{\alpha}$$

Recursive Hennessy-Milner Formulae

3

Sometimes it can be useful to be able to express formulae that hold indefinetely or at some unknown time in the future. This chapter will therefore present HML which was introduced to process theory by Hennessy and Milner in in 1985 [8]. This was later expanded with one variable by [11], allowing for recursion. We present an extended syntax and semantics for recursive HML with multiple variables. We present reductions from formulae to dependency graphs and introduce HML games.

3.1 Syntax

HML consists of boolean connectives, such as conjunctions and disjunctions, modalities, such as exists and for all, and the booleans true and false. The recursive variant we use also includes variables that are defined either as minimum fixed-points or maximum fixed-points. The choice of minimum and maximum fixed-point affects the evaluation of formulae with respect to processes.

Definition 3.1

The set $\mathcal{M}$ of Hennessy-Milner formulae, over a set of actions `Act` and a set of variables `Var` denoted by capital letters, is given by the following abstract syntax:

$$\varphi ::= tt \mid ff \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \langle\langle\alpha\rangle\rangle\varphi \mid [[\alpha]]\varphi \mid X,$$

where $\alpha \in \texttt{Act}$, $X \in \texttt{Var}$, and we use tt and ff to denote 'true' and 'false', respectively. If $A = \{\alpha_1, \ \ldots \ , \alpha_n\} \subseteq \texttt{Act} (n \geq 0)$, we use abbreviation

$\langle A\rangle\varphi$ for the formula $\langle\alpha_1\rangle\varphi \vee \ \ldots \vee \langle\alpha_n\rangle\varphi$,

$\langle\langle A\rangle\rangle\varphi$ for the formula $\langle\langle\alpha_1\rangle\rangle\varphi \vee \ \ldots \vee \langle\langle\alpha_n\rangle\rangle\varphi$,

$[A]\varphi$ for the formula $[\alpha_1]\varphi \wedge \ \ldots \wedge [\alpha_n]\varphi$, and

$[[A]]\varphi$ for the formula $[[\alpha_1]]\varphi \wedge \ \ldots \wedge [[\alpha_n]]\varphi$.

(If $A = \emptyset$, then $\langle A\rangle\varphi = ff$, $\langle\langle A\rangle\rangle\varphi = ff$, $[A]\varphi = tt$, and $[[A]]\varphi = tt$)

Each variable X has to be declared as a either minimum or a maximum fixed-point:

$$X \stackrel{\max}{=} \varphi$$
$$X \stackrel{\min}{=} \varphi$$

where $\varphi \in \mathcal{M}$.

Reference Cycle Length Restriction

The syntax does not prevent defining fixed-point recursively. Nested recursively defined fixed-points can be hard to understand and become expensive to compute. To restrict the costly evaluation we impose a restriction on the fixed-points: a fixed-point declaration may only be mutually recursive with itself. This means that a fixed-point definition for a variable X, may reference variable X, or any other variable Y, as long as Y does not directly, or indirectly, reference X. This restriction will permit us to evaluate fixed-point definitions by substituting referenced variables by their computed fixed-point value.

$$
\begin{aligned}
X &\overset{\max}{=} \langle\alpha\rangle Y \vee \langle\beta\rangle X \\
Y &\overset{\min}{=} \langle\delta\rangle tt \wedge Y \\
P &\overset{\max}{=} \langle\alpha\rangle Q \\
Q &\overset{\min}{=} \langle\delta\rangle P \wedge Q
\end{aligned}
$$

In the above example both X and Y obey the restriction. The definition for X references only itself, or Y which does not reference X in any way. The fixed-point result for Y can be computed independently of X. Afterwards when evaluating the fixed-point for X, the result for Y is known.

Definition P and Q violate this restriction. Variable P references Q which in turn references P. This means neither fixed-point can be found independently of the other. Variable referencing is formally captured by a fixed-point reference graph.

Definition 3.2 — *Fixed-point Reference Graph*

A fixed-point reference graph is a directed graph $G = (V, E)$, where $V = \texttt{Var}$ and $E \subseteq (\texttt{Var} \times \texttt{Var})$. The edge $(X, Y) \in E$ is present if and only if the definition for X references Y directly.

The fixed-point reference graph for the previous example is shown in Figure 3.1.

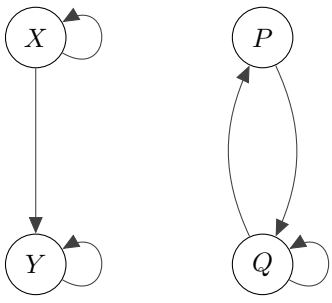


Figure 3.1: Reference graph

Definition 3.3 — *Acyclic Fixed-Points Variables*

Any variable which is not part of any cycle of length greater than one is said to *acyclic*.

All cycles of length greater than one in any reference graph for some fixed-point definitions, violates the restriction and are not acyclic.

Cycles of length one are self-loops, which are equivalent to variables directly referencing themselves, which are permitted. In Figure 3.1 both X and Y are acyclic variables, since the longest cycles they are in are of length 1. Both P and Q violate the restriction since they form a cycle and are not acyclic. If there is only one variable it is always acyclic since it cannot form a cycle of length greater than 1.

We are interested in using HML to describe properties of CCS processes. Informally processes satisfy a formula according to the following rules.

- All processes satisfy tt.
- No processes satisfy ff.
- A process satisfies $\varphi_1 \wedge \varphi_2$ (respectively, $\varphi_1 \vee \varphi_2$) if and only if it satisfies both φ_1 and φ_2 (respectively), either φ_1 or φ_2.
- A process satisfies $\langle\alpha\rangle\varphi$ for some $\alpha \in \texttt{Act}$ if and only if it affords an α-transition leading to a state satisfying φ.
- A process satisfies $[\alpha]\varphi$ for some $\alpha \in \texttt{Act}$ if and only if all of its α-transitions lead to a state satisfying φ.
- A process satisfies $\langle\langle\alpha\rangle\rangle\varphi$ for some $\alpha \in \texttt{Act}$ if and only if it affords a $\stackrel{\alpha}{\Rightarrow}$ transition leading to a state satisfying φ.
- A process satisfies $[[\alpha]]\varphi$ for some $\alpha \in \texttt{Act}$ if and only if all of its $\stackrel{\alpha}{\Rightarrow}$ transitions lead to a state satisfying φ.
- A process satisfies X depending on the value of the minimum or maximum fixed-point value of its definition which we have not defined yet.

3.2 Semantics

The above definitions capture the syntax of recursive HML and give an intuition for the meaning. But they lack in case of recursive variables important details, like what does it mean for a state to satisfy formula φ? This section will provide a formal definition for the semantics.

The function $\mathcal{O}^C_\varphi(S) : 2^{\texttt{Proc}} \to 2^{\texttt{Proc}}$, for a formula φ, in the context C, assuming the set of processes S that are satisfied by C, gives the set of processes then satisfying φ. A context $C \in \texttt{Var} \cup \{\epsilon\}$ is either a variable or the unique context ϵ which is used outside fixed-points.

The function $\mathcal{O}^C_\varphi(S)$ is defined in terms of its context and by induction on the structure of the formula.

$$
\begin{aligned}
\mathcal{O}^X_X(S) &= S \\
\mathcal{O}^C_{tt}(S) &= \texttt{Proc} \\
\mathcal{O}^C_{ff}(S) &= \emptyset \\
\mathcal{O}^C_X(S) &= \begin{cases} \mathit{minfix}\ \mathcal{O}^X_\varphi \text{ if } X \text{ is declared} : X \overset{\min}{=} \varphi \\ \mathit{maxfix}\ \mathcal{O}^X_\varphi \text{ if } X \text{ is declared} : X \overset{\max}{=} \varphi \end{cases} \quad \text{where } X \neq C \\
\mathcal{O}^C_{\varphi_1 \wedge \varphi_2}(S) &= \mathcal{O}^C_{\varphi_1}(S) \cap \mathcal{O}^C_{\varphi_2}(S) \\
\mathcal{O}^C_{\varphi_1 \vee \varphi_2}(S) &= \mathcal{O}^C_{\varphi_1}(S) \cup \mathcal{O}^C_{\varphi_2}(S) \\
\mathcal{O}^C_{\langle\alpha\rangle\varphi}(S) &= \langle\cdot\alpha\cdot\rangle\mathcal{O}^C_\varphi(S) \\
\mathcal{O}^C_{[\alpha]\varphi}(S) &= [\cdot\alpha\cdot]\mathcal{O}^C_\varphi(S) \\
\mathcal{O}^C_{\langle\langle\alpha\rangle\rangle\varphi}(S) &= \langle\langle\cdot\alpha\cdot\rangle\rangle\mathcal{O}^C_\varphi(S) \\
\mathcal{O}^C_{[[\alpha]]\varphi}(S) &= [[\cdot\alpha\cdot]]\mathcal{O}^C_\varphi(S)
\end{aligned}
$$

The four bottommost functions are defined on sets of processes, which are either all the processes that can perform any α-action to a process in the input set, or the processes that can perform an α-action to all processes in the inputs set respectively, defined for both the strict and weak transition relation.

$$
\begin{aligned}
\langle\cdot\alpha\cdot\rangle S &= \{p \in \texttt{Proc} \mid \text{exists } p' \in S \text{ such that } p \xrightarrow{\alpha} p'\} \\
[\cdot\alpha\cdot] S &= \{p \in \texttt{Proc} \mid \text{for all } p' \in S \text{ it is the case } p \xrightarrow{\alpha} p'\} \\
\langle\langle\cdot\alpha\cdot\rangle\rangle S &= \{p \in \texttt{Proc} \mid \text{exists } p' \in S \text{ such that } p \overset{\hat{\alpha}}{\Rightarrow} p'\} \\
[[\cdot\alpha\cdot]] S &= \{p \in \texttt{Proc} \mid \text{for all } p' \in S \text{ it is the case } p \overset{\hat{\alpha}}{\Rightarrow} p'\}
\end{aligned}
$$

The first case, $\mathcal{O}^X_X(S)$, is clear: under the assumption that the set of processes S are satisfied by X, the processes satisfied by X are S. All processes satisfies *tt* and no process satisfies *ff*. The set of processes satisfying a variable that is not the current context is the set of processes in the minimum or maximum fixed-point depending on how the variable is declared, evaluated in the context of the variable. The processes satisfying a conjunction or disjunction is simply the union or intersection respectively of the processes satisfying the constiuent terms.

Since the tuple $(2^{\texttt{Proc}}, \subseteq)$ is a complete lattice and the function $\mathcal{O}^C_\varphi$ is monotonic, there exists a minimum and maximum fixed-point by Tarski [19]:

$$
\begin{aligned}
\mathit{minfix}\ \mathcal{O}^C_\varphi &= \bigcap \{S \in 2^{\texttt{Proc}} \mid \mathcal{O}^C_\varphi(S) \subseteq S\} \\
\mathit{maxfix}\ \mathcal{O}^C_\varphi &= \bigcup \{S \in 2^{\texttt{Proc}} \mid S \subseteq \mathcal{O}^C_\varphi(S)\}
\end{aligned}
$$

Using these definition we can define when a state satisfies a formula.

Definition 3.4

A process $s \in \texttt{Proc}$ is said to satisfy a formula φ, denoted $s \models \varphi$, if and only if $s \in \mathcal{O}^{\epsilon}_{\varphi}(S)$ for any $S \subseteq \texttt{Proc}$.

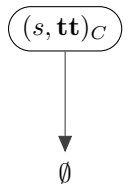
A process s not satisfying a formula φ is denoted by $s \not\models \varphi$. Note that for the particular function $\mathcal{O}^{\epsilon}_{\varphi}$, where the context is ϵ, it does matter what the argument of $\mathcal{O}^{\epsilon}_{\varphi}$ when the context is ϵ, since it is only used in the context of a variable. For convenience we use $\emptyset$ as the argument when the context is ϵ.

3.3 Reduction to Dependency Graphs

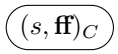
Suppose we have an LTS $T = (\texttt{Proc}, \texttt{Act}, \rightarrow)$, a state $s \in \texttt{Proc}$, and an HML formula φ with only acyclic fixed-point variables, and we wish to determine if $s \models \varphi$. We construct a set of dependency graphs, $\mathcal{DG}$. The vertices of the dependency graphs are triples denoted $(s, \varphi)_C$ where $s \in \texttt{Proc}$, $\varphi \in \mathcal{M}$, and $C \in \texttt{Var} \cup \{\epsilon\}$. The initial dependency graph is constructed with $(s, \varphi)_\epsilon$ as the root.

The reason a set of dependency graphs is constructed is because a single dependency graph cannot encode both a minimum and a maximum fixed-point. Instead each context, including ϵ, will have its own dependency graph containing all vertices under a given context. The fixed-point variables are still indirectly connected: whenever there is a need to know the fixed-point value of vertex $(s, X)_C$, where $C \neq X$, it is necessary to find the fixed-point value of vertex $(s, X)_X$ which is in a different dependency graph. Since all variables are acyclic, so are the indirect dependences.

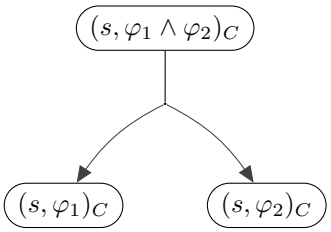
Figure 3.2 shows the rules for how the hyperedges of the dependency graph are constructed depending on the formula is shown. Whenever a vertex is of the form $(s, X)_C$, where $C \neq X$, a new dependency graph is started with $(s, X)_X$ as the root. The dashed arrow indicates an indirect dependence; it is not a hyperedge part of the dependency graph.



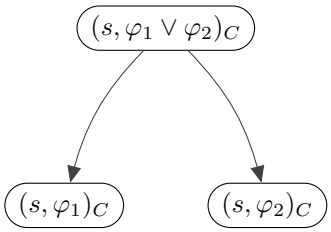
(a) True.



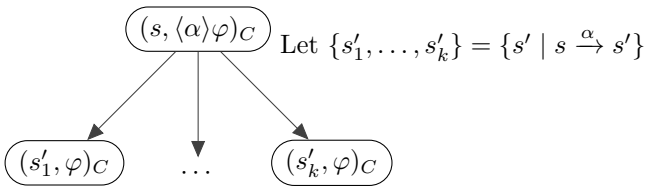
(b) False.



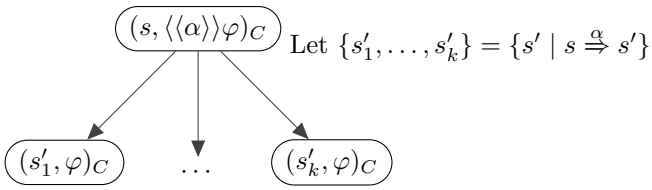
(c) Conjunction.



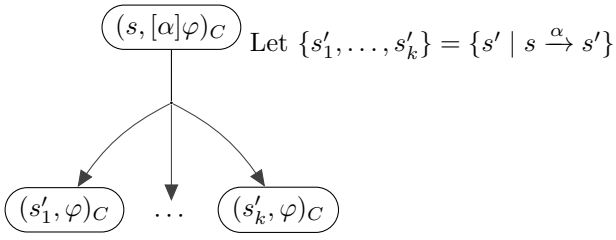
(d) Disjunction.



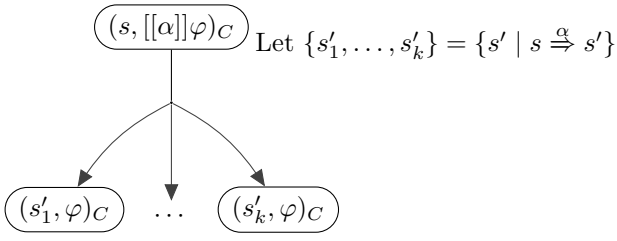
(e) Strong Exists.



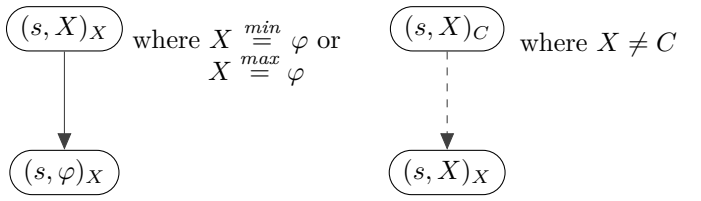
(f) Weak Exists.



(g) Strong For all.



(h) Weak For all.



(i) Encoding of same variable as context

(j) Encoding of indirect dependence on variable different from context

Figure 3.2: Reduction rules for HML variables

Example 3.5

Let $\varphi = [-]ff \vee X$ be the formula and X and Y defined as follows:

$$X \overset{min}{=} Y \wedge \langle - \rangle X$$
$$Y \overset{max}{=} \langle b \rangle tt \vee \langle a \rangle Y,$$

and let Figure 3.3 be the LTS.

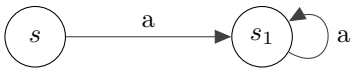


Figure 3.3: LTS to be transformed

The rules from Figure 3.2 are again applied until the dependency graph has been constructed. The full dependency graph can be seen in Figure 3.4

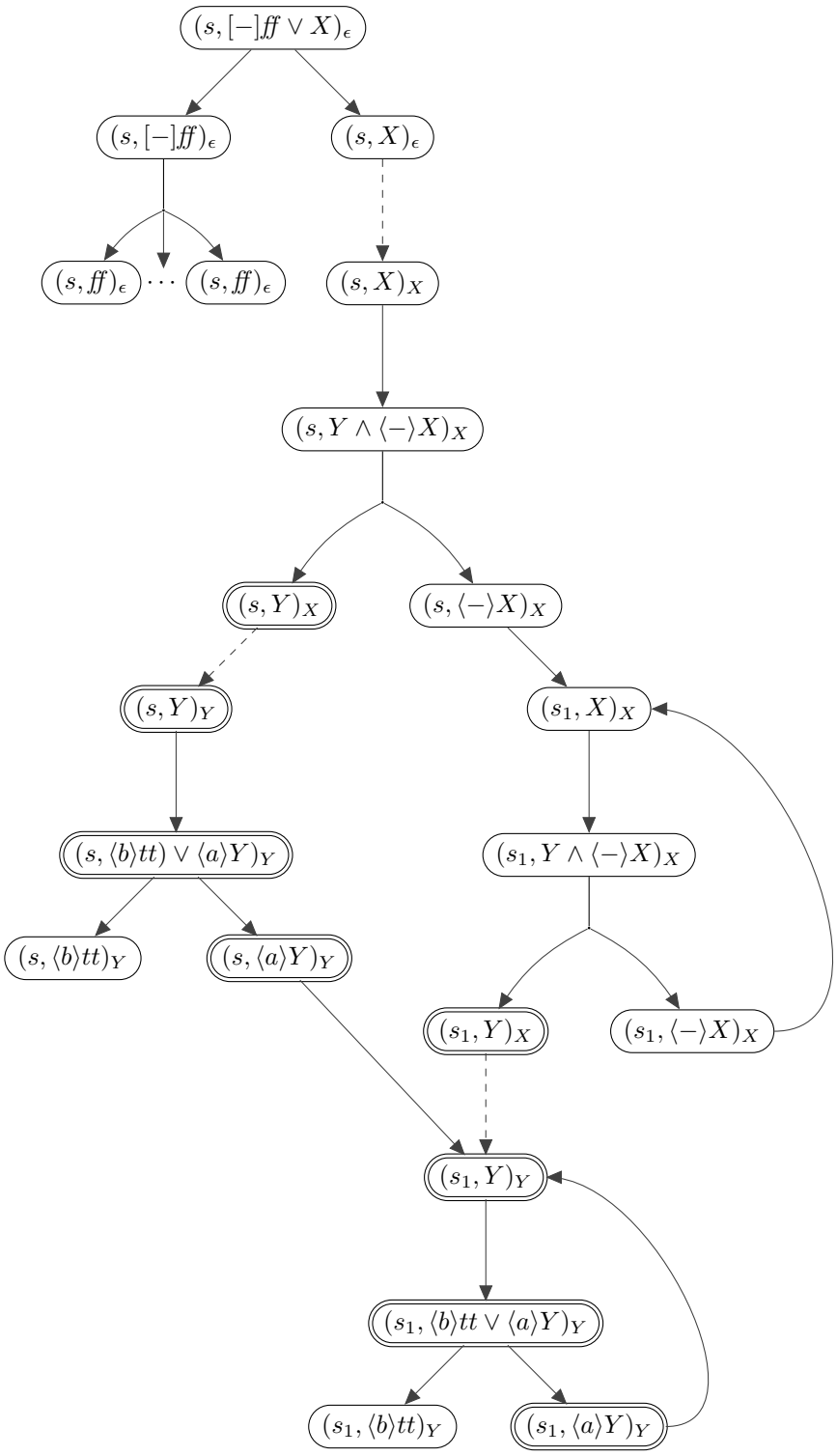


Figure 3.4: Full Dependency Graph for Figure 3.3

In Figure 3.4 vertices that are double bordered have been assigned 1 and vertices that are single bordered have been assigned 0 by a fixed-point algorithm. The vertex $(s_1, Y)_X$ has been assigned 1 because $(s_1, Y)_X$ has been assigned 1. For a similar reason have $(s, X)_\epsilon$ been assigned 0.

Definition 3.6

The fixed-point marking of a vertex in a set of dependency graphs $\mathcal{DG}$ is defined as $A_{\mathcal{DG}}((s, \varphi)_C)$ where:

$$A_{\mathcal{DG}}((s,\varphi)_C) = \begin{cases} A_{min}((s,\varphi)_C) & \text{if } C = \epsilon \text{ or } C \overset{min}{=} \varphi' \\ A_{max}((s,\varphi)_C) & \text{if } C \overset{max}{=} \varphi' \end{cases}$$

To evaluate the fixed-point value for any vertex $(s, \varphi)_C$ in any dependency graph in $\mathcal{DG}$, begin computing the fixed-point as normal. Whenever a vertex $(s, X)_C$, for any variable not in the current context $X \neq C$ is encountered, pause the current fixed-point computation and start either a new minimum or maximum fixed-point computation from $(s, X)_X$ depending on whether X is defined as a minimum or maximum fixed-point respectively. This is guaranteed to terminate since all variable references are acyclic and therefore limits the number of context switches possible. When the fixed-point value for $(s, X)_X$ is known, the same value should be assigned to $(s, X)_C$, such that $A_{\mathcal{DG}}((s, X)_X) = A_{\mathcal{DG}}((s, X)_C)$.

3.4 HML Game

In this section we introduce the game for HML with an arbitrary number of recursive variables. Recall the formula definitions from Definition 3.1. The game is between an "attacker" and a "defender", who have the following goal:

- the attacker has to prove that $s \not\models \varphi$, while
- the defender has to prove that $s \models \varphi$

The configurations of the game has the form $(s, \varphi)_C$ where $s \in \texttt{Proc}$, φ is an HML formula and $C \in \texttt{Var} \cup \{\epsilon\}$ is the current context. For each of the configurations the following successor configurations is defined according to the structure of the formula φ.

- $(s, tt)_C$ and $(s, ff)_C$ have no successor configurations,
- $(s, \varphi_1 \wedge \varphi_2)_C$ and $(s, \varphi_1 \vee \varphi_2)_C$ both have the successor configurations $(s, \varphi_1)_C$ and $(s, \varphi_2)_C$.
- $(s, \langle\alpha\rangle\varphi)_C$ and $(s, [\alpha]\varphi)_C$ have $(s', \varphi)_C$ as successor configuration for all s' such that $s \xrightarrow{\alpha} s'$.

- $(s, \langle\langle\alpha\rangle\rangle\varphi)_C$ and $(s, [[\alpha]]\varphi)_C$ have $(s', \varphi)_C$ as successor configuration for all s' such that $s \stackrel{\alpha}{\Rightarrow} s'$.
- $(s, X)_C$, has a single successor configuration $(s, X)_X$ where $C \neq X$.
- $(s, X)_X$, has a successor configuration $(s, \varphi)_X$, where X is defined as either $X \stackrel{\max}{=} \varphi$ or $X \stackrel{\min}{=} \varphi$.

Definition 3.7 — *Rules for HML Games on DG*

A play of a game starting from $(s, \varphi)_C$ is a maximal sequence of configurations formed by the players according to the following rules. Each round either the attacker or the defender picks a successor configuration if possible.

- The attacker picks a configuration when the formula is of the form $(s, \varphi_1 \wedge \varphi_2)_C$, or when the choices are either $(s, [\alpha]\varphi)_C$, or $(s, [[\alpha]]\varphi)_C$.
- The defender picks a configuration when the formula is of the form $(s, \varphi_1 \vee \varphi_2)_C$, or when the choices are $(s, \langle\alpha\rangle\varphi)_C$, or $(s, \langle\langle\alpha\rangle\rangle\varphi)_C$.

The successor configuration $(s, X)_C$ when $C \neq X$ is uniquely determined and is denoted by $(s, X)_C \to (s, X)_X$. The successor configuration for $(s, X)_X$ is also uniquely defined and is denoted by $(s, X)_X \to (s, \varphi)_X$ where $X \stackrel{\max}{=} \varphi$ or $X \stackrel{\min}{=} \varphi$. The successor configurations chosen by the attacker are denoted $\stackrel{A}{\longrightarrow}$ moves and those chosen by the defender are denoted $\stackrel{D}{\longrightarrow}$ moves.

Notice that every play either

- terminates in $(s, tt)_C$ or $(s, ff)_C$, or
- the attacker or the defender can get stuck in the current configuration, either $(s, [\alpha]\varphi)_C$ or $(s, \langle\alpha\rangle\varphi)_C$ whenever $s \stackrel{\alpha}{\not\to}$, or in $(s, [[\alpha]]\varphi_C)$ or $(s, \langle\langle\alpha\rangle\rangle\varphi_C)$, whenever $s \stackrel{\alpha}{\not\Rightarrow}$.
- the play is infinite.

Definition 3.8 — *Winner in Recursive HML Game*

The winner depends on which configuration the game ends in, or alternatively the context for an infinite play.

- The attacker is a winner in every play ending in a configuration of the form $(s, ff)_C$ or in play in which the defender gets stuck.
- The defender is a winner in every play ending in configuration of the form $(s, tt)_C$ or in a play in which the attacker gets stuck.
- The attacker is a winner in every infinite play in context X, provided that X is a defined as a minimum fixed-point: $X \stackrel{\min}{=} \varphi$. The defender is a winner in every infinite play provided that X is defined as a maximum fixed-point: $X \stackrel{\max}{=} \varphi$.

The intuition for the winner in the context of a fixed-point is as follows: if X is defined as minimum fix-point then the defender has to prove in finitely many rounds that the property is satisfied. If a play of the game is infinite, it means that the defender has failed to prove it in finitely many rounds, which means the attacker wins.

If instead X is defined as maximum fix-point, then the attacker has to prove in finitely many rounds that the property is not satisfied. If the play of the game is infinite, it means that the attacker has failed to prove it in finitely many rounds that the formula is satisfied, which means the defender wins.

With the support for multiple variables, there is a possibility that the context will switch several times throughout a play, each time the context changes between minimum and maximum fix-point, the winning conditions change too. Even with support for multiple variables, an infinite play can only span over one context, as we do not allow variables to be mutually recursive.

Universal Winning Strategy and Satisfiability

In this section we show that a process satisfies a formula if and only if the defender has a universal winning strategy. Otherwise the process does not satisfy the formula and the attacker has a universal winning strategy.

Example 3.9

We show an example of a formula for the process in Figure 3.5.

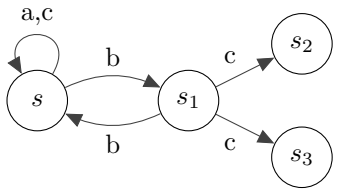


Figure 3.5: Branching example.

Let $\varphi = \langle a \rangle tt \wedge [b]\langle c \rangle tt$ be the formula. To show that $s \models \varphi$, we need to shows that there is a universal winning strategy for the defender starting from $(s, \varphi)_\epsilon$.
For the first move the attacker has two choices:

1. $(s, \langle a \rangle tt \wedge [b]\langle c \rangle tt)_\epsilon \xrightarrow{A} (s, \langle a \rangle tt)_\epsilon$
2. $(s, \langle a \rangle tt \wedge [b]\langle c \rangle tt)_\epsilon \xrightarrow{A} (s, [b]\langle c \rangle tt)_\epsilon$

Choosing case 1 will result in a loss for the attacker, because the defender would respond with

$$(s, \langle a \rangle tt)_\epsilon \xrightarrow{D} (s, tt)_\epsilon$$

and end up in the *true* formula by taking the a-transition, so the attacker is forced to take case 2 and take the b-transition to s_1. The defender now responds with:

$$(s_1, \langle c \rangle tt)_\epsilon \xrightarrow{D} (s_2, tt)_\epsilon$$

and ends up in (s_2, tt) and the defender therefore has a universal winning strategy.

For a given finite-state system, all formulae containing variables have a equivalent formula containing no variables that are satisfied by the same processes. Occurrences of variables in the definition of fixed-point variables are replaced by substituting them for their definition a finite number of times. The number of times when all variables occurs in the scope of a modal operator is the number of states in the finite-state system. The intuition is that for a process to satisfy a minimum fixed-point, it must be possible to show in a finite number of transitions. Similar for process not to satisfy a maximum fixed-point, it must be shown in a finite number of transitions that the process does not satisfy its definition.

The formal notation for this substitution is as follows. For any minimum fixed-point variable $X \overset{min}{=} \psi$ let $X^0 = ff$ and $X^{m+1} = \psi[X^m/X]$. Similar for a maximum fixed-point variable $X \overset{max}{=} \psi$ let $X^0 = tt$ and $X^{m+1} = \psi[X^m/X]$.

The substitution is performed for all variables that do not refer to any other variables in their body. Because we require variables references to be acyclic there is always such a variable. Then the resulting formula can be substituted for the variable in other definitions and so on until there are no more variables left. Then the formula is *variable-free*.

The following theory sums up this folklore result on fixed-point unfolding [2].

Theorem 3.10 — *Reduction of one variable*

Let φ be an acyclic recursive HML formula for which all occurrences of variables in variable definitions occur in the scope of a modal operator. Let T be a finite-state LTS with m states, and pick any variable X in φ. Then $p \models \varphi \Leftrightarrow p \models \varphi[X^m/X]$ for all processes p in T.

The following example shows how the substitution works.

Example 3.11

We use the following process $P \overset{def}{=} a.b.P$ with 2 states, and the following two variable definitions:

$$X \overset{min}{=} \langle a \rangle Y \vee \langle - \rangle X$$
$$Y \overset{max}{=} \langle b \rangle Y$$

and let $\varphi = X \vee \langle c \rangle Y$.

Y refers only to itself, so we substitute Y first and get $Y^2 = \langle b \rangle \langle b \rangle tt$. This is then used to replace occurrences of Y in other definitions. The new definition for X becomes $X \overset{min}{=} \langle a \rangle \langle b \rangle \langle b \rangle tt \vee \langle - \rangle X$, and the new definition for φ becomes $\varphi = X \vee \langle c \rangle \langle b \rangle \langle b \rangle tt$.

We do the same to X and get $X^2 = \langle a \rangle \langle b \rangle \langle b \rangle tt \vee (\langle - \rangle \langle a \rangle \langle b \rangle \langle b \rangle tt \vee \langle - \rangle ff)$. X is then replaced with this new definition in φ which becomes the variable-free formula $\varphi = \langle a \rangle \langle b \rangle \langle b \rangle tt \vee (\langle - \rangle \langle a \rangle \langle b \rangle \langle b \rangle tt \vee \langle - \rangle ff) \vee \langle c \rangle \langle b \rangle \langle b \rangle tt$.

Theorem 3.12

Let $G = (V, E)$ be a dependency graph constructed using the technique in Section 3.3 from a finite-state CCS process and an recursive HML formula φ with acyclic fixed-point variables. Then the following statements hold.

- State s satisfies φ if and only if defender has a universal winning strategy starting from $(s, \varphi)_\epsilon$.
- State s does not satisfy φ if and only if the attacker has a universal winning strategy starting from $(s, \varphi)_\epsilon$.

Proof. First we prove that if state satisfies the formula then the defender has a universal winning strategy. The proof is by structural induction on φ. Note that since the formula can be made variable-free by Theorem 3.10 it is not necessary to have a proof for variables.

- If $s \models tt$ the process s trivially satisfies the formula. From the configuration $(s, tt)_C$, the game has terminated with the defender as the winner.
- The case $s \models ff$, is a contradiction since no process satisfies ff, and the defender does not need, nor have, a universal winning strategy. It can only be the case that $s \not\models ff$, and in the configuration $(s, ff)_C$ the game has terminated with the attacker as the winner.
- Suppose $s \models \varphi_1 \wedge \varphi_2$. The corresponding configuration is $(s, \varphi_1 \wedge \varphi_2)_C$ with the two successor configurations $(s, \varphi_1)_C$ and $(s, \varphi_2)_C$. Since $s \models \varphi_1$ and $s \models \varphi_2$ it does not matter which successor configuration the attacker chooses since by induction the defender has a universal winning strategy.
- Suppose $s \models \varphi_1 \vee \varphi_2$, then either $s \models \varphi_1$ or $s \models \varphi_2$. The corresponding successor configurations are $(s, \varphi_1)_C$ and $(s, \varphi_2)_C$. The defender's play at this point is to simply select the successor configuration corresponding to any satisfied term, for which the defender by induction also has a universal winning strategy for.
- If $s \models \langle \alpha \rangle \varphi$, there must be a process s' such that $s' \models \varphi$ and $s \xrightarrow{\alpha} s'$. The defender then selects the successor configuration $(s', \varphi)_C$, for which the defender, by induction, has a universal winning strategy for.

The defender plays in a similar manner for $s \models \langle\langle\alpha\rangle\rangle\varphi$ using the weak transition relation $s \stackrel{\alpha}{\Rightarrow} s'$.

- If $s \models [\alpha]\varphi$, then for all processes s' where $s \xrightarrow{\alpha} s'$, it is the case that $s' \models \varphi$. Then it does not matter which successor configuration $(s', \varphi)_C$ the attacker chooses, by the induction hypothesis the defender has a universal winning strategy for all of them. Similar holds for $s \models [[\alpha]]\varphi$ using the weak transition relation $s \stackrel{\alpha}{\Rightarrow} s'$.

For the other direction we show that if state s does not satisfy φ then the attacker must have a universal winning strategy.

- $s \not\models tt$, is a contradiction since the process s trivially satisfies the formula. From the configuration $(s, tt)_C$, the game has terminated with the defender as the winner.

- The case $s \not\models ff$ is trivially provable since no process satisfies ff. In the configuration $(s, ff)_C$, the game has terminated with the attacker as the winner.

- Suppose $s \not\models \varphi_1 \wedge \varphi_2$. The corresponding configuration is $(s, \varphi_1 \wedge \varphi_2)_C$ with the two successor configurations $(s, \varphi_1)_C$ and $(s, \varphi_2)_C$. It is either the case that $s \not\models \varphi_1$ or $s \not\models \varphi_2$ or perhaps s does not satisfy both. Suppose $s \not\models \varphi'$ where $\varphi' = \varphi_1$ or $\varphi'\varphi_2$. The attacker selects the successor configuration $(s, \varphi')_C$, for which the attacker has a universal winning strategy from by the induction hypothesis.

- Suppose $s \not\models \varphi_1 \vee \varphi_2$. That means $s \not\models \varphi_1$ and $s \not\models \varphi_2$. The corresponding successor configurations are $(s, \varphi_1)_C$ and $(s, \varphi_2)_C$. It does not matter which configuration the defender picks since, by the induction hypothesis, the attacker has a universal winning strategy for both of them.

- If $s \not\models \langle\alpha\rangle\varphi$, it is the case that $s' \not\models p'$ for all p' where $s \xrightarrow{\alpha} p'$. No matter which successor configuration $(s', \varphi)_C$ the defender picks, by the induction hypothesis, the attacker has a universal winning strategy. This reasoning is also sound for $s \not\models \langle\langle\alpha\rangle\rangle\varphi$ using the weak transition relation $s \stackrel{\alpha}{\Rightarrow} s'$.

- If $s \not\models [\alpha]\varphi$, then there exist some process s', where $s \xrightarrow{\alpha} s'$, where $s' \not\models \varphi$. The attacker then selects the successor configuration $(s', \varphi)_C$, for which the attacker also has a universal winning strategy due to the induction hypothesis. Similar holds for $s \not\models [[\alpha]]\varphi$ using the weak transition relation $s \stackrel{\alpha}{\Rightarrow} s'$.

□

Example 3.13

We show an example of a formula with one variable for the process in Figure 3.6.

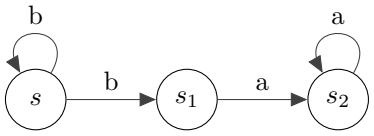


Figure 3.6: Cycle example.

Let $X \overset{min}{=} \langle a \rangle tt \vee (\langle b \rangle tt \wedge [b]X)$ be the HML formula. To show that $s \not\models X$, we must show that the attacker has a universal winning strategy starting from $(s, X)_\epsilon$. The first move is to expand X:

$$(s, X)_\epsilon \rightarrow (s, \langle a \rangle tt \vee (\langle b \rangle tt \wedge [b]X))_X$$

Now the defender has two choices:

1. $(s, \langle a \rangle tt \vee (\langle b \rangle tt \wedge [b]X))_X \overset{D}{\rightarrow} (s, \langle a \rangle tt)_X$
2. $(s, \langle a \rangle tt \vee (\langle b \rangle tt \wedge [b]X))_X \overset{D}{\rightarrow} (s, \langle b \rangle tt \wedge [b]X)_X$

Choosing the case 1 will result in a loss for the defender, because he is supposed to pick an a-successor for state s but $s \overset{a}{\not\rightarrow}$, so he is are forced to choose case 2. The attacker than responds as follows

$$(s, \langle b \rangle tt \wedge [b]X)_X \overset{A}{\rightarrow} (s, [b]X)_X \overset{A}{\rightarrow} (s, X)_X.$$

Now we unfold X and we are now back in the state where we started, which means a cycle has been detected and the attacker win because the fix-point is defined as a minimum fix-point.

Example 3.14

We show an example of a formula with multiple variables for the process in Figure 3.7.

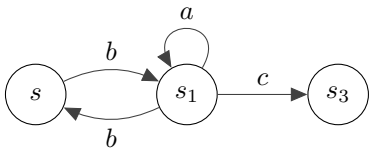


Figure 3.7: Infinite loop example.

Let X or Y be the HML formula and let X and Y be defined as follows

$$X \overset{min}{=} \langle b \rangle X \vee Y$$
$$Y \overset{max}{=} \langle c \rangle tt \wedge \langle a \rangle Y$$

To show that $s \models X$, we must shows that the defender has a universal strategy starting from $(s, X)_\epsilon$. The first move is to expand X:

$$(s, X)_\epsilon \to (s, \langle b \rangle X \vee Y)_X$$

The defender now chooses:

1. $(s, \langle b \rangle X \vee Y)_X \xrightarrow{D} (s, \langle b \rangle X)_X$

The defender now chooses:

1. $(s_1, \langle b \rangle X \vee Y)_X \xrightarrow{D} (s_1, Y)_Y$

This means that the context switches from X to Y and the game is now a maximum fix-point game. The configuration is now $(s_1, Y)_Y$ and Y is expanded. The attacker now has two choices:

1. $(s_1, \langle c \rangle tt \wedge \langle a \rangle Y)_Y \xrightarrow{A} (s_1, \langle c \rangle tt)_Y$
2. $(s_1, \langle c \rangle tt \wedge \langle a \rangle Y)_Y \xrightarrow{A} (s_1, \langle a \rangle Y)_Y$

Choosing case 1 will result in a loss for the attacker as the formula will reach a *true* formula, so the attacker chooses case 2 and the defender responds with:

$$(s_1, \langle a \rangle Y)_Y \xrightarrow{D} (s_1, Y)_Y \to (s_1, \langle c \rangle tt \wedge \langle a \rangle Y)_Y$$

We have already been in this configuration, which means a loop has been detected. According to the definitions this means the defender wins because the game was switched to a maximum fix-point game.

Relation Between Universal Winner and Fixed-Point Assignments

The configurations of the HML game corresponds directly to vertices in a dependency graph. If an attacker has successor configurations, the current configuration has a hyperedge with all the successor configurations. If the defender has successor configurations, the current configuration have multiple hyperedges each to a single configuration.

We show the relation between universal winner in for the game corresponds directly to the fixed-point marking for the set of dependency graphs of the game. The assignment for a vertex in any dependency graph in the set, $A_{\mathcal{DG}}((s, \varphi)_C)$ is defined in Definition 3.6.

Theorem 3.15

Let $\mathcal{DG}$ be the set of dependency graphs constructed by the rules in Figure 3.2. The defender has a universal winning strategy from a configuration $(s, \varphi)_C$ if and only if $A_{\mathcal{DG}}((s, \varphi)_C) = 1$.

Proof. The proof is by induction on the number of indirect references in a path on the set of dependency graphs. The base case is when there is no indirect references and will be proved for a minimum fixed-point context $X \overset{min}{=} \varphi_X$. The proof for a maximum fixed-point context and ϵ are similar.

A level less than infinity implies the minimum fixed-point assignment is 1: $level(V) < \infty \Leftrightarrow A_{\mathcal{DG}}(V) = 1$.

Since the defender may not loop in a minimum fixed-point context, if it has a universal winning strategy it must be in a finite number of steps. We show by induction on the maximum number of steps needed to win, that if the defender has a universal winning strategy from $(s, \varphi)_X$, then $A_{\mathcal{DG}}((s, \varphi)_X) = 1$.

$(s, tt)_X$ The defender has won in 0 steps. From the construction in Figure 3.2a, this vertex has a single hyperedge with no target configurations, implying $A_{\mathcal{DG}}((s, tt)_X) = 1$.

$(s, ff)_X$ The defender loses and we need not prove this case.

$(s, \varphi_1 \vee \varphi_2)_X$ Suppose the defender has a universal winning strategy from $(s, \varphi_1 \vee \varphi_2)_X$ in k steps. We need to show that $A_{\mathcal{DG}}((s, \varphi_1 \vee \varphi_2)_X) = 1$.

From the construction in Figure 3.2d there are two hyperedges going out: one to $(s, \varphi_1)_X$, and one to $(s, \varphi_2)_X$. Since the defender can only win on a finite play it must have at least one universal winning strategy of at most $k - 1$ steps from either $(s, \varphi_1)_X$ or $(s, \varphi_2)_X$ or perhaps both. Suppose it has one from $(s, \varphi_1)_X$, then by the induction hypothesis $A_{\mathcal{DG}}((s, \varphi_1)_X) = 1$. Then by the definition of pre-fixed-point assignment it must be the case that $A_{\mathcal{DG}}((s, \varphi_1 \vee \varphi_2)_X) = 1$.

$(s, \varphi_1 \wedge \varphi_2)_X$ Suppose the defender has a universal winning strategy from $(s, \varphi_1 \wedge \varphi_2)_X$ of k steps. We need to show that $A_{\mathcal{DG}}((s, \varphi_1 \wedge \varphi_2)_X) = 1$.

From the construction in Figure 3.2c there is one hyperedge going out with two target configurations: one to $(s, \varphi_1)_X$, and one to $(s, \varphi_2)_X$. Since the attacker picks, the defender must have a universal winning strategy from both $(s, \varphi_1)_X$ and $(s, \varphi_2)_X$ with the longest being of $k - 1$ steps. Then by the induction hypothesis both successors are assigned 1. Since both target configurations for the hyperedge are assigned 1 it must also be the case that $A_{\mathcal{DG}}((s, \varphi_1 \wedge \varphi_2)_X) = 1$.

$(s, \langle\alpha\rangle\varphi)_X$ Suppose the defender has a universal winning strategy from $(s, \langle\alpha\rangle\varphi)_X$ in k steps. We need to show that $A_{\mathcal{DG}}((s, \langle\alpha\rangle\varphi)_X) = 1$.

From the construction in Figure 3.2e there can be multiple successor configurations $(s_i, \varphi)_X$ where $1 \leq i \leq n$ for some $n \geq 1$. There must be at least one configuration among those for which the defender has a universal winnings strategy in $k - 1$ steps. Suppose that configuration is $(s_j, \varphi)_X$. Then by the induction hypothesis $A_{\mathcal{DG}}((s_j, \varphi)_X) = 1$. Since $(s, \langle\alpha\rangle\varphi)_X$ has a hyperedge with only this configuration in the target set, it must be the case that $A_{\mathcal{DG}}((s, \langle\alpha\rangle\varphi)_X) = 1$.

The proof is similar for weak-exists modality.

$(s, [\alpha]\varphi)_X$ Suppose the defender has a universal winning strategy from $(s, [\alpha]\varphi)_X$ in k steps. We need to show that $A_{\mathcal{DG}}((s, [\alpha]\varphi)_X) = 1$.

From the construction in Figure 3.2g there is a single hyperedge with multiple successor configurations $(s_i, \varphi)_X$ where $1 \leq i \leq n$ for some $n \geq 0$. The defender must have a universal winning strategy for all successor configurations, and each strategy consists of at most $k-1$ steps. Then by the induction hypothesis $A_{\mathcal{DG}}((s_i, \varphi)_X) = 1$ for all $1 \leq i \leq n$. Since all target configurations for the hyperedge are assigned 1 it must also be the case that $A_{\mathcal{DG}}((s, [\alpha]\varphi)_X) = 1$.

If there is no successor configuration at all then the attacker is stuck, but *all* successor configurations are still assigned 1 and therefore $A_{\mathcal{DG}}((s, [\alpha]\varphi)_X) = 1$.

The proof is similar for weak-for-all modality.

$(s, X)_X$ Suppose the defender has a universal winning strategy from $(s, X)_X$ in k steps. We need to show that $A_{\mathcal{DG}}((s, X)_X) = 1$.

Then the defender has a universal winning strategy from $(s, \varphi_X)_X$ in $k-1$ steps implying that $A_{\mathcal{DG}}((s, \varphi_X)_X) = 1$. Then by the construction in Figure 3.2i it must also be the case that $A_{\mathcal{DG}}((s, \varphi_X)_X) = 1$.

We show by induction on the levels of the configurations in the dependency graph that if $A_{\mathcal{DG}}((s, \varphi)_X) = 1$ then the defender has a universal strategy from $(s, \varphi)_X$.

- Assume $A_{\mathcal{DG}}((s, \varphi)_X) = 1$ and $level((s, \varphi)_X) = 1$. From the construction rules in Figure 3.2 this can only happen if φ is of form tt, $[\alpha]\varphi'$ or $[[\alpha]]\varphi'$.

  $(s, tt)_X$ The defender has won for $(s, \varphi)_X$ (and has a universal winning strategy of 0-steps).

  $(s, [\alpha]\varphi')_X$ Since the level is 1 there must be a hyperedge with no target configurations. From the construction in Figure 3.2g this implies the attacker is stuck and the defender has won and therefore have a universal winning strategy of 0-steps.

  $(s, [[\alpha]]\varphi')_X$ This proof is similar to one above for $(s, [\alpha]\varphi')_X$.

- Otherwise the level is some $k > 1$.

  $(s, \varphi_1 \vee \varphi_2)_X$ Suppose $A_{\mathcal{DG}}((s, \varphi_1 \vee \varphi_2)_X) = 1$ and therefore $level((s, \varphi_1 \vee \varphi_2)_X) = k < \infty$. Then we need to show that the defender has a universal winning strategy from this configuration.

  Either $(s, \varphi_1)_X$ or $(s, \varphi_2)_X$ has a level less than k. Suppose $(s, \varphi_1)_X$ does. Then $A_{\mathcal{DG}}((s, \varphi_1)_X) = 1$, and by the induction hypothesis the defender has a universal winning strategy from $(s, \varphi_1)_X$. From this it follows that the defender has a universal winning strategy from $(s, \varphi_1 \vee \varphi_2)_X$: the defender gets to choose and picks $(s, \varphi_1)_X$ as the successor.

  $(s, \varphi_1 \wedge \varphi_2)_X$ Suppose $A_{\mathcal{DG}}((s, \varphi_1 \wedge \varphi_2)_X) = 1$ and therefore $level(s, \varphi_1 \wedge \varphi_2)_X = k < \infty$. Then we need to show that the defender has a universal winning strategy from this configuration.

  Both $(s, \varphi_1)_X$ and $(s, \varphi_2)_X$ have levels less than k and are marked 1. By the induction hypothesis the defender has a universal winning strategy from both successors, and it does not matter which the attacker picks the defender still has a universal winning strategy.

$(s, \langle\alpha\rangle\varphi)_X$ Suppose $A_{\mathcal{DG}}((s, \langle\alpha\rangle\varphi)_X) = 1$ and therefore $level((s, \langle\alpha\rangle\varphi)_X) = k$. Then we need to show that the defender has a universal winning strategy from this configuration.

There must be exists at least one hyperedge with a single target configuration with a lower $level((s_j, \varphi)_X) < k$ implying $A_{\mathcal{DG}}((s_j, \varphi)_X) = 1$. By the induction hypothesis the defender has a universal winning strategy from that configuration. Then the defender also has a universal winning strategy from $(s, \langle\alpha\rangle\varphi)_X)$. The defender simply picks $(s_j, \varphi)_X)$.

The proof is similar for weak-exist-modality.

$(s, [\alpha]\varphi)_X$ Suppose $A_{\mathcal{DG}}((s, [\alpha]\varphi)_X) = 1$ implying $level((s, [\alpha]\varphi)_X) = k$. Then we need to show that the defender has a universal winning strategy from this configuration.

There exists a single hyperedge with multiple target configurations $(s_i, \varphi)_X$ where $1 \leq i \leq n$ for some $n \geq 1$. By definition of level is must be the case that $level((s_i, \varphi)_X) < k$ implying that $A_{\mathcal{DG}}((s_i, \varphi)_X) = 1$, for all $1 \leq i \leq n$. Then by the induction hypothesis the defender has a universal winning strategy from all of them and it does not matter which the attacker picks. If there is no configurations to pick from the attacker is stuck and the defender has won.

The proof is similar for weak-forall-modality.

$(s, X)_X$ Suppose $A_{\mathcal{DG}}((s, X)_X) = 1$ implying $level((s, X)_X) = k < \infty$. Then we must show the attacker has a universal winning strategy.

Based on the construction in Figure 3.2i, it must be the case that $level((s, \varphi)_X) = k - 1$ implying $A_{\mathcal{DG}}((s, \varphi)_X) = 1$. By the induction hypothesis the defender has a winning strategy from $(s, \varphi_X)_X$. Since the successor configuration for $(s, X)_X$ is uniquely determined (it is $(s, \varphi_X)_X$) the defender has a universal winning strategy from $(s, X)_X$ also.

For the inductive case the rules are similar, except now we must handle the context switch made possible by the indirect references:

"$\Rightarrow$". Suppose the the defender has a universal winning strategy from $(s, X)_C$ where $C \neq X$ in k steps. We show by induction on the number of steps in the universal winning strategy that $A_{\mathcal{DG}}((s, X)_C) = 1$.

The defender must have a universal winning strategy from $(s, X)_X$ in $k-1$ steps implying that $A_{\mathcal{DG}}((s, X)_X) = 1$. From the construction Figure 3.2j there is a single hyperedge from $(s, X)_C$ to $(s, X)_X$. Then it must also be the case that $A_{\mathcal{DG}}((s, X)_C) = 1$.

"$\Leftarrow$". Suppose $A_{\mathcal{DG}}((s, X)_C) = 1$ implying $level((s, X)_C) = k < \infty$. Then we show by induction on the level of a configuration that the defender has a universal winning strategy.

Since $level((s, X)_C) = k$ it must be the case that $level((s, X)_X) = 1$. Then by the induction hypothesis the defender has a winning strategy from $(s, X)_X$. Since the successor configuration for $(s, X)_C$ is uniquely determined the defender then has a universal winning strategy from $(s, X)_C$ also. □

We have showed the relation between satisfiability and universal winning strategy, and the relation between universal winning strategy and the assignment to vertices in the dependency graph. This makes it possible to show the relation between satisfiability and the fixed-point assignment.

Corollary 3.16

Let $T(\texttt{Proc}, \texttt{Act}, \rightarrow)$ be a LTS where and φ an acyclic recursive HML formula. Then for all $s \in \texttt{Proc}$ it is the case that $s \models \varphi$ if and only if $A_{\mathcal{DG}}((s, \varphi)_\epsilon) = 1$.

Proof. Assume $s \models \varphi$. By Theorem 3.12 it must be the case that the defender has a universal winning strategy from $(s, \varphi)_\epsilon$. From Theorem 3.15 we then know that if the defender has a universal winning strategy it must be the case that $A_{\mathcal{DG}}((s, \varphi)_\epsilon) = 1$. For the other direction assume $A_{\mathcal{DG}}((s, \varphi)_\epsilon) = 0$. Then the defender does not have a universal winning strategy implying the attacker has one. If the attacker has a universal winning strategy then $s \not\models \varphi$. □

With this final proof we have shown that satisfiability, fixed-point assignments, and universal winning strategies are related to each other. This completes the omission of proofs for multiple acyclic recursive HML that was left unfinished in previous work. In the next chapter we use recursive HML formulae to describe differences in behaviour of non-bisimilar processes.

Distinguishing Formula

4

In Chapter 2 it is described that two processes are bisimilar they exhibit similar behaviour, and if they are not bisimilar they have behave differently. The CCS processes $P \stackrel{def}{=} a.b.0 + a.c.0$ and $Q \stackrel{def}{=} a.(b.0 + c.0)$ are not bisimilar. After P takes an a-transition the resulting process can only perform a b-transition or a c-transition, whereas after Q takes an a-transition it can do both. There is a theorem describing that bisimilar processes satisfies the same HML formulae.

Theorem 4.1 — *Hennessy and Milner [8]*

Let

$$(\texttt{Proc}, \texttt{Act}, \{\stackrel{a}{\rightarrow} | \ a \in \texttt{Act}\}),$$

be an LTS generated from a *weakly guarded* CCS expression. Assume that P, Q are processes in `Proc`, then:

$$P \sim Q \text{ if and only if } \forall \varphi \ P \models \varphi \Leftrightarrow Q \models \varphi,$$

without the modalities $\langle\langle \alpha \rangle\rangle$ and $[[\alpha]]$. And

$$P \approx Q \text{ if and only if } \forall \varphi \ P \models \varphi \Leftrightarrow Q \models \varphi,$$

without the modalities $\langle \alpha \rangle$ and $[\alpha]$.

The theorem implies that if two processes P and Q are not bisimilar there exists a formula φ such that either P or Q satisfies it, but not both. Let $\varphi = \langle a \rangle(\langle b \rangle tt \wedge \langle c \rangle tt)$. Then $P \not\models \varphi$, $Q \models \varphi$ and we call φ a distinguishing formula for P and Q. This chapter describes a method for finding distinguishing formulae for non-bisimilar processes.

4.1 Distinguishing Formula

From Theorem 4.1 it is known that two CCS processes are bisimilar if and only if they satisfy exactly the same formula. If two processes are not bisimilar then there must be a formula that distinguishes the two processes. If processes P and Q are not bisimilar, we call any formula that one of the processes satisfies, but not the other, a distinguishing formula. A distinguishing formula can aid in understanding why two processes are not bisimilar.

In Section 2.2, we described how to determine bisimilarity for CCS processes by constructing a dependency graph and computing the minimum pre-fixed-point for that dependency graph. Each vertex in the dependency graph corresponds to a process pair. The fixed-point computation assigns either 1 or 0 to all vertices. After termination of the fixed-point computation, all vertices assigned 0 mean the processes in the corresponding pairs are bisimilar, whereas the assignment 1 signals that the corresponding process pairs are not bisimilar.

Based on the fixed-point assignment of vertices in the dependency graph, we will show how to create a formula φ, such that for two processes $P \not\sim Q$ it is the case that $P \models \varphi$ and $Q \not\models \varphi$.

Before describing the formal definition of the construction, we provide an example using the following non-bisimilar processes:

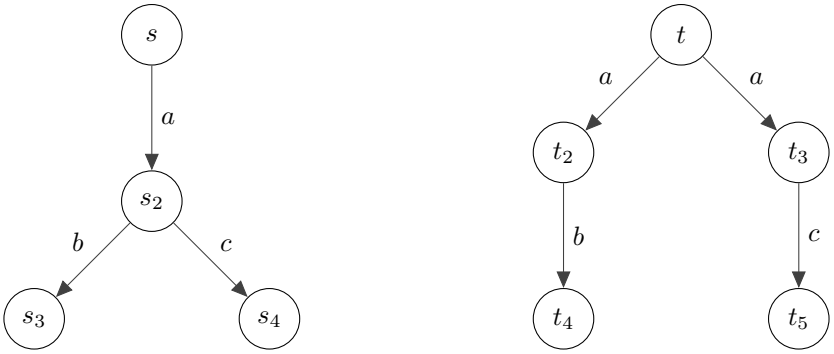


Figure 4.1: Two non-strongly-bisimilar LTSs.

The dependency graph, marked with the corresponding pre-fixed-point assignment, is shown in Figure 4.2.

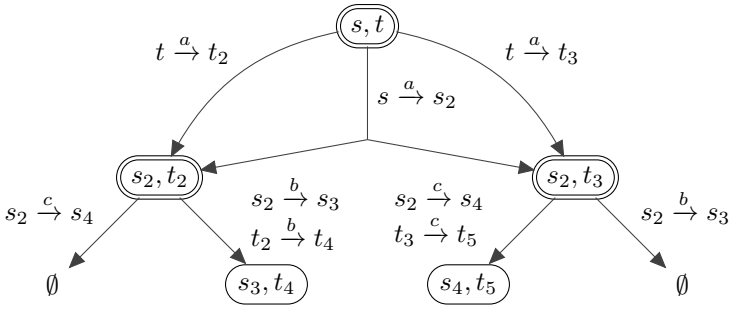


Figure 4.2: Bisimulation dependency graph for process s and t.

The vertices represent process pairs, and are marked according to the minimum pre-fixed-point A_{min}. For a single bordered vertex x, it is the case that $A_{min}(x) = 0$, and for a double bordered vertex x, it is the case that $A_{min}(x) = 1$. A marking of 1 for a vertex indicates that the processes in the pair are not bisimilar, whereas, a marking of 0 indicates they are bisimilar. As an example, $s_3 \sim t_4$, but $s_2 \not\sim t_2$.

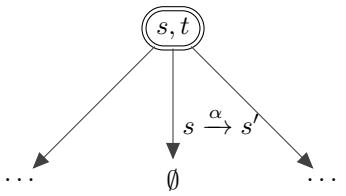
The edges are labelled with the transitions that *triggered* the construction of the hyperedge. The hyperedge originating in (s, t), with both (s_2, t_2) and (s_2, t_3) as target vertices, was triggered because $s \xrightarrow{a} s_2$ and t could match with both $t \xrightarrow{a} t_2$ and $t \xrightarrow{a} t_3$. The reason (s_2, t_2) is marked one, is because of the hyperedge with the empty target set, $\emptyset$, which is caused by s being able to take a c transition, but t_2 is unable to do that.

From the assignment to the dependency graph it is possible to extract the information necessary to construct a distinguishing formula. The formula that distinguishes s_2 and t_2 is thus $\langle c\rangle tt$. It is the case that $s_2 \models \langle c\rangle tt$, and that $t_2 \not\models \langle c\rangle tt$. For the formula distinguishing s and t, we already have a formula distinguishing s_2 and t_2 to build upon. By prepending $[a]$ to our formula, getting $[a]\langle c\rangle tt$, we have a distinguishing formula for s and t since $s \models [a]\langle c\rangle tt$, and $t \not\models [a]\langle c\rangle tt$. Note that prepending $\langle a\rangle$ to the formula getting $\langle a\rangle\langle c\rangle tt$ would not work, since that formula is also satisfied by t.

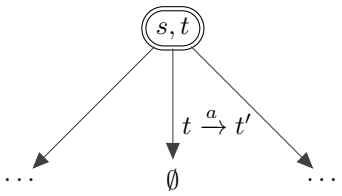
Note that there are other formulae that distinguish s and t, like the symmetric case $[a]\langle b\rangle tt$. A third one is $\langle a\rangle(\langle b\rangle tt \wedge \langle c\rangle tt)$. The third one might be considered more complex, based on its syntactical length and nesting. If only one distinguishing formula is required, it might be desirable to return the simplest one. We will describe how to accomplish this later in this chapter.

4.2 Construction of Distinguishing Formula

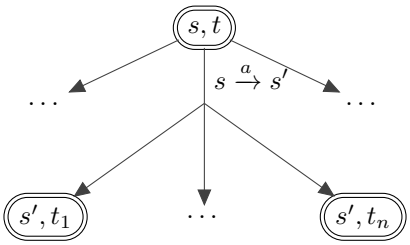
The formula is inductively constructed from the dependency graph. The formula found is one satisfied by s, but not t, for a vertex (s, t) representing $s \sim t$. Nodes marked 1 are followed until reaching a hyperedge with no targets (the target is the empty set). There are two base cases, and two recursive cases. The two base cases are when a vertex has a hyperedge with the empty target set, which trivially marks the vertex 1. A vertex (s, t), can represent one case when it has a hyperedge to the empty set because $s \xrightarrow{\alpha} s_2$, but t has no α-transition. The other base case is symmetric. The recursive cases are similar, but handle the cases where there is a hyperedge the target set is not empty, and all vertices in the target set are marked 1.



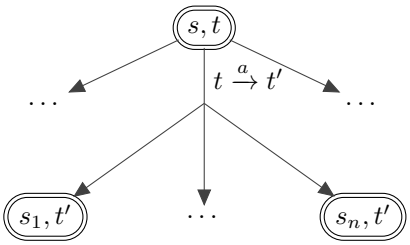
(a) Case 1: $\varphi = \langle\alpha\rangle tt$



(b) Case 2: $\varphi = [a]ff$



(c) Case 3: $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ where for $1 \leq i \leq n$, $s' \models \varphi_i$ and $t_i \not\models \varphi_i$.



(d) Case 4: $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ where for $1 \leq i \leq n$, $s_i \models \varphi_i$ and $t' \not\models \varphi_i$.

Figure 4.3: Cases for construction of distinguishing formula.

Figure 4.3 shows the various cases. From these cases we define the distinguishing formula function DF, implemented by Algorithm 4.1 for a vertex (s,t).

Algorithm 4.1 Distinguishing Formula (DF) for $s \not\sim t$.

Input: State s and t s.t. $s \not\sim t$.
Output: Formula φ s.t. $s \models \varphi$ and $t \not\models \varphi$.

1: ▷ Build dependency graph $G^T_{\sim} = (V, E)$ and compute minimum-fixed point and levels, using strong bisimilarity reduction on s and t.
2: $H \leftarrow \{((s,t),T) \mid ((s,t),T) \in E$ and for all $u \in T$ we have $level(u) < level((s,t))\}$
3: **if** $((s,t),\emptyset) \in H$ **then**
4: **if** $s \xrightarrow{\alpha} s'$ and $t \not\xrightarrow{\alpha}$ **then**
5: **return** $\langle\alpha\rangle tt$
6: **if** $t \xrightarrow{\alpha} t'$ and $s \not\xrightarrow{\alpha}$ **then**
7: **return** $[\alpha]ff$
8: **else**
9: Pick $h = ((s,t),T) \in H$
10: **if** h was triggered by $s \xrightarrow{\alpha} s'$ **then**
11: **return**

$$\langle\alpha\rangle \bigwedge_{(s',t')\in T} DF((s',t'))$$

12: **else** ▷ h was triggered by $t \xrightarrow{\alpha} t'$
13: **return**

$$[\alpha] \bigvee_{(s',t')\in T} DF((s',t'))$$

Theorem 4.2 — *Distinguishing Formula*

Let (s,t) be a vertex in a Bisimulation DG, and $\varphi = DF((s,t))$. Then it is the case that $s \models \varphi$, and that $t \not\models \varphi$.

Proof. Proof by induction on *level* of a vertex (s,t).

- If $level((s,t)) = 1$, then by definition of *level*, it is the case that the condition in line 3 is true, and also that at least one of the conditions in the if-statements in lines 4 or 6 is true.

  Assume the condition in the first if-statement holds: it is the case that $s \xrightarrow{\alpha} s'$ and $t \not\xrightarrow{\alpha}$. Then $s \models \langle\alpha\rangle tt$, but $t \not\models \langle\alpha\rangle ff$.

  For the other case, where $t \xrightarrow{\alpha} t'$ and $s \not\xrightarrow{\alpha}$, the formula $[\alpha]ff$ is constructed. Then $s \models [\alpha]ff$, since s cannot perform an α-action, but $t \not\models [\alpha]ff$, since it can perform an α-action.

- If $level((s,t)) > 1$, then by the definition of *level* there must be a hyperedge, $((s,t),T)$, such that $\forall t \in T.level(t) < level((s,t))$, implying that

the hyperedge is in H. By the construction of the dependency graph it is the case that if $s \xrightarrow{\alpha} s'$, then there exists a hyperedge for the matching α-transitions from t: $((s,t), \{(s',t') \mid t \xrightarrow{\alpha} t'\})$. There is a symmetric case for t. Since the *level* is larger than 1, the condition in line 3 is not true and the else branch in line 8 is taken. Then by definition of level, there must be at least one hyperedge $h = ((s,t), T)$ in H.

Assume the the condition in if-statement at line 10 is true. Let $\varphi_1 = DF((s',t_1)), \varphi_2 = DF((s',t_2)), \ldots, \varphi_n = DF((s',t_n))$, where $(s',t_i) \in T$ for $1 \leq i \leq n$. By the induction hypothesis, $s' \models \varphi_i$, and $t_i \not\models \varphi_i$. Since $s' \models \varphi_i$, it is also the case that $s' \models \bigwedge_1^n \varphi_i$. Since $s \xrightarrow{\alpha} s'$, then it must be the case that $s \models \langle\alpha\rangle \bigwedge_1^n \varphi_i$.

It is also the case that $t \not\models \langle\alpha\rangle \bigwedge_1^n \varphi_i$. Suppose for contradiction that $t \models \langle\alpha\rangle \bigwedge_1^n \varphi_i$. This implies there is some $t \xrightarrow{\alpha} t_i$ such that $t_i \models \varphi_i$, contradicting the induction hypothesis that $t_i \not\models \varphi_i$.

The final case for the else branch in line 12 is analogous.

□

The algorithm can be modified slightly for finding distinguishing formulae for weak non-bisimilar processes. Instead of computing the strong bisimulation dependency graph $G^T_{\sim}$, the weak bisimulation graph $G^T_{\approx}$ is constructed, and rather than prepending $\langle\alpha\rangle$ or $[\alpha]$, the weak variants $\langle\langle\alpha\rangle\rangle$ and $[[\alpha]]$ are used instead, and the transition relation $\Rightarrow$ is used instead of $\rightarrow$.

Formulae may contain redundant parts that can safely be removed without changing the semantics of the formula. Any weak exists prefixed or postfixed with a weak exists on τ-action can be removed. The same holds for weak for all.

Lemma 4.3 — *Tau Removal*

For any process p and formula φ the following biimplications holds, and the right side is considered a simplification of the left side.

$$p \models \langle\langle\tau\rangle\rangle\langle\langle\alpha\rangle\rangle\varphi \Leftrightarrow p \models \langle\langle\alpha\rangle\rangle\varphi$$
$$p \models \langle\langle\alpha\rangle\rangle\langle\langle\tau\rangle\rangle\varphi \Leftrightarrow p \models \langle\langle\alpha\rangle\rangle\varphi$$
$$p \models [[\tau]][[\alpha]]\varphi \Leftrightarrow p \models [[\alpha]]\varphi$$
$$p \models [[\alpha]][[\tau]]\varphi \Leftrightarrow p \models [[\alpha]]\varphi$$

Proof. We consider only the first example prefixed weak-exist on τ-actions. The proof for the others are similar.

Suppose $p \models \langle\langle\tau\rangle\rangle\langle\langle\alpha\rangle\rangle\varphi$. Then there is some process $p' \models \langle\langle\alpha\rangle\rangle\varphi$ and $p'' \models \varphi$, where $p \stackrel{\tau}{\Rightarrow} p' \stackrel{\alpha}{\Rightarrow} p''$. This is equivalent to $p(\xrightarrow{\tau})^* p' (\xrightarrow{\tau})^* q \xrightarrow{\alpha} q' (\xrightarrow{\tau})^* p''$. Then p' can be subsumed and $p(\xrightarrow{\tau})^* q \xrightarrow{\alpha} q' (\xrightarrow{\tau})^* p''$, which means $p \stackrel{\alpha}{\Rightarrow} p''$ and therefore $p \models \langle\langle\alpha\rangle\rangle\varphi$.

For the other direction suppose $p \models \langle\langle\alpha\rangle\rangle\varphi$. Then there is some process $p' \models \varphi$, where $p \stackrel{\alpha}{\Rightarrow} p'$, which is equivalent to $p(\xrightarrow{\tau})^* q \xrightarrow{\alpha} q' (\xrightarrow{\tau})^* p'$. It is

always possible for p to take zero τ-transitions to itself: $p(\xrightarrow{\tau})^* p$, therefore $p(\xrightarrow{\tau})^* p(\xrightarrow{\tau})^* q \xrightarrow{\alpha} q'(\xrightarrow{\tau})^* p' = p \stackrel{\tau}{\Rightarrow} p \stackrel{\alpha}{\Rightarrow} p'$, which imply $p \models \langle\langle\tau\rangle\rangle\langle\langle\alpha\rangle\rangle\varphi$. □

4.3 Finding the Simplest Distinguishing Formula

In Section 4.1, there were several formulae that distinguished process s and t. One was $[a]\langle c\rangle tt$, and another was $\langle a\rangle(\langle b\rangle tt \wedge \langle c\rangle tt)$. We argue that the first formula is simpler than the other because it has a shorter syntactical length. While syntactical length is an indicator of complexity, a long formula does not necessarily have to be complex to understand.

One possible metric is the total number of conjunctions or disjunctions in a formula. The left and right side of conjunctions and disjunctions can be thought of as branches to be satisfied, with nesting further complicating understanding.

Another metric is modal depth. The modal depth is the longest nesting of modal operators in a formula:

Definition 4.4 — *Modal Depth for HML*

For an HML formula φ we define the modal depth MD as:

$$MD(\varphi) = \begin{cases} 0 & \text{if } \varphi = tt \\ 0 & \text{if } \varphi = ff \\ max(MD(\varphi_i) \mid 1 \leq i \leq n) & \text{if } \varphi = \varphi_1 \wedge \ldots \wedge \varphi_n \\ max(MD(\varphi_i) \mid 1 \leq i \leq n) & \text{if } \varphi = \varphi_1 \vee \ldots \vee \varphi_n \\ 1 + MD(\varphi') & \text{if } \varphi = \langle\alpha\rangle\varphi' \\ 1 + MD(\varphi') & \text{if } \varphi = [\alpha]\varphi' \\ 1 + MD(\varphi') & \text{if } \varphi = \langle\langle\alpha\rangle\rangle\varphi' \\ 1 + MD(\varphi') & \text{if } \varphi = [[\alpha]]\varphi' \end{cases}$$

We conjecture that the general problem of determining whether it is possible to find a distinguishing formula for two processes with at most a given modal depth and number of conjunction and disjunctions is NP-hard. We were unable to prove this conjecture, but compensate by offering a greedy algorithm that simplifies formulae.

The greedy algorithm, shown in Algorithm 4.2, is itself a modified version of Algorithm 4.1. The change is that whenever it creates a conjunction and disjunction, in lines 11 and 13, it simplifies the formula by removing redundant terms by calling SIMPLIFY with either the conjunction or disjunction as argument, and the process pairs reached by taking α-transitions from s and t.

The function SIMPLIFY first initializes the result terms denoted by D to the empty set. Assume it was called with a conjunction and therefore the branch starting at line 16 has been taken. At this point, by induction, we know that for all $(s', t') \in T$: $s' = s''$ for all $(s'', t'') \in T$, and that $s' \models \varphi_i$, and finally that there exists some φ_i such that $t' \models \varphi_i$, where $1 \leq i \leq n$.

The loop invariant is that for all $s \in S$ it is the case there is no $\varphi \in D$ such that $s \not\models D$. In the loop any formula φ_i, that is not satisfied by more processes

Algorithm 4.2 Greedy Distinguishing Formula (*GDF*) for $s \not\sim t$.

Input: State s and t s.t. $s \not\sim t$.
Output: Formula φ s.t. $s \models \varphi$ and $t \not\models \varphi$.

▷ Build dependency graph $G^T_{\sim} = (V, E)$ and compute minimum-fixed point and levels, using strong bisimilarity reduction on s and t.
$H \leftarrow \{((s,t),T) \mid ((s,t),T) \in E$ and for all $u \in T$ we have $level(u) < level((s,t))\}$
if $((s,t),\emptyset) \in H$ **then**
  if $s \xrightarrow{\alpha} s'$ and $t \not\xrightarrow{\alpha}$ **then**
    return $\langle\alpha\rangle tt$
  if $t \xrightarrow{\alpha} t'$ and $s \not\xrightarrow{\alpha}$ **then**
    return $[\alpha]ff$
else
  Pick $h = ((s,t),T) \in H$
  if h was triggered by $s \xrightarrow{\alpha} s'$ **then**
    return

$$\langle\alpha\rangle\textsc{Simplify}\Big(\bigwedge_{(s',t')\in T} GDF((s',t')),T\Big)$$

  else ▷ h was triggered by $t \xrightarrow{\alpha} t'$
    return

$$[\alpha]\textsc{Simplify}\Big(\bigvee_{(s',t')\in T} GDF((s',t')),T\Big)$$

function Simplify(φ, T)
  $D \leftarrow \emptyset$
  if $\varphi = \bigwedge_{i=1}^{n} \varphi_i$ **then**
    $S \leftarrow \{t' \mid (s',t') \in T\}$
    while $S \neq \emptyset$ **do**
      Pick $\varphi_i \in \varphi$ s.t. $|\textsc{NotSatisfy}(S,\varphi_i)| \geq |\textsc{NotSatisfy}(S,\varphi_j)|$ for all $\varphi_j \in \varphi$
      $D \leftarrow D \cup \{\varphi_i\}$
      $S \leftarrow S \setminus \textsc{NotSatisfy}(S,\varphi_i)$
    return $\bigwedge_{\varphi\in D} \varphi$
  else ▷ $\varphi = \bigvee_{i=1}^{n} \varphi_i$
    $S \leftarrow \{s' \mid (s',t') \in T\}$
    while $S \neq \emptyset$ **do**
      Pick $\varphi_i \in \varphi$ s.t. $|\textsc{Satisfy}(S,\varphi_i)| \geq |\textsc{Satisfy}(S,\varphi_j)|$ for all $\varphi_j \in \varphi$
      $D \leftarrow D \cup \{\varphi_i\}$
      $S \leftarrow S \setminus \textsc{Satisfy}(S,\varphi_i)$
    return $\bigvee_{\varphi\in D} \varphi$
function Satisfy(S, φ)
  return $\{s \mid s \in S$ and $s \models \varphi\}$
function NotSatisfy(S, φ)
  return $\{s \mid s \in S$ and $s \not\models \varphi\}$

than any other formula is picked; this is the greedy part. That formula is then added to D. To maintain the invariant, all processes that does not satisfy φ_i is removed from S.

At the end of the loop S is empty and for all $(s', t') \in T$, it is the case that $s \models \varphi_i$, and $t' \not\models \varphi_i$ for all $\varphi_i \in D$. Since we only want a single formula all the terms in D are conjoined as the result. The branch for disjunction works in a symmetric manner.

Example 4.5 — *Simplify*

We use the following definitions and formulae.

$$\begin{aligned} P_1 &= a.(b + c).0 \\ P_2 &= a.b.0 \\ P_3 &= a.c.0 \\ \varphi &= \langle a \rangle [c] ff \wedge \langle a \rangle \langle c \rangle tt \wedge \langle a \rangle \langle b \rangle tt \end{aligned}$$

For the first iteration $D = \emptyset$ and $S = \{P_1, P_2, P_3\}$ and the following processes do not satisfy the corresponding term:

$$\begin{aligned} \textsc{NotSatisfy}(S, \langle a \rangle [c] ff) &= \{P_1, P_3\} \\ \textsc{NotSatisfy}(S, \langle a \rangle \langle c \rangle tt) &= \{P_2\} \\ \textsc{NotSatisfy}(S, \langle a \rangle \langle b \rangle tt) &= \{P_3\} \end{aligned}$$

The term is added to D and now $D = \{\langle a \rangle [c] ff\}$. Both P_1 and P_3 are removed from S which now becomes $S = \{P_2\}$. The only term left to choose from is $\langle a \rangle \langle c \rangle tt$ and the remaining process, P_2, does not satisfy it. The term is added to D which becomes $D = \{\langle a \rangle [c] ff, \langle a \rangle \langle c \rangle tt\}$, and the process P_2 is removed from S which now is empty. Now the while condition is false and the result $\langle a \rangle [c] ff \wedge \langle a \rangle \langle c \rangle tt$ is returned.

The greedy algorithm may not find the solution with the fewest terms. The problem of selecting the minimal subset of terms from a conjunction or disjunction is NP-hard. This will be shown by reducing the CNF-SAT to a problem named the k-matrix problem, and then reducing the matrix problem to our problem of reducing the number of conjunctions and disjunctions in the formula.

CNF-SAT

To remind, CNF-formulae are conjunctions of clauses, which themselves are disjunctions of s literals [16]. The formal definition used here is:

$$\begin{aligned} \varphi &= C_1 \wedge C_2 \wedge \cdots \wedge C_p \\ C_a &= l_a^1 \vee l_a^2 \vee \cdots \vee l_a^{q_a} \text{ where } q_a \in \mathcal{N} \\ l_a^b &= x_r \text{ or } \overline{x_r} \text{ where } 1 \leq r \leq s. \end{aligned}$$

The CNF-SAT problem about determining whether a CNF-formula is satisfiable: is it possibly to assign true and false to all variables of a formula such that the formula is true? The following formula is in CNF-form and is satisfiable:

$$(\overline{x} \vee \overline{y} \vee w) \wedge (x \vee \overline{y} \vee w \vee z) \wedge z$$

This assignment shows the formula is satisfiable:

$$x = \mathit{false},\ y = \mathit{true},\ z = \mathit{true},\ w = \mathit{false}$$

k-Matrix Problem

We reduce the CNF-SAT problem to the k-matrix problem.

Definition 4.6 — *k-Matrix Problem*

The k-column matrix problem, is the problem of determining whether it is possible to pick $k \leq n$ columns from M a $m \times n$ matrix, with elements being either 1 or 0, such that each row has at least one number 1 in the selected columns. If that is possible we say that M is solvable under the k-matrix problem.

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

For $k = 1$, it is not possible to pick a single column from the matrix above such that all rows for the selected columns are 1. It is possible to satisfy if $k = 2$, by selecting column 1 and 3, all rows have at least one element that is 1 in the selected columns. Another valid example is picking column 3 and 4. If problem is solvable for some k, it is also solvable for $k + 1 \leq n$; just pick the same k columns and some other arbitrary column.

Reduction from CNF-SAT to k-Matrix Problem

We reduce the CNF-SAT problem to the k-Matrix problem by constructing the corresponding matrix. For a CNF-formula with p clauses and s variables we create a $(p + s) \times 2s$ matrix for the $k = s$ problem.

The idea is to construct a matrix such that if there are p clauses the matrix has a row for each clause and variable. The row for each clause ensures that in order for the clause to be true, a column corresponding to a literal in the clause must be picked. The row for each variable ensures that a variable is picked to be either true or false. Finally, there are two columns for each variable, one for the non-negated literal, and one for the negated one, each corresponding

to assigning either true or false to the corresponding variable. The reduction from a CNF-formula to a matrix M is shown in Equation 4.1.

$$M(i,j) = \begin{cases} 1 \text{ if } i \leq p, j \leq s \text{ and } x_j \in C_i \\ 1 \text{ if } i \leq p, s < j \leq 2s \text{ and } \overline{x_{j-s}} \in C_i \\ 1 \text{ if } p < i \leq p+s, \text{ and } j = i-p \text{ or } j = i-p+s \\ 0 \text{ otherwise} \end{cases} \tag{4.1}$$

The first two cases covers ones in the the top part of the matrix and represents the clauses with 1 indicating the literal appears on the clause. The left half covers non-negated variables, whereas the right covers the negated ones.

The third case, requires either the column of the non-negated literal or the negated literal to be picked in order for the corresponding row to have at least a single 1. Since it is a k-matrix problem, and $k = s$, it is necessary to choose s-columns. Either the column for the non-negated literal or the negated literal has to be chosen: if both columns are picked, it is not possible to select a column for some other variable, meaning one of the rows does not have a 1 in any of the selected columns, and thus the result is not valid.

Consider constructing the corresponding k-column matrix problem from the example from before:

$$\varphi = \underbrace{\overline{x} \vee \overline{y} \vee w)}_{C_1} \wedge \underbrace{(x \vee \overline{y} \vee w \vee z)}_{C_2} \wedge \underbrace{z}_{C_3}$$

The formula has three clauses and four variables, which implies that for the corresponding k-colum matrix problem, $k = 4$, and the matrix is of dimensions 7×8. By the definition above for the matrix entries, the matrix looks like this (the left column and top row are not part of the matrix; empty fields represent 0):

$k=4$	x	y	z	w	$\overline{x}$	$\overline{y}$	$\overline{z}$	$\overline{w}$
C_1				1	1	1		
C_2	1		1	1		1		
C_3			1					
$x_=$	1				1			
$y_=$		1				1		
$z_=$			1				1	
$w_=$				1				1

The formula is satisfiable: by selecting columns 1 (x), 3 (z), 4 (w), and 6 ($\overline{y}$), all the rows of those columns have at least a single 1.

Theorem 4.7 — *CNF-SAT Reduction to k-Matrix Problem*

Let φ be a formula in CNF with k variables, and let M be the construction described in Equation 4.1 from φ. Then φ is satisfiable if and only if M has a solution in the k-matrix problem.

Proof. Let $\varphi = C_1 \wedge \ldots \wedge C_p$ be CNF-formula with s variables $x_1, x_2, \ldots, x_s$, and let M be the matrix constructed from φ using Equation 4.1.

1. Assume φ is satisfiable. Then there exists an assignment assigning each variable x_i, where $1 \leq i \leq s$, true or false, such that evaluating φ under this assignment yields true. Then M is satisfisable for the s-matrix problem: it is possible to pick s columns in M such that each row has at least one number 1 in each of the selected columns. For each variable x_i, if x_i is true pick column i otherwise pick column $s + i$. Since each clause have at least one literal that evaluate to true, at least one column corresponding to a literal evaluating to true must have been selected for all of the top p-rows. Then each of those rows has at least one number 1 in the selected columns. Also since each variable is only true or false, all of the s bottommost rows in the matrix has at least one number 1 in the selected columns.

2. Assume M is satisfiable for the s-matrix problem. Then there is an selection of columns Y, where $|Y| = s$ and $\forall_{y \in Y} 1 \leq y \leq 2s$ such that all rows of M has at least one number 1 in a column $y \in Y$. For all variables x_i, $1 \leq i \leq s$ in φ, assign true to x_i if $i \in Y$, otherwise assign false to x_i. Since each of the p topmost rows have at least one number 1 in a selected column, a literal must evaluate to true in each clause under the chosen assignment. Since each of the s bottommost rows also have at least one number 1 in a selected column, all variables have been assigned either true or false.

□

k-Terms Problem

The k terms is the decision problem of whether some terms can be removed from a formula.

Definition 4.8 — k-*Terms Problem*

Given formula $\varphi_1 \wedge \cdots \wedge \varphi_m$ and processes $t_i \not\models \varphi_i$ for all $1 \leq i \leq m$, and number k, is there a subset $X \subseteq \{1, \ldots, m\}$, such that $|X| \leq k$ and for all $1 \leq j \leq m$ there exists $i \in X$ such that $t_j \not\models \varphi_i$.

Alternatively the formula may be a disjunction $\varphi_1 \vee \cdots \vee \varphi_m$ and processes $s_i \models \varphi_i$ where for all $1 \leq i \leq m$, and number k, is there a subset $X \subseteq \{1, \ldots, m\}$, such that $|X| \leq k$ and for all $1 \leq j \leq m$ there exists $i \in X$ such that $s_j \not\models \varphi_i$.

The processes and formula from Example 4.5 are repeated here:

$$
\begin{aligned}
P_1 &= a.(b.0 + c.0) \\
P_2 &= a.b.0 \\
P_3 &= a.c.0 \\
\varphi &= \langle a \rangle [c] ff \wedge \langle a \rangle \langle c \rangle tt \wedge \langle a \rangle \langle b \rangle tt
\end{aligned}
$$

There is a solution for the 2-Terms Problem: the two terms $P_1, P_3 \not\models \langle a \rangle [c] ff$ and $P_2 \not\models \langle a \rangle \langle c \rangle tt$ are enough. There is no solution to the 1-Term Problem.

Reduction from k-Matrix Problem to k-Terms Problem

To show that the k-Terms Problem is NP-hard we make a reduction from the k-matrix problem. First construct a LTS with a process for each row of the matrix along with a special process 0. Each column is represented by an action. A process can take an action to process 0, if and only if the corresponding entry in the matrix is 0.

Let M be a $m \times n$ matrix consisting of only ones or zeros.
Let $T = (S, Act, \rightarrow)$ be an LTS where $S = \{0, 1, \ldots, m\}$, $Act = \{1, \ldots, n\}$, and $\rightarrow = \{(p \xrightarrow{\alpha} 0) \mid M(p, \alpha) = 0\}$. Let $\varphi_i = \langle i \rangle tt$ for $1 \leq i \leq n$.

Theorem 4.9

The k-Matrix problem for M is solvable if and only if the k-Terms Problem is solvable for formula $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$ or formula $\varphi = \varphi_1 \vee \ldots \vee \varphi_n$ and for processes S.

Proof. For the proof assume φ is a conjunction. The proof for disjunction is similar.

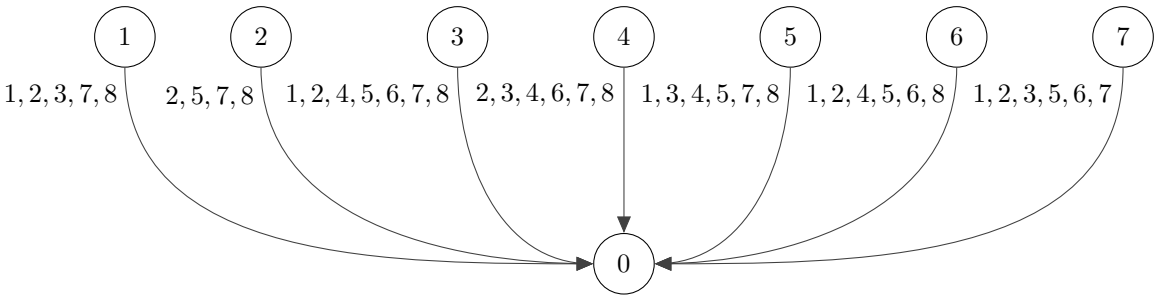
Suppose the k-Matrix problem is solvable for M. Then there is a some columns $C \subseteq \{1, \ldots, n\}$ with $|C| \leq k$. For each row r there must be a column $i \in C$ such that $M(r, i) = 1$. By definition this means $r \xrightarrow{i}$, implying $r \not\models \langle i \rangle tt$. Then the solution to the k-Term Problem is the formulae $\{\langle i \rangle tt \mid i \in C\}$.

For the other direction assume the k-Terms Problem is solvable. Then there is a subset $X \subseteq \{1, \ldots, n\}$, $|X| \leq k$, such that for all p_i, $1 \leq i \leq m$ there exists $j \in X$ such that $p_i \not\models \langle j \rangle tt$. By the reduction that implies for all rows $1 \leq i \leq m$, there exists $j \in X$, such that $M(i, j) = 1$. Then the set X is a solution of columns for the k-Matrix problem. □

We continue the reduction with the same example. But instead each row of the matrix is annotated with processes, and the columns are annotated with actions.

$k = 4$	1	2	3	4	5	6	7	8
1				1	1	1		
2	1		1	1		1		
3			1					
4	1				1			
5		1				1		
6			1				1	
7				1				1

Construction the corresponding LTS yields:



It can be seen how the LTS resembles the matrix. Process 2 can take the transition $2 \xrightarrow{7} 0$ because the entry $M(2,7)$ is blank. A solution to the 4-matrix problem was picking columns $1, 3, 4$ and 6. Then the terms $\langle 1\rangle tt$, $\langle 3\rangle tt$, $\langle 4\rangle tt$ and $\langle 6\rangle tt$ is a solution to the 4-terms problem; for all processes they do not satisfy at least one of the terms since $2, 4 \not\models \langle 1\rangle tt$, $3, 6 \not\models \langle 3\rangle tt$, $1, 7 \not\models \langle 4\rangle tt$ and $5 \not\models \langle 6\rangle tt$.

Equivalence Collapse

5

Equivalence collapse can be used to hide equivalent processes to simplify the visualised LTS shown in the explorer in Caal. This collapse can simplify the representation of complex systems, and provide an overview and give insights to the behavioural aspects of a process. We explain the theory behind the equivalence collapse using strong bisimulation, and proceed to explain how to implement equivalence collapse. We offer examples for equivalence collapse using both strong and weak bisimulation. The weak bisimulation collapse example also shows that not only can we simplify the LTS by collapsing equivalent processes, but we can also hide unimportant process behaviour to to simplify the LTS even further. This can be achieved by relabelling the action to τ, and visualise the LTS using weak bisimulation collapse.

5.1 Bisimulation Collapse

Consider two bisimilar processes $Q \sim R$, and then another process with a choice of both: $P \stackrel{def}{=} \alpha.Q + \alpha.R$. Since Q and R are bisimilar it is not possible to distinguish between them. Then there is no need to have both and it would not matter which choice was made, and P can be simplified by removing the choice: $P \stackrel{def}{=} \alpha.Q$.

Since bisimulation equivalence is an equivalence, the set of all processes can be divided into subset classes, for which all processes in a subset are all bisimilar. For each class it is possible to select a process as a *representative* for the class. Any process in the LTS can then be replaced by the representative for its class without changing the behaviour of the system.

Figure 5.1 shows an LTS where processes can be collapsed.

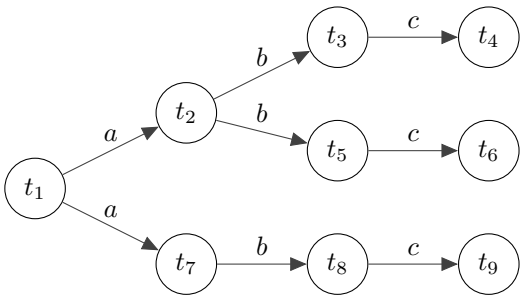


Figure 5.1: LTS that can be collapsed to simpler process.

The processes $t_4 \sim t_6 \sim t_9$ can not perform any actions at all and are therefore bisimilar and could be all be replaced by a single representative, say t_6, shown in Figure 5.2.

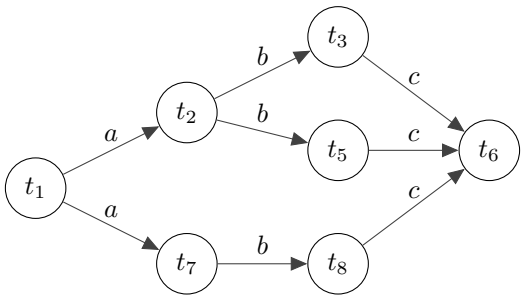


Figure 5.2: LTS that can be collapsed to simpler process.

Likewise the processes $t_3 \sim t_5 \sim t_8$ can all take a b-transition to a process that cannot do anything at all and are therefore bisimilar, and could all be replaced by a representative, say t_5. Likewise for t_2 and t_7 the representative is t_2 and finally we will end up as shown in Figure 5.3

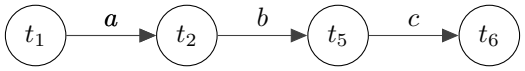


Figure 5.3: LTS that can be collapsed to simpler process.

The initial state (t_1) of this final LTS is bisimilar with the initial state of the first one.

We generalize this to any equivalence, not just bisimulation.

Definition 5.1 — *Equivalence Class*

Let $\equiv$ be equivalence relation. Then the equivalence class for a process p is the set of all processes that are equivalent, $[p]_{\equiv}$, defined as $[p]_{\equiv} = \{p' \mid p' \equiv p\}$.

For any LTS we can then define a corresponding collapsed version where all equivalent processe have been grouped.

Definition 5.2 — *Collapsed Labelled Transition System*

Let $T = (\texttt{Proc}, \texttt{Act}, \rightarrow)$ be a LTS where `Proc` is a set of states, `Act` a set of actions, $\rightarrow \subseteq \texttt{Proc} \times \texttt{Act} \times \texttt{Proc}$ the transition relation, and $\equiv$ be a equivalence relation between processes.

The collapsed labelled transition system $T_C = (\texttt{Proc'}, \texttt{Act}, \rightarrow_C)$ is then also an LTS with $\texttt{Proc'} \subseteq 2^{\texttt{Proc}}$, and $\rightarrow \subseteq \texttt{Proc'} \times \texttt{Act} \times \texttt{Proc'}$. The set of states `Proc'`, are all the equivalence classes, defined by $\texttt{Proc'} = \{[p]_{\equiv} \mid p \in \texttt{Proc}\}$. For any original transition in T, T_C has a transition between the corresponding partitions: $\rightarrow_C = \{[p]_{\equiv} \xrightarrow{\alpha} [p']_{\equiv} \mid p \xrightarrow{\alpha} p'\}$.

For strong and weak bisimulation, the collapsed transition system is bisimilar to the original.

Theorem 5.3

Let $T = (\texttt{Proc}, \texttt{Act}, \rightarrow)$ be a LTS. For all $p \in \texttt{Proc}$: $p \sim [p]_\sim$, and $p \approx [q]_\approx$.

Proof. To proof this we construct the bisimulation relation R. Let $R = \{(p, [p]_\sim) \mid p \in \texttt{Proc}\}$.

For the first direction, let $(p, [p]_\sim) \in R$ for any $p \in \texttt{Proc}$, and suppose $p \xrightarrow{\alpha} q$. Then by definition of $\rightarrow_C$ it is the case that $[p]_\sim \xrightarrow{\alpha}_C [q]_\sim$, and we have that $(q, [q]_\sim) \in R$.

For the other direction, suppose $[p]_\sim \xrightarrow{\alpha}_C [q]_\sim$. Then there must exists some processes $p' \in [p]_\sim$ and $q' \in [q]_\sim$ such that $p' \xrightarrow{\alpha} q'$. Since $p \sim p'$ there must be a transition $p \xrightarrow{\alpha} q''$ such that $q'' \sim q'$. Observe that $q'' \sim q$ which implies $q'' \in [q]_\sim$ which in turn implies $[q]_\sim = [q'']_\sim$. By definition of R it is the case that $(q'', [q]_\sim) \in R$.

For weak bisimulation the proof is similar, except the transition relation used, including in the construction of the collapsed LTS, is based on $\Rightarrow$ instead. □

Disjoint-Set Data Structure.

We need some way of representing groups of processes that are equivalent. For this we use a data structure called Disjoint-set data structure [6]. This data structure is managed using the three functions defined in Algorithm 5.1. The Disjoint-set data structure helps keeps track of disjoint subsets of processes that are equivalent. We start by calling the function MAKE-SET(x) with each process $x \in \texttt{Proc}$. The function is shown in Algorithm 5.1 on line 1. Each process is assigned itself as the representative process. This will later change as we link the equivalent processes. The FIND-SET function which is defined in

Algorithm 5.1 Disjoint-Set Procedures

```
1:  function MAKE-SET(x)
2:      x.parent = x
3:
4:  function FIND-SET(x)
5:      if x.parent == x then
6:          return x
7:      else
8:          return FIND-SET(x.parent)
9:
10: function UNION(x, y)
11:     xRoot = FIND-SET(x);
12:     yRoot = FIND-SET(y);
13:     xRoot.parent = yRoot;
```

Algorithm 5.1 on line 4, finds and returns the input process x's representative process. It does this by recursively calling itself with x's parent, until it reaches the representative process and the condition in line 5 is true.

When we have two equivalent processes x and y, we link them using the function UNION(x, y), defined in Algorithm 5.1 on line 10.

Lines 11-12 Find the representative process of the x and y process.

Line 13 Assign the representative process of x to y's representative process.

In our implementation we have optimised the algorithms using union by rank and path compression [6] to flatten the linked relationship between processes. The amortized running time of each operation is $\mathcal{O}(\alpha(n))$ where n is the number of MAKE-SET operations. The function $\alpha(n)$ is an extremely slowly growing function – so slow that in practice it might as well be a constant.

The equivalence collapse algorithm which is defined by Algorithm 5.2, uses the three functions, MAKE-SET, FIND-SET, and UNION, for linking equivalent processes so they later can be collapsed.

Algorithm 5.2 Equivalence Collapse

Input: LTS $T = (\texttt{Proc}, \texttt{Act}, \rightarrow)$ and equivalence relation $\equiv$
Output: LTS $T_C = (\texttt{Proc'}, \texttt{Act}, \rightarrow_C)$

```
1:  for all p ∈ Proc do
2:      MAKE-SET(p)
3:  for all p ∈ Proc do
4:      for all q ∈ Proc do
5:          pRoot = FIND-SET(p)
6:          qRoot = FIND-SET(q)
7:          if pRoot = qRoot then
8:              continue                    ▷ p and q have already been linked
9:          if pRoot ≡ qRoot then
10:             UNION(p,q)
11: Proc' = ∅
12: →_C = ∅
13: for all p --α--> q do
14:     →_C = →_C ∪ {(FIND-SET(p), α, FIND-SET(q))}
15:     Proc' = Proc' ∪ {FIND-SET(p), FIND-SET(q)}
16: return T_C = (Proc', Act, →_C)
```

Lines 1-2 We initialises the processes by calling MAKE-SET for every process in the LTS. This will cause the parent reference for p to point to p, i.e. causing a self-loop.

Lines 3-10 We then proceed to verify that p and q are equivalent. If they are equivalent we call UNION(p,q), to link the two processes.

Lines 11-16 We create the set `Proc'` and $\rightarrow_C$. For all transitions $(p, \alpha, q) \in \rightarrow$, we create a transition (FIND-SET$(q), \alpha,$ FIND-SET$(q))$ and add it to $\rightarrow_C$. We also add the representative processes FIND-SET(p) and FIND-SET(q) to `Proc'`.

The time complexity of our algorithm is $\mathcal{O}(|\texttt{Proc}|^2 \cdot \alpha(|\texttt{Proc}|) + |\texttt{Act}| \cdot \alpha(|\texttt{Proc}|))$ and the additional space used for the disjoint sets in computing the collapsed LTS is $\mathcal{O}(|\texttt{Proc}|)$.

Example 5.4 — *Strong Bisimulation Collapse*

Let $\texttt{Proc} = \{s, s_1, s_2, s_3, s_4, s_5\}$ of processes, where $s \sim s_1$, and $s_3 \sim s_4 \sim s_5$. An illustration of the LTS T is shown on Figure 5.4.

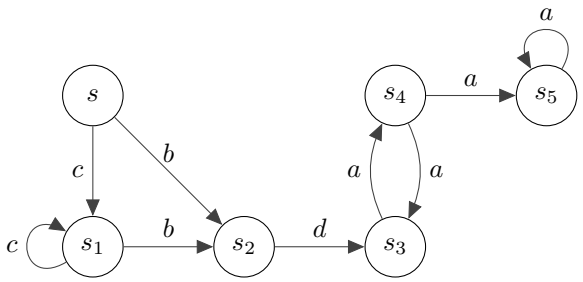


Figure 5.4: Illustration of LTS T.

We use Algorithm 5.2 to collapse the LTS T to T_C, for strong bisimulation. This is done in two steps.

Step 1 Create the Disjoint-set data structure, by linking all bisimilar processes together.

Step 2 Create `Proc'` and $\rightarrow_C$, using the Disjoint-set data structure.

We start with step 1, namely the lines 1-10 in Algorithm 5.2. We first apply the function MAKE-SET(x) on each process $x \in \texttt{Proc}$, as seen on Figure 5.5. Note that the arrows in the following figures illustrates the reference to the process's parent, and not their transitions.

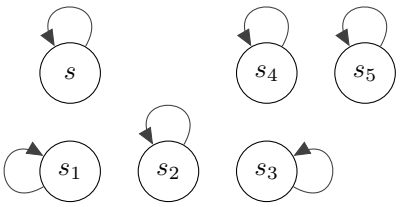


Figure 5.5: MAKE-SET has been applied on all processes.

We know that process $s \sim s_1$ and $s_3 \sim s_4$, and can therefore link them using the function UNION(s_3, s_4) and UNION(s, s_1), which is shown in Figure 5.6.

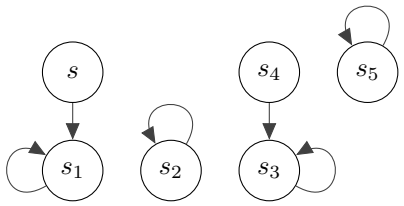


Figure 5.6: The function UNION(s_3, s_4) and UNION(s, s_1) has been applied

We also know that $s_4 \sim s_5$, and we apply UNION(s_4, s_5), and get the result shown in Figure 5.7.

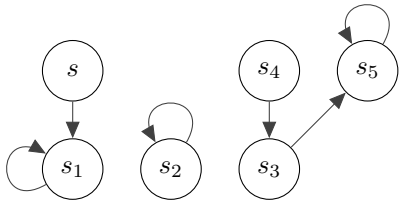


Figure 5.7: The function UNION(s_4, s_5) has been applied

We have now found a representative process for all bisimilar processes, which are $\{s_1, s_2, s_5\}$. Now we can proceed with step 2, namely lines 11-16 in Algorithm 5.2. For all transitions $p \xrightarrow{\alpha} q \in$ `Proc` we find the representaive of p and q and add them to the new transition relation $\rightarrow_C$, under the same action α. Thereafter we add the representative of p and q to `Proc'`. In Table 5.1 we have the execution of step 2. The returned LTS

i	$p \xrightarrow{\alpha} q$	$Find(p)$	$Find(q)$	$\rightarrow_C \cup \{(Find(p), \alpha, Find(q))\}$	`Proc'` $\cup$ $\{Find(p), Find(q)\}$
1	(s, c, s_1)	$\{s_1\}$	$\{s_1\}$	$\emptyset \cup \{(s_1, c, s_1)\}$	$\emptyset \cup \{s_1, s_1\}$
2	(s, b, s_2)	$\{s_1\}$	$\{s_2\}$	$\{(s_1, c, s_1)\} \cup \{(s_1, b, s_2)\}$	$\{s_1\} \cup \{s_1, s_2\}$
3	(s_1, c, s_1)	$\{s_1\}$	$\{s_1\}$	$\{(s_1, c, s_1), (s_1, b, s_2)\} \cup \{(s_1, c, s_1)\}$	$\{s_1, s_2\} \cup \{s_1, s_1\}$
4	(s_1, b, s_2)	$\{s_1\}$	$\{s_2\}$	$\{(s_1, c, s_1), (s_1, b, s_2)\} \cup \{(s_1, b, s_2)\}$	$\{s_1, s_2\} \cup \{s_1, s_2\}$
5	(s_2, d, s_3)	$\{s_2\}$	$\{s_5\}$	$\{(s_1, c, s_1), (s_1, b, s_2)\} \cup \{(s_2, d, s_5)\}$	$\{s_1, s_2\} \cup \{s_2, s_5\}$
6	(s_3, a, s_4)	$\{s_5\}$	$\{s_5\}$	$\{(s_1, c, s_1), (s_1, b, s_2), (s_2, d, s_5)\} \cup \{(s_5, a, s_5)\}$	$\{s_1, s_2, s_5\} \cup \{s_5, s_5\}$
7	(s_4, a, s_3)	$\{s_5\}$	$\{s_5\}$	$\{(s_1, c, s_1), (s_1, b, s_2), (s_2, d, s_5), (s_5, a, s_5)\} \cup \{(s_5, a, s_5)\}$	$\{s_1, s_2, s_5\} \cup \{s_5, s_5\}$
8	(s_4, a, s_5)	$\{s_5\}$	$\{s_5\}$	$\{(s_1, c, s_1), (s_1, b, s_2), (s_2, d, s_5), (s_5, a, s_5)\} \cup \{(s_5, a, s_5)\}$	$\{s_1, s_2, s_5\} \cup \{s_5, s_5\}$
9	(s_5, a, s_5)	$\{s_5\}$	$\{s_5\}$	$\{(s_1, c, s_1), (s_1, b, s_2), (s_2, d, s_5), (s_5, a, s_5)\} \cup \{(s_5, a, s_5)\}$	$\{s_1, s_2, s_5\} \cup \{s_5, s_5\}$

Table 5.1: Execution of step 2 in Algorithm 5.2 with LTS T as input

$T_C = (\texttt{Proc'}, \texttt{Act}, \rightarrow_C)$, where $\texttt{Proc'} = \{s_1, s_2, s_5\}$, $\texttt{Act} = \{a, b, c, d\}$, and

$\rightarrow_C = \{(s_1, c, s_1), (s_1, b, s_2), (s_2, d, s_5), (s_5, a, s_5)\}$. Figure 5.8 illustrates the collapsed LTS T_C.

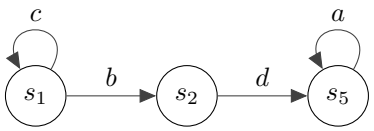


Figure 5.8: Illustration of strong bisimulation collapsed LTS T_C

Example 5.5 — *Weak Bisimulation Collapse*

We use the same example as in Example 5.4 Let $\texttt{Proc} = \{s, s_1, s_2, s_3, s_4, s_5\}$ of processes, where $s \sim s_1$, and $s_3 \sim s_4 \sim s_5$. An illustration of the LTS T is shown on Figure 5.9.

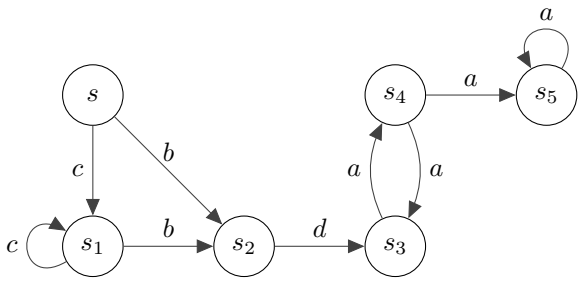


Figure 5.9: Illustration of LTS T.

Say we are only interested in the d-transitions, meaning that we do not want the a, b, c-transitions visualised in the LTS. To achieve this we relabel the a, b, c-transitions to τ for all processes. We create LTS T' under this relabelling, where $p \in \texttt{Proc}$

$$p' = p[tau/a, tau/b, tau/c],$$

LTS T' is illustrated Figure 5.10

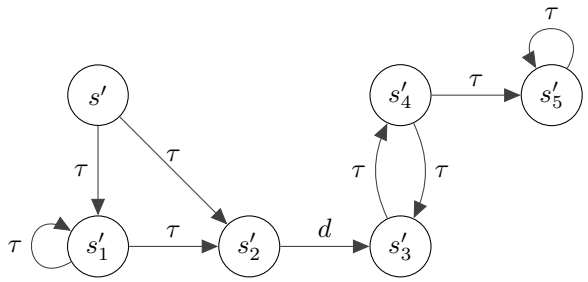


Figure 5.10: Illustration of LTS T', relabelled a, b, c-transition to τ.

We use Algorithm 5.2 to collapse the LTS T' to T'_C, for weak bisimulation, see Figure 5.11 for the collapsed LTS T'_C.

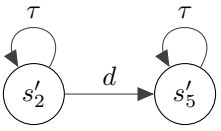


Figure 5.11: Illustration of weak bisimulation collapsed LTS T'_C

We have now simplified the LTS T to LTS T'_C, with relabelling and weak bisimulation, to the point where the interesting d-transition is the only visualised transition, other than τ-transitions.

CWB uses a partition refinement algorithm [5]. While our equivalence collapse algorithm merges disjoint sets together the refine algorithm assumes an opposite stance: all processes are in the same block are equivalent. Initially all processes are in the same block. If two processes in the same block are not equivalent under some criteria the block is split into those that are and those that are not. The time complexity of the algorithm is $\mathcal{O}(|\texttt{Proc}| \cdot |\texttt{Act}|)$ and the space complexity is listed to be $\mathcal{O}(|\texttt{Proc}| + |\texttt{Act}|)$. Compared to our algorithm, this one scales better with the number of processes but uses more memory.

Parallel Fixed-Point Computation

6

Caal only runs in the browser and it mostly only uses a single thread of computation. Leveraging the multiple cores in computers is an interesting challenge. In this chapter we describe a parallel algorithm for computing minimum fixed-points and how existing code is leveraged and combined with a Redis [1] database.

6.1 Overview

The parallel algorithm will only implement a minimum fixed-point algorithm adapted from the one by Liu and Smolka in [12], and will initially only compute strong and weak bisimulation. Several processes will be started that each compute part of the fixed-point computation for the graph.

For the implementation of the parallel fixed-point algorithm we have reused the JavaScript (JS) (compiled from TypeScript) implementation in Caal for faster development of the prototype. This enables reuse of all code not related to user interfaces and prevents having to reimplement several features like the successor generator for dependency graphs, parsers, and transformations that simplify processes.

The choice of the Google V8 JavaScript engine [7] to access our libraries in JS resulted in C++ being chosen as the implementation language for the parallel program. V8 allows C++ programs to create JS values, call JS functions, and create callbacks in C++ that can be called from JS. Our use of V8 includes loading CCS processes and getting hyperedges from a successor generator.

To communicate between workers, share fixed-point values, and partition out work we use Redis (REmote DIctionary Server) [1], a key-value database where keys are byte-safe strings and the values can be a variety of data structures. Our instance will manage data used by the algorithm and maintain the current fixed-point assignments. A useful property of Redis is that all commands sent to it are atomic. If multiple commands are required to be executed in sequence without errors it is necessary to use a transaction. For complex transactions it might be simpler to use Lua scripting in Redis. Lua scripts can be loaded into Redis and later executed by clients. Each script runs in isolation and has access to any arguments the client passed.

Since Redis primarily works with strings we augment the JS in Caal to serialize and deserialize processes and hyperedges to strings.

Figure 6.1 shows an overview of the parts described. Each worker is a C++ process started by MPI. A worker wraps its own successor generator and runs its own partial fixed-point computation on data received from the server. When it has processed data it sends updates back to the server.

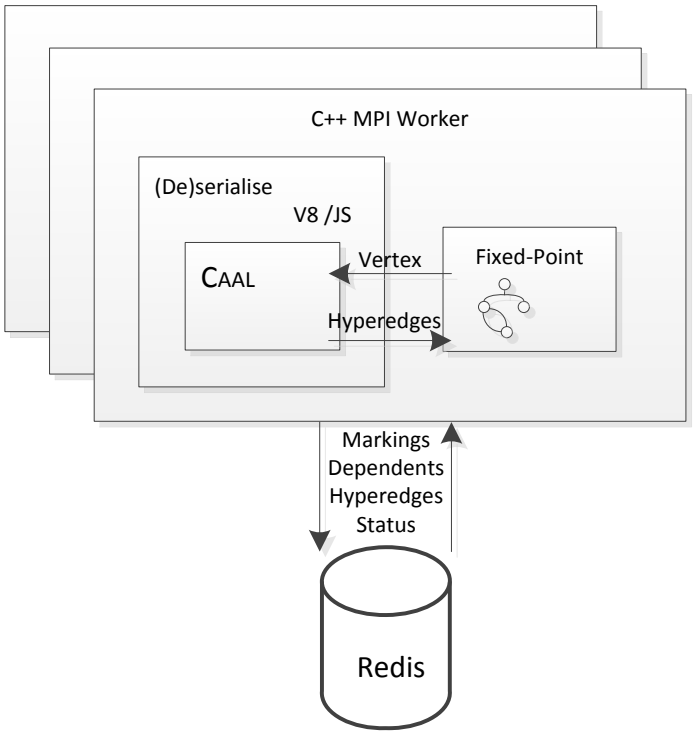


Figure 6.1: Diagram for parallel computation

6.2 Pseudocode

The pseudocode for the algorithm is shown in Algorithm 6.1. All function prefixed with *Atomic* can be assumed to operate on global data. Other data is local. The algorithm is run simultaneous on multiple processes, but each process has its own rank. The process with rank 1 initialises the waiting list by adding the outgoing hyperedges from s_{init} to the waiting list, initializes the set of active workers to the emptyset, and finally joins the other processes at the barrier. No process can cross the barrier until all are ready.

After the first barrier all processes mark themselves as active, and sets their list of dependent edges to the empty set. Dependent edges are hyperedges that needs to be added back to the waiting list when a target node is assigned 1.

The function *AtomicCount* ensures termination when there is no more work. The count only reaches 0 when no more processes have any hyperedges to process and all were unable to acquire any. If only one process is working on any hyperedges and the waiting list is empty the other processes will not exit since the working process may add more work soon. In case s_{init} is assigned 1 all processes are allowed to exit the loop a the next iteration since the result is known (for pseudocode brevity this is not shown).

If the process didn't exit the loop it requests some hyperedges along with assignments for all vertices among those hyperedges in lines 9–14. Then it

Algorithm 6.1 Parallel Minimum-Fixed Point Algorithm

Input: Reference to the database DB, rank of process $1 \leq rank \leq m$, and initial state $s_{init} \in S$.
Output: Change A and D as side effect.

1: **if** $rank = 1$ **then**
2: $\quad AtomicAddEdges(\{succ(s_{init})\})$
3: $\quad Active = \emptyset$
4: $BARRIER$
5: $AtomicPut(Active, rank)$
6: $D = \emptyset$
7: $BARRIER$
8: **while** $AtomicCount(Active) \neq 0$ **do**
9: $\quad H, A_{local} \leftarrow AtomicGetEdgesAndValues(DB)$
10: $\quad$ **if** $H = \emptyset$ **then**
11: $\quad\quad AtomicDel(Active, rank)$
12: $\quad\quad continue$
13: $\quad$ **else**
14: $\quad\quad AtomicPut(Active, rank)$
15: $\quad H_{new} \leftarrow \emptyset$
16: $\quad D_{new} \leftarrow \emptyset$
17: $\quad$ **for** $(s, T) \in H$ **do**
18: $\quad\quad targetsAssignedOne \leftarrow 0$
19: $\quad\quad$ **for** $u \in T$ **do**
20: $\quad\quad\quad$ **if** $A_{local}(u) = \perp$ **then**
21: $\quad\quad\quad\quad A_{local}(u) \leftarrow 0$
22: $\quad\quad\quad\quad D_{new} \leftarrow D_{new} \cup (u, \{(s, T)\})$
23: $\quad\quad\quad\quad H_{new} \leftarrow H_{new} \cup succ(u)$
24: $\quad\quad\quad$ **else if** $A_{local}(u) = 0$ **then**
25: $\quad\quad\quad\quad D_{new} \leftarrow D_{new} \cup (u, \{(s, T)\})$
26: $\quad\quad\quad$ **else**
27: $\quad\quad\quad\quad targetsAssignedOne \leftarrow targetsAssignedOne + 1$
28: $\quad\quad$ **if** $targetsAssignedOne = |T|$ **then**
29: $\quad\quad\quad A_{local}(s) \leftarrow 1$
30: $\quad BEGIN(DB)$
31: $\quad AtomicAddEdges(H_{new})$
32: $\quad AtomicAddDependents(D_{new} \setminus D)$
33: $\quad AtomicUpdateAssignments(A_{local})$
34: $\quad END(DB)$
35: $\quad D \leftarrow D \cup D_{new}$

initializes the local variables H_{new} and D_{new} to hold new hyperedges and dependents respectively.

All hyperedges are iterated over starting at lines 17. For each hyperedge it iterates the target vertices. If the target vertex has not been seen before, in line 20, it is given the new marking of 0, is added as a dependent for s, and its successors are added to H_{new} to later be added to the global waiting list. Otherwise if the vertex is assigned 0 we simply add the hyperedge as a new dependent. Finally, if it is assigned 1, we simply increment the number of target vertices assigned 1 so far. If all target vertices of a hyperedge are assigned 1, then in order for the assignment to be a valid prefixed-point assignment the source vertex must also be assigned 1, done in line 29. This is where a check for early termination could be made.

Finally the data needs to be sent back to the database using a transaction. The set of new hyperedges are added to the database, followed by any new dependent edges, and finally the updated assignment. The database adds dependent hyperedges to the waiting list internally using a Lua script for any new assignment of 1.

6.3 Parallel Fixed-Point Computation Results

The university has a compute cluster for running parallel programs. The cluster consists of nine compute nodes each with 1Tb of memory, 4 AMD Opteron 6376 Processors each containing 16 cores running at 2.3Ghz, and a 1Tb disk [17].

Our parallel algorithm was run on the compute nodes using up to 63 cores. The last core was reserved since the Redis database used by the algorithm ran on the same compute node. The algorithm was run on the verification problem known as Alternating Bit Protocol (ABP). The problem can be tuned for different buffer sizes with larger sizes yielding a more complex verification problem. The dependency graph constructed checks for weak bisimilarity between the implementation and specification of ABP.

Table 6.1 shows information about each run. Single-core verification time is how much time the algorithm spent using a single process. The entries below the number of cores is the relative speed compared to single core. For instance the run with buffer size 4 and 32 processes had a relative speed of 2.14 – slightly more than twice as fast as when run with a single process.

Buffer size	Single-worker-process verification time (s)	Number of worker processes						
		2	4	8	16	24	32	63
2	3.24	1.14	1.24	1.32	1.15	1.17	1.18	0.99
3	19.10	1.30	1.39	1.34	1.51	1.76	1.86	1.49
4	128.10	1.28	1.66	1.85	1.92	2.15	2.14	1.85
5	838.97	1.33	1.66	1.87	1.79	1.78	1.73	2.19

Table 6.1: Verification time

It was attempted to run ABP with a buffer size of 6, however the program crashed after roughly 30 minutes. The successor generators in CAAL was meant

for educational aspects and for running in the browser. This means the successor generators are not completely state-free and internally cache results. The cached data accumulates over many calls and eventually the memory limit in V8 is reached [15].

From the table it is apparent the relative speeds when running multiple processes are not satisfactorily. It would be reasonable to expect the speedup when running 32 processes with available cores to be significantly higher than 2.

To better understand why it was not faster we examine some data logged during the run with 32 processes for buffer size 4 compared to the run with just one process shown in Table 6.2:

Logged data item	1 process	32 processes
Avg. system time reading [ms]	1081	4403
Avg. system time writing [ms]	921	4811
Avg. system time loading [ms]	214	45
Avg. system time processing [ms]	125391	48437
Sum of the above [ms]	127608	57698
Avg. system time spent in JS [ms]	125193	48371
Total num edges processed	2657	16021
Total num vertices processed	69382	509686
Total MB Written	87	567
Total MB Read	87	567
Total iterations	890	12799
Total wait iterations	1	7384
Total work iterations	889	5415
Total running time [s]	128.10	58.87

Table 6.2: Data for ABP run with 32 processes and buffer size 4.

For our runs we know the dependency graph has to be fully explored. The number of wait iterations indicate the number of iterations a process attempted to received hyperedges to work on, but none were available. This number is 1 for one process and this happens when there are no more hyperedges left to process. For 32 processes the number of iteration not doing anything is greater. However, waiting does not seem to be the major contributing factor to the poor speedup: except reading and writing to Redis, almost all of the remaining time is spent running JavaScript code as indicated by the column for JS Time.

The number of edges and vertices listed in the column for one process is therefore a reasonable approximation of the complexity of the graph. With this realization it becomes apparent by comparing the total number of hyperedges and vertices processed that the 32 processes must have worked on overlapping vertices. The run with 32 processes go through roughly 7 times more work indicated by number of edges, nodes, data written and read, and number of work iterations. This might have been caused by processes exploring and both adding the same newly-found hyperedges to the waiting list. We hypothesize that redundant work loads are the major reason for the poor speedup.

It should also be noted that the serialized string representation of vertices can become many kilobytes and hyperedges can reach megabytes. The overhead of serializing and deserialising vertices in JS are far from negligible. Running one process for ABP with buffer size 4 takes around 125s in the parallel algorithm. Doing the same in CAAL takes around 35s. The only difference is the serialization and deserialisation.

```
*** Send
Send_0 = accept.Send_out_0;
Send_1 = accept.Send_out_1;

Send_out_0 = 'send_0.Send_wait_0;
Send_out_1 = 'send_1.Send_wait_1;

Send_wait_0 = Send_out_0 + dack_0.Send_1 + dack_1.Send_wait_0;
Send_wait_1 = Send_out_1 + dack_1.Send_0 + dack_0.Send_wait_1;

***  Receive
Receive_0 = dsend_0.Receive_ack_0 + dsend_1.'ack_1.Receive_0;
Receive_1 = dsend_1.Receive_ack_1 + dsend_0.'ack_0.Receive_1;

Receive_ack_0 = 'deliver.'ack_0.Receive_1;
Receive_ack_1 = 'deliver.'ack_1.Receive_0;

*** Medium (lossy)
Mediuml_send = send_0.Mediuml_send_0 + send_1.Mediuml_send_1;
Mediuml_send_0 = 'dsend_0.Mediuml_send + Mediuml_send;
Mediuml_send_1 = 'dsend_1.Mediuml_send + Mediuml_send;

BufferSend2 = (Mediuml_send[a0/dsend_0, a1/dsend_1] |
               Mediuml_send[a0/send_0, a1/send_1]) \ {a0,a1};
BufferSend3 = (BufferSend2[a0/dsend_0, a1/dsend_1] |
               Mediuml_send[a0/send_0, a1/send_1]) \ {a0,a1};
BufferSend4 = (BufferSend3[a0/dsend_0, a1/dsend_1] |
               Mediuml_send[a0/send_0, a1/send_1]) \ {a0,a1};
BufferSend5 = (BufferSend4[a0/dsend_0, a1/dsend_1] |
               Mediuml_send[a0/send_0, a1/send_1]) \ {a0,a1};

Mediuml_ack = ack_0.Mediuml_ack_0 + ack_1.Mediuml_ack_1;
Mediuml_ack_0 = 'dack_0.Mediuml_ack + Mediuml_ack;
Mediuml_ack_1 = 'dack_1.Mediuml_ack + Mediuml_ack;

*** The alternating bit protocol - Implementation
set Internals = { send_0, send_1, ack_0, ack_1,
                  dsend_0, dsend_1, dack_0, dack_1 };
ABPl = (Send_0|Receive_0|BufferSend5|Mediuml_ack) \ Internals;

*** The alternating bit protocol - Specification
SPEC = accept.'deliver.SPEC;
```

Implementation 7

In the following sections we show the debugging tools implemented in Caal based on theory described in previous chapters. This includes HML game, distinguishing formula, and bisimulation collapse. We do this by showing examples and describe how these debugging tools might aid an user in understanding model checking, or the behavioural aspects of a process. Caal is implemented in JavaScript and is licensed under the MIT License. In the previous report [14] we chose to implement Caal as a web application using JavaScript, which resulted in a portable application that can be accessed by different operating systems, such as Windows, Linux, and Mac OS. The benefit of having Caal implemented as a web application is the easy accessibility and no need to for installation. Because Caal aims to be an educational tool, performance was never highly prioritised, since a web application will never have the computation power of a native application. Instead user friendliness and usability as a teaching tool were prioritised when creating Caal.

7.1 HML Game

Our implementation of HML game share some similarities to the bisimulation game done in our previous report [14]. This section describes how we implemented the HML game in Caal according to the theory in Chapter 3

The game consists of an "attacker" and a "defender", a process and a formula. The attacker's goal is to prove that the process does not satisfy the formula, whereas the defender's goal is prove that the process does satisfy the formula. A full description of the rules and winning conditions of an HML game can be found in Section 3.4. In the current state of Caal we only support HML games with a minimum fixed-point, but plan to implement support for maximum fixed-points.

The HML game is designed such that the human player will always play the losing role, meaning the computer always has a winning strategy. In case of an infinite loop, the winner depends on what context the game is in.

Figure 7.1 gives a full overview of the implemented HML game. The bottom left corner has a game log, which gives the user an overview of what configuration the game is in, whose turn it is to make a move and what context the game is in. To the right of the game log there is a table showing the possible subformulae and transitions it is possible for the user to choose.

For the example we use the process S seen in Figure 7.1 and the formula

$$X \overset{min}{=} \langle c \rangle \langle a \rangle tt \vee (\langle b \rangle tt) \wedge [b]X)$$

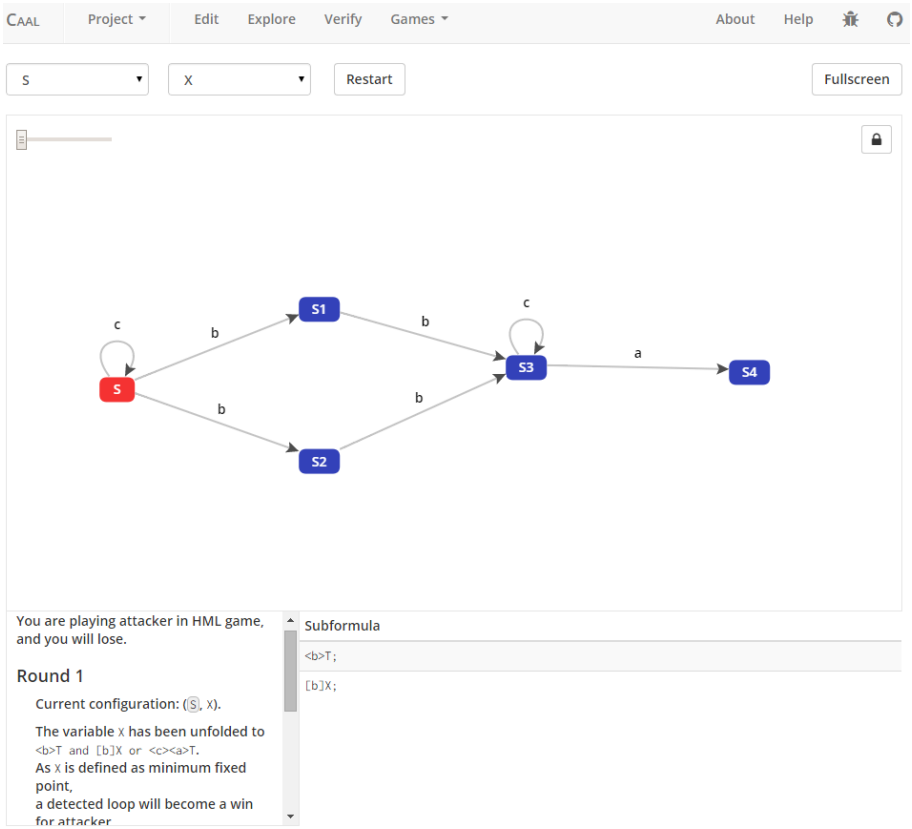


Figure 7.1: Full overview of HML game

Figure 7.2 shows the result of the verification done by CAAL. The formula is satisfied. The user can click the three vertical dots to the right and click "Play" to be convinced of the correctness of the property. The user is then taken to the HML game seen in Figure 7.1

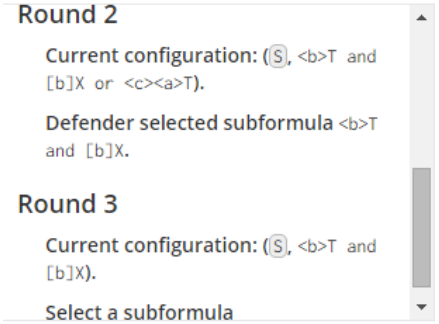
Figure 7.2: Result of the formula verification

Because the formula was satisfied by the process S, we are playing as attacker and we are going to lose.

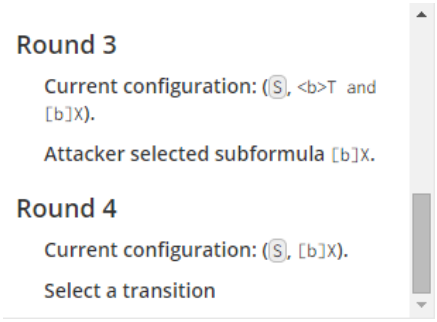
Figure 7.3a shows the defender's first move; it chose the subformula $\langle b\rangle tt \wedge [b]X$. The configuration is now a conjunction meaning it is our turn as attacker to pick. Figure 7.3b shows we picked the subformula $[b]X$ and we have to choose a b-transition to either $S1$ or $S2$.

Regardless of which transition we pick, we end up in a state with only an out-going b-transition, so the defender picks the configuration $\langle b\rangle tt \wedge [b]X$ again and we respond by choosing $[b]X$ and we end up in $S3$.

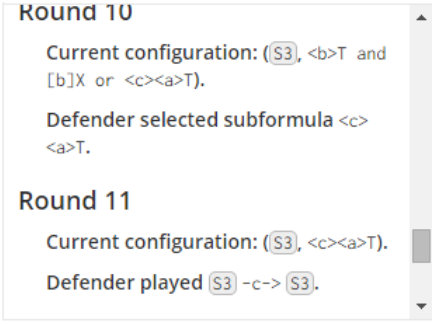
Figure 7.3c shows that this time the defender picks $\langle c\rangle\langle a\rangle tt$ and afterwards pick the c-transition to $S3$. The defender follows up by picking the a-transition and reaches the *true* formula and therefore win as seen on Figure 7.3d.



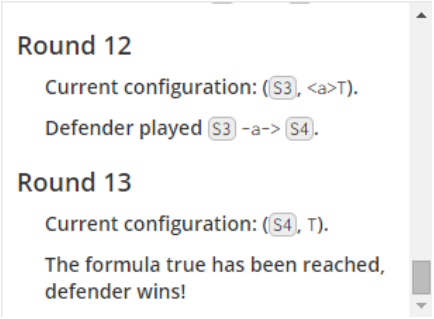
(a) Defender picks subformula



(b) Attacker picks subformula



(c) Defender picks subformula



(d) Defender wins

Figure 7.3: Game log for HML game

7.2 Distinguishing Formula

For two non-bisimilar processes it can be useful to know a property that one satisfies but not the other. This property can be described by a distinguishing formula. Although these formulae are not always easy to find by hand, they are guaranteed to exist. This section shows our implementation in CAAL, based on the theory in Chapter 4, for generating distinguishing formulae.

For our example we define the following processes

$$Man \stackrel{def}{=} wakeUp.shower.\overline{start}.cook.\overline{stop}.eat.Man$$
$$Stove \stackrel{def}{=} start.Stove1$$
$$Stove1 \stackrel{def}{=} \overline{cook}.stop.Stove + break.0$$
$$Dinner \stackrel{def}{=} (Man \mid Stove)\backslash\{start, stop, cook\}$$
$$Spec \stackrel{def}{=} wakeUp.shower.tau.tau.tau.eat.Spec$$

We can use the explorer to visualise our CCS program, on Figure 7.4 we explore the process *Dinner*.

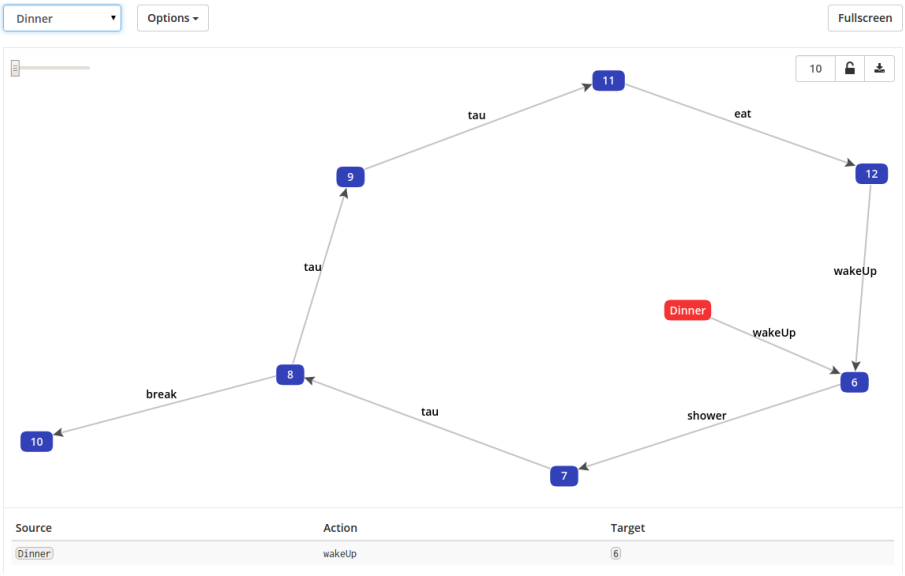


Figure 7.4: LTS of the process *Dinner*.

Our question is then, does $Dinner \sim Spec$? Is the man able to wake up, take a shower, cook his meal on the stove and thereafter eat it? To answer our questions we go to the verifier, and add the strong bisimulation property for *Dinner* and *Spec*. See Figure 7.5 for the verified property.

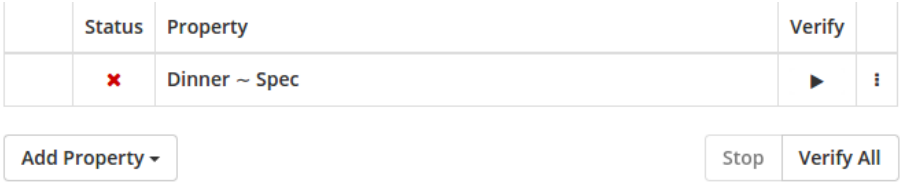


Figure 7.5: Verified strong bisimulation property.

So by looking at Figure 7.5 we can see that the property is not satisfied. We can now generate a distinguishing formula to show what exactly separates the *Dinner* and *Spec* process.

Figure 7.6: Generated distinguishing formula.

On Figure 7.6 two new HML properties have now been generated with the distinguishing formula

$$\langle wakeUp \rangle \langle shower \rangle \langle tau \rangle \langle break \rangle tt$$

. And we can see that

$$Dinner \models \langle wakeUp \rangle \langle shower \rangle \langle tau \rangle \langle break \rangle tt$$
$$Spec \not\models \langle wakeUp \rangle \langle shower \rangle \langle tau \rangle \langle break \rangle tt$$

If we need more convincing that e.g. *Dinner* satisfies the distinguishing formula we can play the HML game, which is explained in Section 7.1.

7.3 Equivalence Collapse

In this section we show how the equivalence collapse works in CAAL. We give an example to show that it is possible to collapse an LTS using strong and weak bisimulation. We will also show that with a combination of relabelling and weak collapse, it is possible to hide uninteresting process behaviour for certain transitions, and thereby highlight other transitions.

In Figure 7.7 an LTS is presented in CAAL's debugging tool explorer. In the explorer we have the possibility to collapse the LTS using either strong or weak bisimulation collapse. The equivalence collapse has been implemented according to the theory presented in Chapter 5

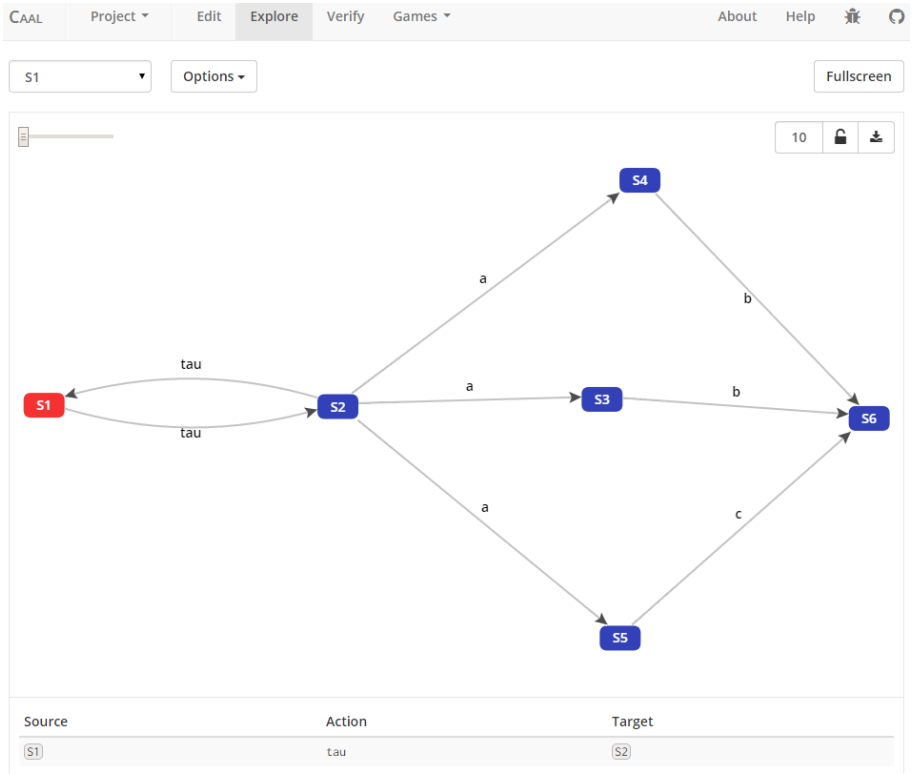


Figure 7.7: Uncollapsed LTS

Strong Bisimulation Collapse

If we press the "Options" dropdown menu, and select strong bisimulation collapse we get the graph shown in Figure 7.8.

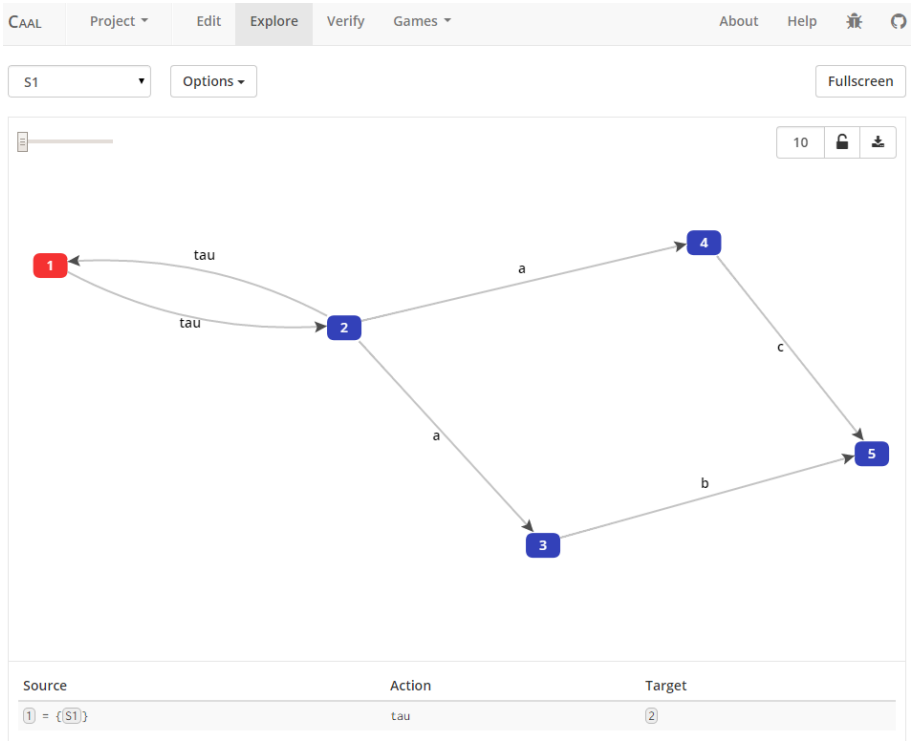


Figure 7.8: Collapsed LTS, with strong bisimulation

We see that the sequence $S2 \xrightarrow{a} S4 \xrightarrow{b}$ S6 and $S2 \xrightarrow{a} S3 \xrightarrow{b} S6$ has been collapsed to $6 \xrightarrow{a} 4 \xrightarrow{b} 1$. It has been collapsed since $S3 \sim S4$.

Weak Bisimulation Collapse

We can also use weak bisimulation collapse to abstract from τ-transitions, and further simplify the LTS. Notice that the same processes has been collapsed from the strong bisimulation collapse figure, because if two processes are strongly bisimilar they are also weakly bisimilar. Process $S1$ and $S2$ are weakly bisimilar and therefore are they collapsed into process 2, see Figure 7.9.

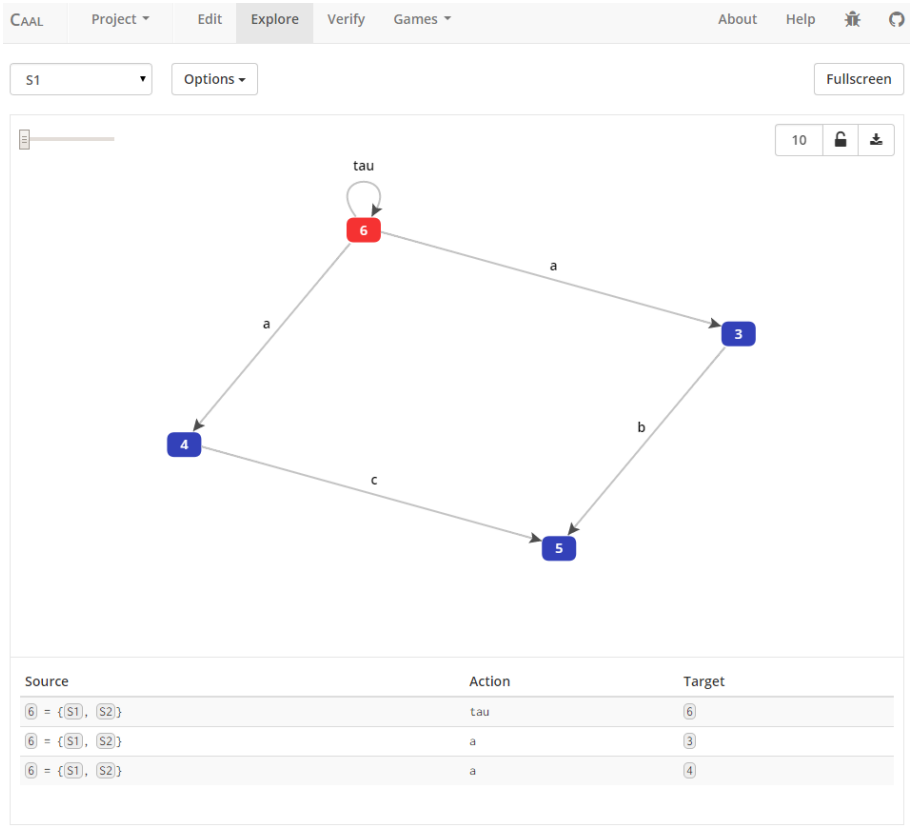


Figure 7.9: Collapsed LTS, with weak bisimulation

Together with relabelling to τ and weak bisimulation collapse, we can hide unimportant processes behaviour. Say we define $S1$ to be:

$$S1 = (\tau.S2)[\tau/a, \tau/c];.$$

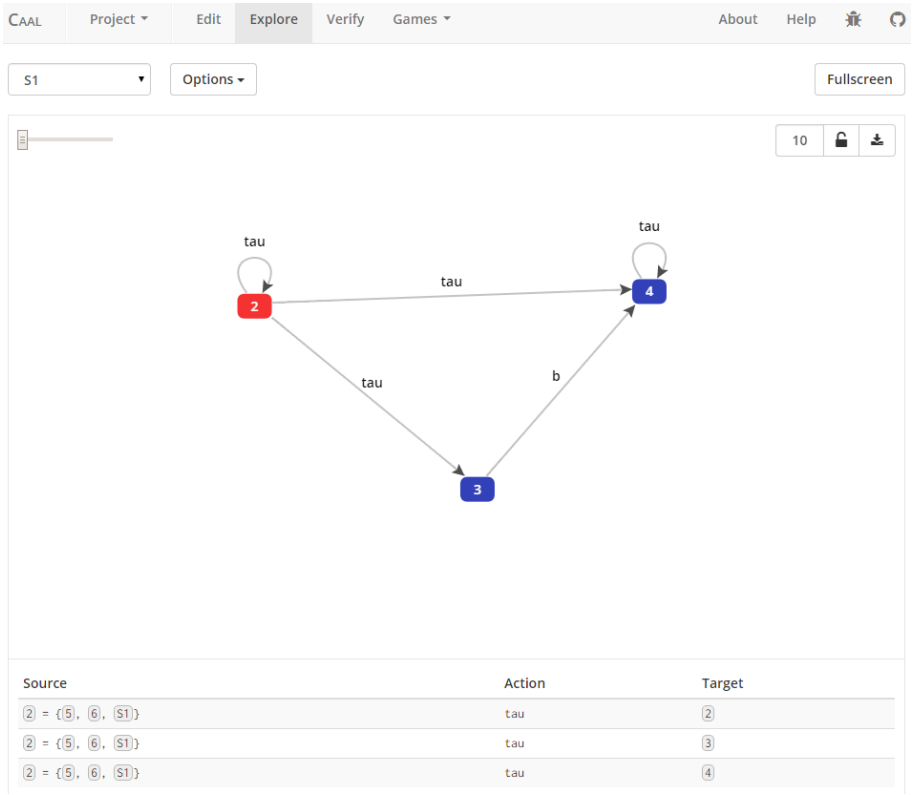


Figure 7.10: LTS after relabelling and weak bisimulation collapse.

If we explore $S1$ we can see that a, c-transitions for process $S1$ has changed to τ-transitions, and with weak bisimulation collapse enabled we get Figure 7.10. This feature can be beneficial if you have a large LTS and you are only interested the behaviour for a process for certain actions.

CAAL *Tutorial*

8

This tutorial gives an informal introduction to the main features of CAAL and how to use them. CAAL supports the process algebras Calculus of Communicating Systems (CCS) and Timed Calculus of Communicating Systems (TCCS), and both equivalence and model checking analysis of processes through verification and games. CAAL consists of four different modules; an editor module for modelling processes, a module for visualization of processes, a module for equivalence and model checking, and a game module. CAAL is available at:

`http://caal.cs.aau.dk`.

8.1 The Language CCS

CCS is a process algebra used to model concurrent systems. We shall now informally introduce CCS.

The most basic process of all is the 0 (or *nil*) process. It cannot perform any action whatsoever and thus stops all computation. The most basic process constructor in CCS is *action prefixing*. The formation rule for action prefixing is as follows:

If P is a process and a is a label, then $a.P$ is a process.

Using the formation rule for action prefixing and the 0 process we can build two example processes:

$$\texttt{shake}.0 \quad \texttt{shake}.\texttt{walk}.0 \ .$$

The first process can only perform the shake action and then *dies* (becomes the 0 process). The second process is a more complex process, which after performing the shake action, can also perform the walk action. Names can also be assigned to processes. For example, we can give the second process a name:

$$\texttt{Boy} = \texttt{shake}.\texttt{walk}.0 \ . \tag{8.1}$$

Naming processes allows us to introduce recursive definitions of process behaviors. For example, we can define a recursive process specification as follows:

$$\texttt{Tree} = \texttt{shake}.\overline{\texttt{'apple}}.\texttt{Tree} \ .$$

This tree can be shaken which causes it to deliver an apple and afterwards returns to its initial state where it can be shaken again. Note the bar over $\overline{\texttt{'apple}}$ (the apostrophe denotes a bar in CAAL), which indicates that it is an output action. This tree only allows one type of apple. In order for the tree to support multiple colors of apples, we use the *choice operator*. Now the tree can be defined as:

$$\texttt{ColorTree} = \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{ColorTree} + \overline{\texttt{'redapple}}.\texttt{ColorTree}) \ . \tag{8.2}$$

The idea is that after the tree has been shaken, it can deliver either a green or a red apple. In general, the formation rule for choice is:

If P and Q are processes, then $P + Q$ is also a process.

The process $P + Q$ is able to do either P or Q, but not both. As soon as P is performed any further execution of Q is preempted and vice versa.

Another operator is the *parallel composition operation*. Composition describes two or more processes running in parallel and possibly interacting with each other. For example, if we continue the example from Equation 8.1, we can shake the tree in order to receive an apple and then walk to the next tree after an apple has fallen to the ground. This can be described by the CCS process:

$$\texttt{Girl} = \overline{\texttt{'shake}}.\texttt{apple}.\overline{\texttt{'walk}}.\texttt{Girl} \ . \tag{8.3}$$

The CCS expression `Tree | Girl` describes a system consisting of two processes; the tree and the girl. These two processes can communicate through their shared communication channels; shake and apple.

However, neither the girl nor tree are required to communicate with each other. They could communicate over their channels with any other processes they have been composed with, or simply perform the shake, apple, or $\overline{\text{walk}}$ actions directly without communication.

P and Q may proceed independently or they may communicate through shared channels.

When two processes communicate through the same input and output action the resulting action is called a τ-action. It might be best if only one had access to the apples that fall from the tree. CCS allows this through an operation called *restriction*. This allows us to hide certain channels from the environment. If we continue from Equation 8.3 and expand the example to accept red or green apples:

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{Man1 + Man2}) \ , \\ \texttt{Man1} &= \texttt{redapple}.\overline{\texttt{'walk}}.\texttt{Man} \ , \\ \texttt{Man2} &= \texttt{greenapple}.\overline{\texttt{'throw}}.\texttt{Man} \ . \end{aligned} \tag{8.4}$$

Now we can define the Orchard using the ColorTree from Equation 8.2 and the refined Man from Equation 8.4:

$$\texttt{Orchard} = (\texttt{ColorTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ . \tag{8.5}$$

The restricted channels shake, redapple, and greenapple may only be used for communication between the tree and the man. Their scope is restricted to the process Orchard. In general, the formation rule for restriction can be described as follows:

If P is a process and L is set of channel names, then $P \setminus L$ is a process.

In $P \setminus L$ the channel names in L can only be used to communicate within P. It might be beneficial for the orchard to have access to other sorts of fruit. This can be done by defining a generic orchard that can be shaken, then drop its fruit and reset:

$$\texttt{GenericOrchard} = \texttt{shake}.\overline{\texttt{'fruit}}.\texttt{GenericOrchard} \ .$$

Through appropriate renaming of the GenericOrchard it is possible to obtain a more specific Orchard. For example:

$$\texttt{PearOrchard} = \texttt{GenericOrchard}\ [\texttt{pear/fruit}]\ .$$

PearOrchard is a process that behaves like GenericOrchard but outputs pears instead of a generic fruit. The renaming operation can be described as:

If P is a process and f is a function from labels to labels, then $P[f]$ is a process.

8.2 The Language TCCS

TCCS is an extension of CCS with time, which means that we still have all the syntactical elements of CCS but with a new syntactic element, the *delay prefixing operator*. With this operator we can model processes like

$$5.\texttt{a}.0\ ,$$

which means that after delaying for 5 time units the a-action becomes available.

We extend the Orchard example and add time to it. We add a time constraint to the tree specifying that if the falling apple has not been caught within 3 time units then it falls to the ground. Extending the ColorTree from Equation 8.2 we get:

$$\texttt{ColorTree} = \texttt{shake}.\texttt{ColorTree1}\ ,$$
$$\texttt{ColorTree1} = \overline{\texttt{'greenapple}}.\texttt{ColorTree} + \overline{\texttt{'redapple}}.\texttt{ColorTree} + 3.\texttt{tau}.\texttt{ColorTree}\ .$$

The ColorTree has the choice of dropping either a green or a red apple. If the tree drops a particular apple then it commits to that choice, but simply delaying will not commit to any choice. For example, after delaying for 2 time units the tree can still drop green or red apples.

However, after 3 time units the τ-action becomes available which prevents any further delays. An action must be performed immediately when a τ-action is available. If no one is ready to catch the apple within 3 time units the apple falls to the ground.

Let us say that after the man has shaken the tree he needs to rest for 2 time units before he is ready to catch an apple. Extending the Man from Equation 8.4 we get:

$$\texttt{Man} = \overline{\texttt{'shake}}.2.(\texttt{Man1 + Man2})\ .$$

It is not possible for the man to rest for more than 2 time units because a handshake is available between the Man and the ColorTree (i.e. a τ-action becomes available). We can also define an unfit man who requires a longer break after shaking the tree:

$$\texttt{SlowMan} = \overline{\texttt{'shake}}.5.(\texttt{Man1 + Man2})\ .$$

If we define the orchard as

$$\texttt{Orchard} = (\texttt{ColorTree} \mid \texttt{SlowMan}) \setminus \{\texttt{shake, redapple, greenapple}\}\ .$$

then the slow man will never be able to catch an apple since his required break makes him unable to catch the apples before they fall to the ground.

8.3 Editor

The editor is used to input CCS and TCCS programs. The editor has full support for CCS and TCCS syntax, and features live syntax checking to assist the user if syntactical errors occur. The "Parse"-button will notify the user of any contextual errors, such as referencing an undefined process. Furthermore, CAAL supports saving of the project to both a local file and the browser cache, as well as an autosave feature that allows the user to restore unsaved work if an unexpected error should occur. Using the editor we can input the examples from Equation 8.2 and Equation 8.4 as shown in Figure 8.1.

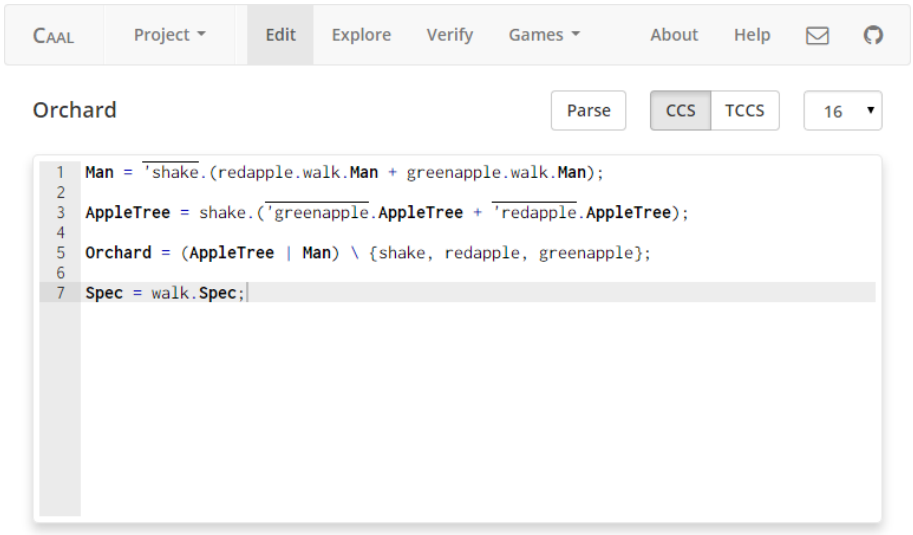


Figure 8.1: Editor.

8.4 Verification

After having defined a process in the editor we may want to verify its correctness. We introduce the several forms of verification that CAAL supports.

Equivalence Checking

CAAL supports the following equivalences and preorders:

- simulation,
- simulation equivalence,
- bisimulation,
- trace inclusion, and
- trace equivalence.

This section focuses on bisimulation. Strong bisimulation is a notion relating two processes such that whenever one of the processes can perform an α-action the other process must also be able to perform an α-action. The resulting pair must again be related by strong bisimulation.

We also have the notion of weak bisimulation. We use the term "weak" to indicate that we abstract away from τ-actions. Whenever one of the processes can perform an α-action the other process also must be able to perform a matching α-action, where it is allowed to perform zero or more τ-actions before and after performing the α-action. The resulting pair must again be related by weak bisimulation.

Example 8.1

We have the CCS processes:

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man} + \texttt{greenapple.walk.Man}) \ , \\ \texttt{AppleTree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{AppleTree} + \overline{\texttt{'redapple}}.\texttt{AppleTree}) \ , \\ \texttt{Orchard} &= (\texttt{AppleTree} \mid \texttt{Man}) \ \backslash \{\texttt{shake, redapple, greenapple}\} \ , \\ \texttt{Spec} &= \texttt{walk.Spec} \ , \end{aligned}$$

We want to check if the processes Orchard and Spec are strongly or weakly bisimilar. Figure 8.2 shows the result of the verification. The processes Orchard and Spec are not strongly bisimilar, but they are weakly bisimilar, as indicated by the red cross and the green check mark, respectively.

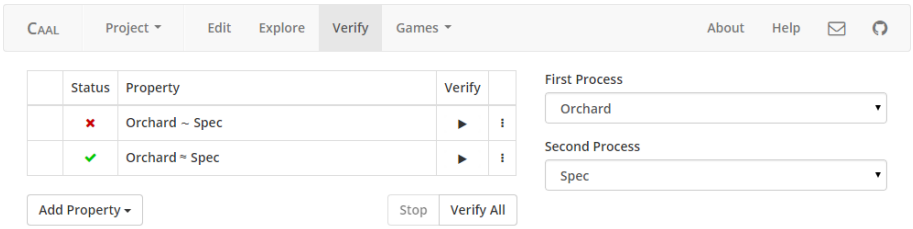


Figure 8.2: Verification of bisimulation.

Model Checking

CAAL supports model checking through use of recursive Hennessy-Milner logic (HML) formulas. HML formulas are used to check if a given process satisfies certain properties. For instance we might want to check if our man:

- is always able to walk after receiving an apple,
- is able to shake the tree right now,
- is able to get hold of a red apple.

CAAL has support for the full syntax and semantics of recursive HML, and also supports formulas with multiple nested variables, with the restriction that variables are not allowed to be mutually recursive.

```
X min= <a>X or Y
Y max= [b]Y
```
(8.6)

Equation 8.6 is an example of a supported HML formula.

```
X min= <a>X or Y
Y max= [b]Y and X
```
(8.7)

Equation 8.7 is an example of an HML formula that is not allowed because Y refers back to X.

Example 8.2

We have the CCS processes

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{AppleTree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{AppleTree + } \overline{\texttt{'redapple}}.\texttt{AppleTree}) \ , \\ \texttt{Orchard} &= (\texttt{AppleTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ . \end{aligned}$$

We want to check if it is possible to reach a state from the Orchard where the Man will never be able to perform a *walk*-transition again. We can express this as the recursively defined property

```
X min= [[walk]]ff or <->X,
```

where `-` is the set of all actions. Figure 8.3 shows the result of the verification. As we can see, this property is not satisfied, as indicated by the red cross.

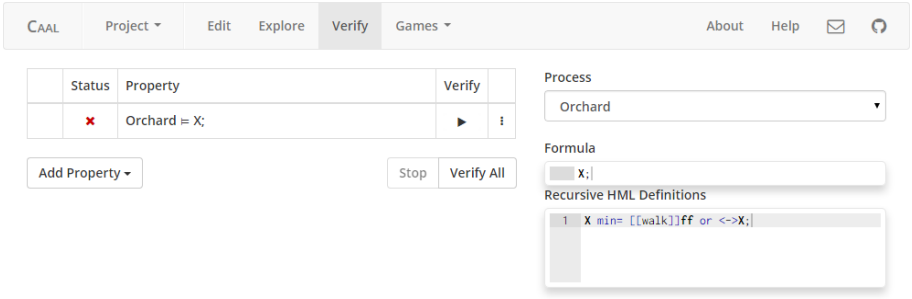


Figure 8.3: Verification of a recursive HML formula.

Timed Equivalence and Model Checking

When verifying TCCS we support the same equivalences and preorders as mentioned earlier, as well as an extended version of HML with time called Timed HML (THML). Strong timed bisimulation is almost the same as regular strong bisimulation. Whenever a process can make a move by some action α, the other process must be able to match the move by the same action α. Whenever a process can make a delay, the other process must be able to match the delay. The resulting pairs must again be related by strong timed bisimulation.

Just like it can be useful to abstract away from τ in CCS, it can be useful to abstract away from time in TCCS, which is called "untimed".

Example 8.3

We have the TCCS processes:

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.2.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{Tree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{Tree + } \overline{\texttt{'redapple}}.\texttt{Tree + 3.tau.Tree}) \ , \\ \texttt{Orchard} &= (\texttt{Tree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ , \\ \texttt{Spec} &= \texttt{walk.Spec} \ , \end{aligned}$$

We want to check if the Orchard is weakly timed or weakly untimed bisimilar to Spec. The Orchard is not weakly timed bisimilar to Spec, but they are weakly untimed bisimilar shown in Figure 8.4.

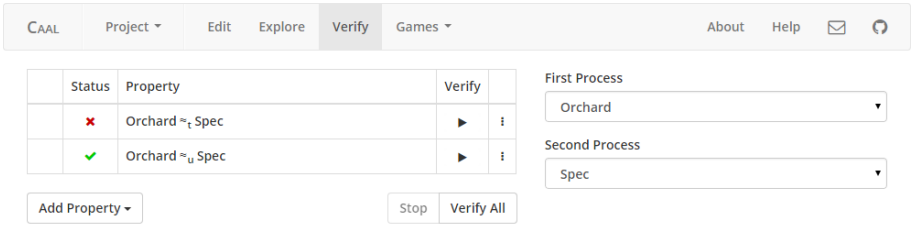


Figure 8.4: Verification of bisimulation with time.

As seen in Example 8.2, HML allows us to verify that the system satisfies certain properties, but it is often interesting to verify that the system does so with respect to time.

Example 8.4

We want to verify that the Man from Example 8.3 can never receive a red apple if he waits for less than 2 time units after shaking the tree. We can express this property as the recursively defined THML formula:

```
X max= ['shake]<0,1>[redapple]ff and X;
```

As seen in Figure 8.5 the property is satisfied.

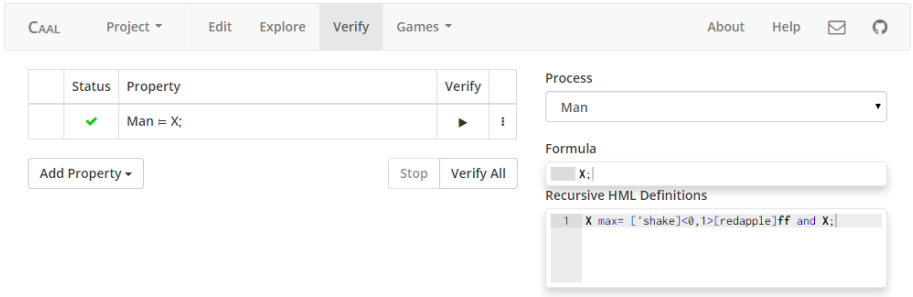


Figure 8.5: Verification of a recursive THML formula.

8.5 Debugging Options

Verifying properties for CCS processes might not always yield the expected result. This might mean a bug is present in the CCS processes. We introduce the tools available for debugging in CAAL.

Explorer

The explorer makes it possible to graphically explore the Labelled Transition System (LTS) generated by a process. To begin, the desired process is selected from the drop-down menu at the top left. The outgoing transitions from the selected process are then displayed. The explorer is shown in Figure 8.6.

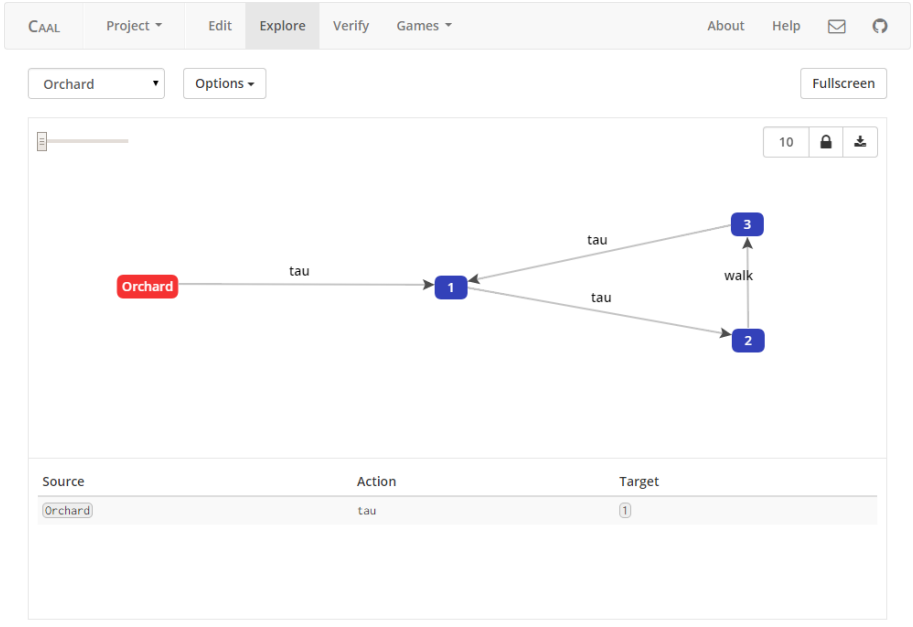


Figure 8.6: Explorer.

The states in the LTS can be selected by clicking them. The currently selected state is colored red and the outgoing transitions from that state will be displayed in the table below the LTS.

A number of different options are available in the explorer:

Zoom The slider at the top left will zoom in on the currently selected state. Sometimes the LTS becomes too large to tell the different states and transitions apart, which is when zooming helps. When zoomed in, the LTS will automatically be centered on the currently selected state whenever it is changed.

Expand Depth The number at the top right is the number of states to expand the LTS with. For example, if we have a depth of five, then all states which are up to five transitions away from the currently selected state will be displayed.

Lock The padlock at the top right will lock/unlock the LTS. The states in the LTS are automatically positioned, but may sometimes become cluttered if there are too many states or transitions. Locking the LTS makes it possible to manually rearrange the states in the LTS.

Export The download button at the top right will download an image of the currently displayed LTS.

Transitions The LTS can be displayed using either strong or weak transitions. By default the LTS displayed is using strong transitions. In the case of TCCS, there are also options for timed and untimed transitions.

Collapse The LTS can be collapsed using either strong or weak bisimulation collapse. Strong bisimulation collapse means that all states which are strongly bisimilar are collapsed into a single state. Figure 8.7 shows the Orchard with strong bisimulation collapse, and Figure 8.8 shows the Orchard with weak bisimulation collapse. In cases where the LTS becomes very large the zoom option might not be sufficient. In such cases all unwanted actions can be relabelled to τ and removed using weak bisimulation collapse.

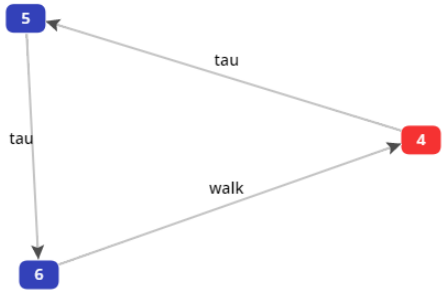


Figure 8.7: Orchard with strong bisimulation collapse.

Figure 8.8: Orchard with weak bisimulation collapse.

Games for Equivalences and Preorders

CAAL supports games for the following equivalences and preorders:

- strong/weak bisimulation,
- strong/weak simulation, and
- strong/weak simulation equivalence.

Furthermore, CAAL also has games for the timed and untimed variations of the above equivalences and preorders. In this section we will focus on the game for strong bisimulation. The games for the other equivalences and preorders are similar, but with different rules.

The strong bisimulation game consists of an "attacker", a "defender", and two processes s and t to play on. The goal of the attacker is to show that the processes are not strongly bisimilar, and the goal of the defender is to show that they are. The game is played over a number of rounds, where each round starts in a pair of states called the current configuration. Initially, the current configuration will be (s, t). Each round is played according to the following rules:

1. The attacker performs a transition under some action α from s to s' or from t to t'. If the attacker cannot perform any transition the defender wins.
2. The defender must now respond with a transition.
   - If the attacker played s to s', then the defender must perform a transition t to t' under the same action α. If the defender cannot perform any transitions, then the attacker wins.
   - If the attacker played t to t', then the defender must perform a transition s to s' under the same action α. If the defender cannot perform any transitions, then the attacker wins.
3. The game continues for another round with the pair (s', t') as the current configuration.

If a cycle is detected in the game, i.e. if we reach a configuration (s', t') which has previously been the current configuration the defender wins the game.

If the attacker has a universal winning strategy, then s and t are not strongly bisimilar. If the defender has a universal winning strategy, then s and t are strongly bisimilar. If a player has a universal strategy, then that player will always be able to win regardless of what the other player does.

We show an example of a strong bisimulation game. Instead of showing the simple game between the Orchard and Spec processes, we will define a pear tree to play against the apple tree. We can define the pear tree as a relabelling of the ColorTree from Equation 8.2:

$$\texttt{PearTree} = \texttt{ColorTree}\ [\texttt{pear/greenapple}, \texttt{pear/redapple}]\ .$$

Figure 8.9 shows a strong bisimulation game where the player is playing as attacker against the computer in the defender role.

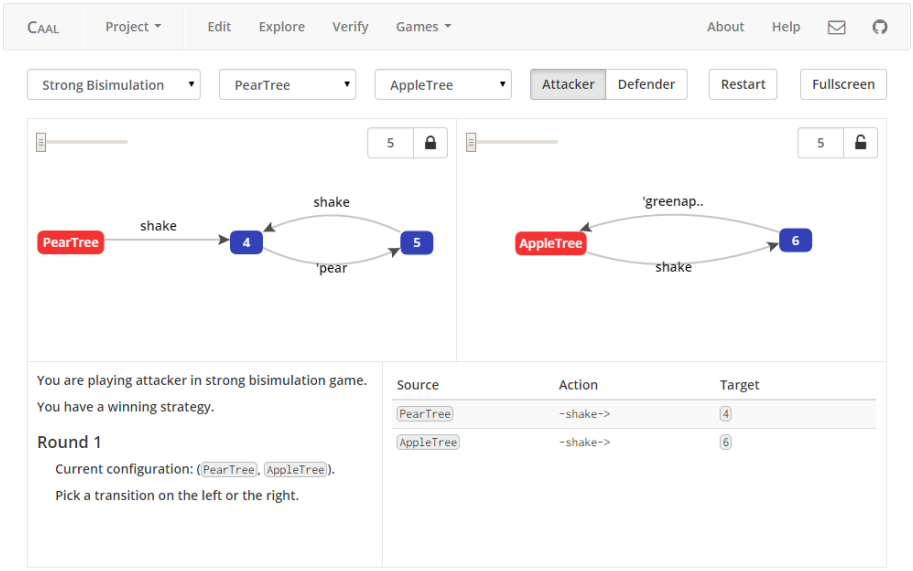


Figure 8.9: Screenshot of the strong bisimulation game.

The game settings in the top specifies that it is a strong bisimulation game between the processes PearTree and ColorTree where the player is playing as attacker. We also have the option to restart the game to the (PearTree, ColorTree) configuration.

The LTSs generated by the processes PearTree and ColorTree are shown in Figure 8.10, where the current configuration of the game is highlighted in red. The two LTSs have the same options (e.g. lock, zoom, etc.) as in the explorer.

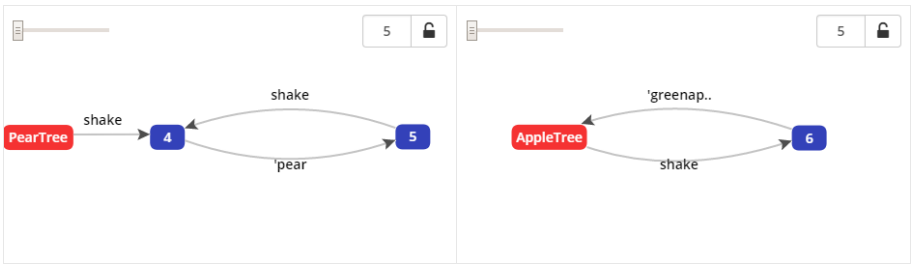


Figure 8.10: Game LTSs.

Figure 8.11 shows the different transitions available to the player. It consists of three columns:

Source The source state of the transition. Can be either the current state in the left LTS or the current state in the right LTS.

Action The label of the transition.

Target The destination state of the transition.

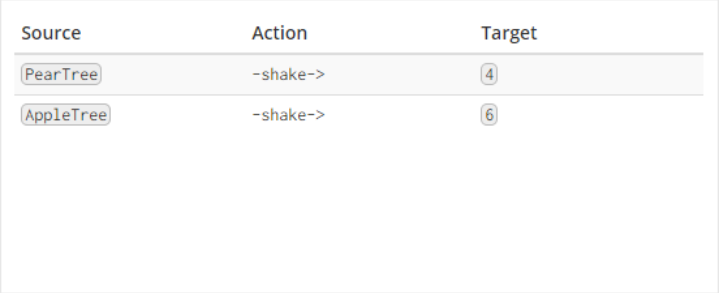


Figure 8.11: Available transitions.

Figure 8.12 shows the different steps of a full game in the game log. The initial state of the game log is shown in Figure 8.12a, where the role of the player and whether or not the player has a universal winning strategy is shown. The player then knows if a loss was due to a bad move. The game log then prompts the player to pick a transition.

Figure 8.12b shows the game log after the player has made the attack

$$\text{PearTree} \xrightarrow{\text{shake}} 4$$

on the left LTS, where 4 is the identifier of the target state.

Figure 8.12c shows the response of the defender

$$\text{ColorTree} \xrightarrow{\text{shake}} 6$$

on the right. The next round of the game starts and the game log shows the current configuration of the game $(4, 6)$. The player can now attack again on the left or right.

Figure 8.12d shows the player attacking with the transition

$$4 \xrightarrow{\text{pear}} 5$$

on the left. The defender cannot match the pear transition on the right. The player wins the game which means that the processes PearTree and ColorTree are not strongly bisimilar.

(a) Game log with introduction.

(b) Game log after an attack.

(c) Game log after a defend.

(d) Game log with a winner.

Figure 8.12: The game log.

HML Game

An HML game consists of an "attacker", a "defender", a process s, and a formula φ. A play of a game starting from the start state s is a maximal sequence of configurations formed by the players according to the following rules. Each round either the attacker or the defender picks a successor configuration if possible.

- The attacker picks a configuration when the formula is of the form $(s, \varphi_1 \wedge \varphi_2)$, or when the choices are either $(s, [\alpha]\varphi)$ or $(s, [[\alpha]]\varphi)$.
- The defender picks a configuration when the formula is of the form $(s, \varphi_1 \vee \varphi_2)$, or when the choices are $(s, \langle\alpha\rangle\varphi)$ or $(s, \langle\langle\alpha\rangle\rangle\varphi)$.

The winner depends on which configuration the game ends in, or alternatively the context of an infinite play.

- The attacker is the winner in every play ending in a configuration of the form (s, ff) or in play in which the defender gets stuck.
- The defender is the winner in every play ending in configuration of the form (s, tt) or in a play in which the attacker gets stuck.
- The attacker is the winner in every infinite play in context X provided that X is a defined as a minimum fixed-point: $X \overset{\min}{=} \varphi$. The defender is the winner in every infinite play provided that X is defined as a maximum fixed-point: $X \overset{\max}{=} \varphi$.

Figure 8.13 shows an HML game where the user is playing as defender against the computer.

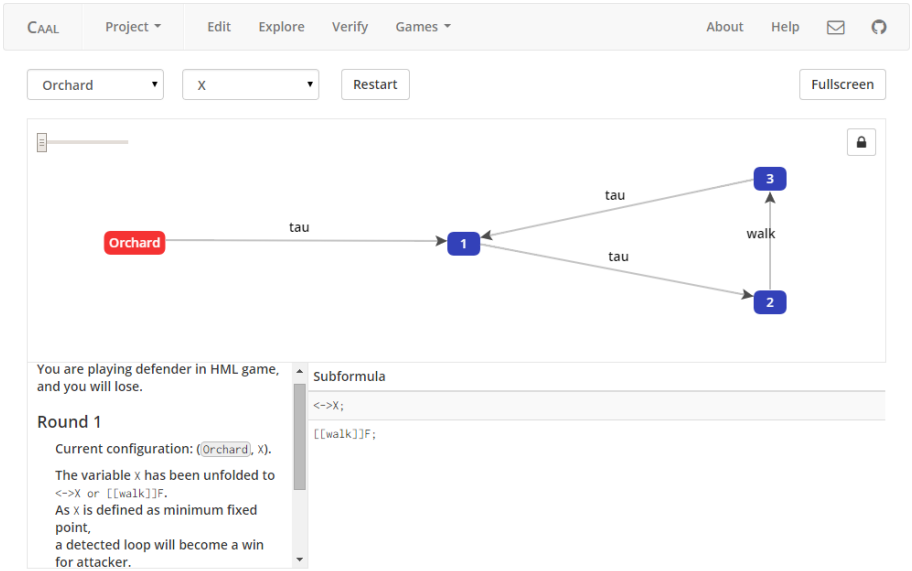


Figure 8.13: HML Game.

The game consists of a few different elements:

- the process and formula at the top left,
- the LTS in the middle,
- game log at the bottom left, and
- subformula and transition table at the bottom right.

Using Example 8.2 we now want to play an HML game to verify that the result is correct and that the formula is indeed not satisfied. We have the Orchard process given by

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{AppleTree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{AppleTree + } \overline{\texttt{'redapple}}.\texttt{AppleTree}) \ , \\ \texttt{Orchard} &= (\texttt{AppleTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ , \end{aligned}$$

and the formula

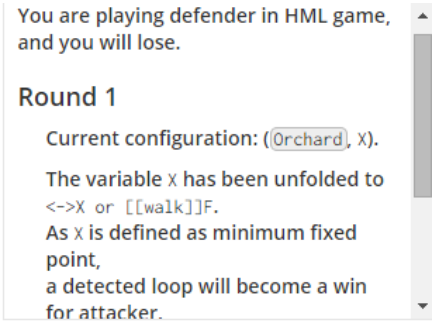
$$\texttt{X min= [[walk]]ff or <->X} \ .$$

The initial game log can be seen in Figure 8.14a. As it can be seen, we are playing as defender and we are going to lose, matching the claim from Figure 8.3 that the formula is not satisfied.

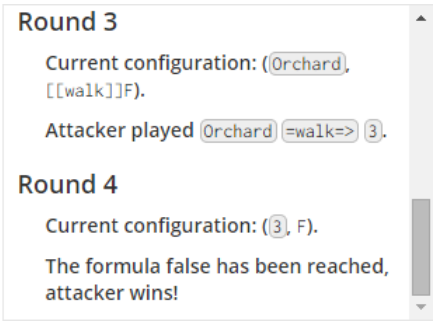
As defender we have to choose which of the disjunctions we want to continue from. We can pick between two subformulas:

1. `[[walk]]ff` and
2. `<->X` .

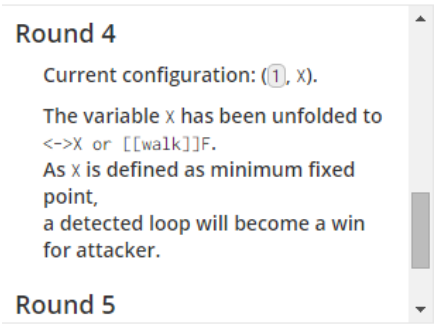
Taking case 1 will result in a loss in the next round because the attacker picks a transition so we reach a false formula as shown in Figure 8.14b. Instead the defender picks case 2 and picks a transition, resulting in `X` being unfolded as it can be seen on Figure 8.14c. The game continues with the defender picking one of the two cases, each time unfolding `X` and having to pick a transition as seen on Figure 8.14d. Figure 8.15 shows the transition table for the defender. Eventually the game will detect a cycle as it can be seen in Figure 8.14e, which means the defender loses because we played in a minimum fix-point game.



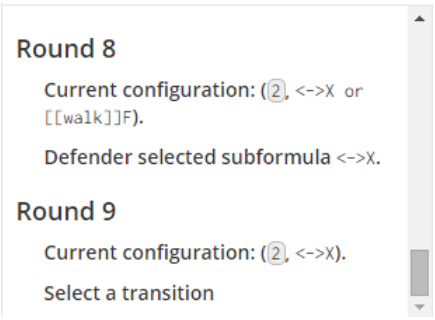
(a) Game log with introduction.



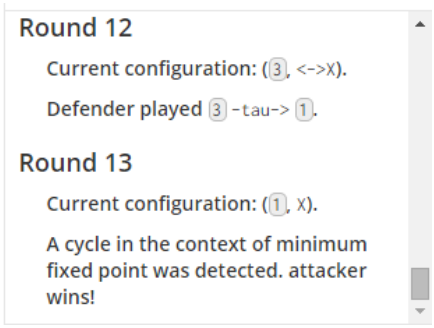
(b) Reaching false formula.



(c) Unfold X.



(d) Select a transition.



(e) Game log with a winner.

Figure 8.14: Game log for HML Game.

Source	Action	Target
2	walk	3

Figure 8.15: Transition table.

Distinguishing Formula

HML formulas can also be used to check if two processes are strongly bisimilar. Two processes are strongly bisimilar if and only if they satisfy the same formulas. This also implies that if two processes are not strongly bisimilar, then there must exist a formula that distinguishes them.

Example 8.5

We have the processes

$$\texttt{Man} = \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ ,$$
$$\texttt{FastMan} = \overline{\texttt{'shake}}.(\texttt{redapple.}(\texttt{walk.FastMan + FastMan}) + \texttt{greenapple.}(\texttt{walk.FastMan + FastMan}) \ .$$

CAAL is able to generate a distinguishing formula for two processes. Figure 8.16 shows a generated distinguishing formula for the two processes Man and FastMan which are not strongly bisimilar. This is done by clicking the three vertical dots on the right-hand side and clicking on 'Distinguishing formula'.

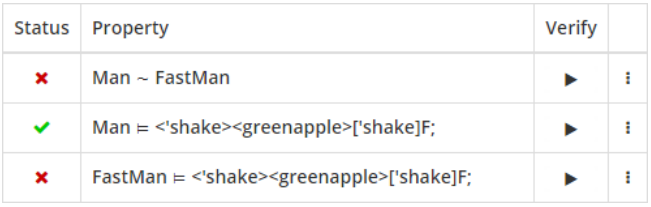


Figure 8.16: Distinguishing formula.

Distinguishing Trace

Much like the distinguishing formula, CAAL can also generate a trace that distinguishes two processes. When checking whether two processes are trace equivalent or if one process is a trace inclusion of the other, CAAL will output the distinguishing trace if this is not the case. Figure 8.17 shows a distinguishing trace for the processes Man and FastMan from Example 8.5.

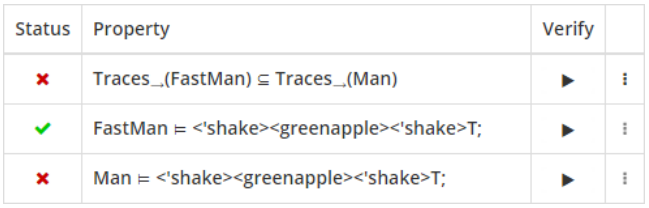


Figure 8.17: Distinguishing trace.

As we can see, the FastMan affords the trace

$$\overline{\text{'shake}}.\text{greenapple}.\overline{\text{'shake}}\ ,$$

which the Man does not. The distinguishing trace is given as an HML formula so that the HML game can be loaded.

8.6 Closing Remarks

CAAL is an open source project developed at Aalborg University by Jacob Karstensen Wortmann, Jesper Riemer Andersen, Nicklas Andersen, Mathias Munk Hansen, Simon Reedtz Olesen, and Søren Enevoldsen under the supervision of Jiří Srba and Kim Guldstrand Larsen.

The source code can be found on GitHub at `https://github.com/caal/caal`. We welcome suggestions and bug reports either through the issue tracker on GitHub, or via e-mail at caal@cs.aau.dk.

Conclusion *9*

Progressing on earlier work we described the theory and implementation of various extensions to Caal.

The complete semantics for recursive HML with regards to fixed-points and variables was left unfinished in earlier work. We refine and complete the semantics for recursive HML and describe the translation of HML formulae to sets of dependency graphs which encodes satisfiability, the result of which can be found by computing the fixed-point assignment to the dependency graphs.

To complement the bisimulation game already implemented in Caal, we describe the rule of the recursive HML game, and the relation between satisfiability and which player, either the attacker or defender, has a universal winning strategy in the game. This is followed by a reduction to dependency graphs for implementing the game. The implemented game is played against the computer who always beat the player using its universal winning strategy in order to to show why a model checking result holds.

Given a dependency graph used for computing bisimulation, we describe how the fixed-point assignment can be interpreted for non-bisimilar processes in order to produce a distinguishing formula. Ideally the user would prefer the simplest distinguishing formula possible. However, with the exception of minor simplifications like removing redundant pre- or post-fixed tau-modalities, this turns out not to be straightforward. We conjecture that the general problem of finding the simplest possible distinguishing formula is NP-hard, but offer no proof. Instead we present a greedy algorithm that will eliminate redundant terms from formulae, but will not always eliminate the most possible. We show a related decision-problem is NP-hard, implying this particular problem of eliminating the most terms from formulae has exponential time complexity.

Bisimulation collapse is another feature that provides overview when visualising processes in Caal. Equivalent processes are simplified by picking a representative among the equivalence class. We describe an efficient algorithm for computing the representatives using the disjoint-set algorithm. Collapsing processes in the process explorer in Caal can reduce the clutter and enable to user to hide uninteresting processes.

The non-visual part of Caal was adapted for use in our parallel algorithm prototype. The algorithm combines a database, and wrapped JavaScript successor generators in C++, to run multiple worker processes in parallel, each computing part of a large dependency graph. The speedup when running multiple processes was poor and it underlying successor generators accumulated memory resulting in the prototype algorithm not working for large problems. The main culprit or poor performance appears to be poor work distributions; processes make redundant computations on the same data.

An earlier version of Caal, including support for generating distinguishing formulae without simplifications, was used by students in February 2015 for the course in semantics and verification. Feedback from the students were in general on the positive side; for returned questionnaires asking about the usability of Caal, the average score was 7.0/10.

10 Future Work

Improvements to Caal

Play as Winner In the current state of the HML game, the user is only able to play as the losing side. While this might help the user getting a better understanding of why a formula is satisfied or not satisfied it might not help the user understand why that exact strategy is the universal winning strategy.

Since Caal is meant as an educational tool, being able to play as the winning side and allowing the user to "discover" the universal winning strategy for themselves might give them a better understanding of their system and convince them even more.

Rework Verifier Interface Since Caal is an educational tool it is important that it is easy and intuitive to use. Therefore a rework of the verifier interface could be needed to enhance the usability and make it easier for the users to use.

HML Maximum Fixed-Point in HML Game In the current state, the HML game does not support maximum fixed-point formulae.

More Operators Operators such as "Until" and "Weak Until" known from CTL could be implemented. The users already have the expressive power from HML but the syntax from CTL may be easier to comprehend for users not accustomed to HML.

Multi Person Editor Group work is a very big part of studying at Aalborg University, and for that reason it could be interesting to implement the functionality for more users to work in the same editor at the same time.

Parallel Algorithm with Better Speedup

The parallel algorithm implemented has poor speedup. It would be interesting to see what performance could be obtained if more work was invested in correcting the shortcomings of the algorithm. The successor calculations take almost exclusively take up the running time. Moving the successor generation to a faster performing language might increase the speed by several factors. Making the successor generator completely stateless might enable the algorithm to handler larger problems, however the caching done helps performance and without it the algorithm might become slower.

The most important change would like be better work distribution. Processes should never handle the same work; ideally work should be distributed evenly, but also in such a way that the partial graph a process work on is connected as much as possible.

NP-hardness of Simplest Distinguishing Formula

In Section 4.3 we conjectured that determining whether two non-bisimilar processes have a distinguishing formula with at most a given modal depth and number of conjunctions or disjunctions is NP-hard. The complexity of this problem is interesting. If the problem is NP-hard then it would show our greedy algorithm is justified. It it is not NP-hard then there may be an algorithm that can compute the simplest distinguishing formula in polynomial time.

Bibliography

[1] REmote DItionary Server. `http://redis.io/`. Last Accessed June 8th 2015.

[2] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiří Srba. *Reactive Systems: Modelling Specification and Verification*. Cambridge University Press, 2007. ISBN 978-0-521-87546-2.

[3] Nicklas Andersen, Mathias Munk Hansen, and Jesper Riemer Andersen. CAAL 2.0 - Equivalences, Preorders and Games for CCS and TCCS, June 2015.

[4] Rance Cleaveland, Steve Sims, and Tan Li. The Concurrency Workbench of the New Century. `https://www3.cs.stonybrook.edu/~cwb/overview.html`.

[5] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, January 1993. ISSN 0164-0925. doi: 10.1145/151646.151648. URL `http://doi.acm.org/10.1145/151646.151648`.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Riverst, and Clifford Stein. *Introduction To Algorithms*, pages 561–572. MIT Press, 3rd edition, 2009. ISBN 978-0262033848.

[7] Google. Chrome v8. `https://developers.google.com/v8/`, May 7th 2015.

[8] Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM*, 32:137–161, January 1985.

[9] Jonas Finnemann Jensen, Kim Guldstrand Larsen, Jiří Srba, and Lars Kaerlund Oestergaard. Local model checking of weighted CTL with upper-bound constraints. In Ezio Bartocci and C.R. Ramakrishnan, editors, *Model Checking Software*, volume 7976 of *Lecture Notes in Computer Science*, pages 178–195. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39175-0. doi: 10.1007/978-3-642-39176-7_12. URL `http://dx.doi.org/10.1007/978-3-642-39176-7_12`.

[10] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360251. URL `http://doi.acm.org/10.1145/360248.360251`.

[11] Kim G. Larsen. Proof systems for Hennessy-Milner Logic with recursion. *Theoretical Computer Science*, pages 215–230, June 2005. ISSN 0302-9743.

[12] Xinxin Liu and Scott A.Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In *Automata, Languages and Programming*, pages 53–66. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64781-2.

[13] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. ISBN 0-387-10235-33.

[14] Simon Reedtz Olesen, Nicklas Andersen, Søren Enevoldsen, Mathias Munk Hansen, Jesper Riemer Andersen, and Jacob Karstensen Wortmann. CAAL: Concurrency Workbench Aalborg Edition, January 2015.

[15] Stack Overflow. Configuring v8's memory management to be smart for a node.js process. `http://stackoverflow.com/a/16591881`, June 7th 2015.

[16] Michael Sipser. *Introduction to the Theory of Computation.* Thomson, 2006. ISBN 0-534-95097-3.

[17] Google Sites. Hardware organisation. `https://sites.google.com/site/mccaau/home/hardware`, June 7th 2015.

[18] Perdita Stevens. The Edinburgh Concurrency Workbench. `http://homepages.inf.ed.ac.uk/perdita/cwb`.

[19] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.