Aalborg University

Caal 2.0

Equivalences, Preorders and Games for CCS and TCCS

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

Title:
CAAL

Subject:
Semantics and
Verification

Project period:
Spring semester 2015

Project group:
DES104F15

Attendees:
Jesper Riemer Andersen
Mathias Munk Hansen
Nicklas Andersen

Supervisors:
Jiří Srba
Kim Guldstrand Larsen

Finished: 09-06-2015

Number of pages: 86

Synopsis:

This report documents the continued development of the tool Concurrency Workbench - Aalborg Edition (CAAL). CAAL supports modelling, visualization, and verification of concurrent systems. Concurrent systems are modelled using the well-known process algebra CCS.

In this project we extend CAAL with additional equivalences and preorders, and add interactive games for process equivalence, where the user plays a game against the computer in an attempt to either prove or disprove the result of a given equivalence checking problem.

We also implement the timed process algebra TCCS in CAAL. In connection with this, we also implement timed variants of the existing equivalences and preorders, support for model checking on timed systems, and timed games.

We also propose a general equivalence relation which can express a wide range of equivalences and, including timed variants, and show reductions from such relations to dependency graphs.

Signatures

Jesper Riemer Andersen

Nicklas Andersen

Mathias Munk Hansen

Preface

This project was written as part of a Master's Thesis by a group of Software Engineering students from the Department of Computer Science at Aalborg University during the spring of 2015.

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

Contents

Introduction 1

This report documents the continued development of the tool Concurrency Workbench - Aalborg Edition (Caal) which started in the fall semester of 2014 [14]. Caal supports modelling, visualization, and verification of concurrent systems. Concurrent systems are modelled using the well-known process algebra Calculus of Communicating Systems (CCS) [8], and can be graphically explored, which is an invaluable feature when debugging. Verification is supported both through the equivalence checking approach, where an implementation is tested against a specification, and through the model checking approach, where it is determined if a system (or part of it) enjoys certain properties. All verification algorithms are based on the notion of dependency graphs and use on-the-fly exploration of the state space.

Other tools with similar functionality to Caal already exist, where the Edinburgh Concurrency Workbench (CWB) [11] is the most prominent. CWB is a powerful tool with support for multiple process algebras, equivalences, model checking, etc. and has served as a great source of inspiration for Caal. Unfortunately, CWB is no longer maintained, there are no official binaries available, and the tool cannot be built with newer versions of the Standard ML compiler. As a result, simply acquiring and installing the tool can be difficult. Furthermore, the tool does not have a graphical user interface and can only be used through a command-line interface. Having a graphical user interface allows for a wider range of visualization and debugging options and makes for a more engaging user experience. The tool Concurrency Workbench of the New Century (CWB-NC) [3] was created as a continuation of CWB, but now suffers from some of the same problems. Namely that it can no longer be downloaded from the official website and that the last update was in the year 2000. Caal has been designed as a web application to avoid some of the aforementioned problems. In particular, no local installation is required and portability is ensured across all major operating systems.

Caal was developed primarily to support the course *Semantics and Verification* offered at Aalborg University to Computer Science and Software Engineering students at their 6th semester, and has been tested by the students following the course during the spring of 2015. Being an educational tool, Caal does not aim to be the fastest nor the most powerful tool, but instead on offering visualization and debugging tools to promote learning. In this project we extend Caal with interactive games for process equivalence, where the user plays a game against the computer in an attempt to either prove or disprove the result of a given equivalence checking problem. Games are useful in an educational context as a way of visualizing counterexamples for equivalence checking problems.

Real-time systems often have strict timing constraints that cannot be expressed using CCS alone. To accommodate this, we extend Caal with support for the timed process algebra known as Timed CCS. In connection with this, we add timed variants of the existing equivalences, support for model checking on timed systems, and timed games.

Inspired by the rules of the aforementioned games, we introduce a general equivalence relation, which can be used to define a wide range of equivalences and preorders. The relation takes a set of parameters corresponding to the rules of the game which characterizes the equivalence that we are trying to define. We then show how problems of this type can be verified using dependency graphs. Finally, we extend this general equivalence relation to cover timed equivalences and preorders.

1.1 Preliminaries

We extend the set of all natural numbers $\mathbb{N}$ with ∞ and 0, and define it as $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, $\mathbb{N}_0^\infty = \mathbb{N} \cup \{\infty, 0\}$, and assume that $\infty + n = \infty$ and $\infty - n = \infty$ for any $n \in \mathbb{N}_0$.

We model processes using Labelled Transition Systems (LTSs). An LTS consists of a set of states (or processes), a set of actions, and a transition relation.

Definition 1.1 - *Labelled Transition System*

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

If a state p can perform an α-labelled transition and become p' we write $p \xrightarrow{\alpha} p'$, which is also called a strong transition.

The τ-actions are internal actions which are supposed to be unobservable. The weak transition relation $\Rightarrow$, defined in Definition 1.2, abstracts away from τ-actions.

Definition 1.2 - *Weak Transition*

Let $(\texttt{Proc}, \texttt{Act}, \rightarrow)$ be an LTS and let $s, t \in \texttt{Proc}$ be states. For each action $\alpha \in \texttt{Act}$ we write $s \stackrel{\alpha}{\Rightarrow} t$ if and only if either

- $\alpha \neq \tau$ and there are states s' and t' such that
$$s \left(\xrightarrow{\tau}\right)^* s' \xrightarrow{\alpha} t' \left(\xrightarrow{\tau}\right)^* t$$
- or $\alpha = \tau$ and $s \left(\xrightarrow{\tau}\right)^* t$,

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

In what follows, we use the symbol $\hookrightarrow$ to denote either the strong transition relation $\rightarrow$ or the weak transition relation $\Rightarrow$.

1.1.1 Equivalences and Preorders

We now define a number of weak and strong preorders and equivalences between states in an LTS using the transition relation $\hookrightarrow$.

Definition 1.3 - *Simulation*

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

$$\text{if } s_1 \xrightarrow{\alpha} s_1' \text{ then there is a transition } s_2 \overset{\alpha}{\hookrightarrow} s_2' \text{ such that } (s_1', s_2') \in \mathcal{R}.$$

A state s is said to *simulate* a state t if and only if there is a simulation that relates them. From now on the relation $\sqsubseteq_{\sim}$ will be referred to as *strong simulation* when $\hookrightarrow = \rightarrow$, and $\sqsubseteq_{\approx}$ will be referred to as *weak simulation* when $\hookrightarrow = \Rightarrow$.

Having defined simulation, we now use this to define simulation equivalence, which is the symmetric case of simulation.

Definition 1.4 - *Simulation Equivalence*

Two states s and t are *strong simulation equivalent* if and only if $s \sqsubseteq_{\sim} t$ and $t \sqsubseteq_{\sim} s$, and *weak simulation equivalent* if and only if $s \sqsubseteq_{\approx} t$ and $t \sqsubseteq_{\approx} s$. From now on the relation $\simeq$ will be referred to as *strong simulation equivalence* and $\approxeq$ will be referred to as *weak simulation equivalence*.

We now define the perhaps most well-known notion of equivalence, namely that of bisimulation equivalence, introduced by David Park in 1981 [9] and popularized by Robin Milner in 1989 [7].

Definition 1.5 - *Bisimulation*

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

$$\text{if } s_1 \xrightarrow{\alpha} s_1' \text{ then there is a transition } s_2 \overset{\alpha}{\hookrightarrow} s_2' \text{ such that } (s_1', s_2') \in \mathcal{R} \text{ and}$$
$$\text{if } s_2 \xrightarrow{\alpha} s_2' \text{ then there is a transition } s_1 \overset{\alpha}{\hookrightarrow} s_1' \text{ such that } (s_1', s_2') \in \mathcal{R}.$$

Two states s and t are *bisimilar* if and only if there is a bisimulation that relates them. From now on the relation $\sim$ will be referred to as *strong bisimilarity* when $\hookrightarrow = \rightarrow$, and $\approx$ will be referred to as *weak bisimilarity* when $\hookrightarrow = \Rightarrow$.

It is easy to confuse simulation equivalence and bisimulation equivalence since their respective definitions look similar, but there is an important difference. We now illustrate this difference with an example.

Example 1.6

Consider the two states s_1 and t_1 shown in Figure 1.1. To show that $s_1 \simeq t_1$ we must show that $s_1 \sqsubseteq_{\sim} t_1$ and that $t_1 \sqsubseteq_{\sim} s_1$. We show a strong simulation $\mathcal{R}_1$

such that $(s_1, t_1) \in \mathcal{R}_1$ and a strong simulation $\mathcal{R}_2$ such that $(t_1, s_1) \in \mathcal{R}_2$:

$$\mathcal{R}_1 = \{(s_1, t_1), (s_2, t_2), (s_3, t_3)\},$$
$$\mathcal{R}_2 = \{(t_1, s_1), (t_2, s_2), (t_3, s_2), (t_3, s_3)\}.$$

As we can see, s_1 and t_1 are indeed strong simulation equivalent, but they are not strongly bisimilar because of the transition $t_1 \xrightarrow{a} t_3$ which can only be matched by the transition $s_1 \xrightarrow{a} s_2$. However, the state s_2 can perform the transition $s_2 \xrightarrow{b} s_3$ which t_3 cannot match.

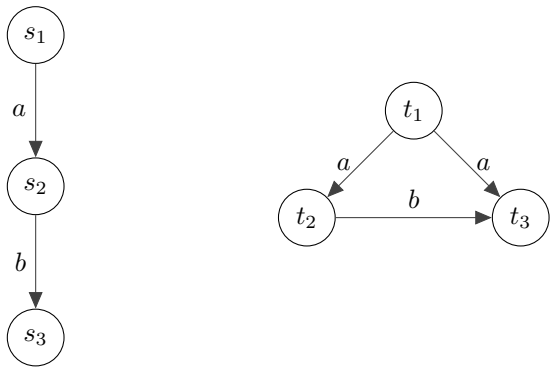


Figure 1.1: Two simulation equivalent states s_1 and t_1.

We now show an example to illustrate the difference between strong bisimulation and weak bisimulation.

Example 1.7

We relabel all b-actions in the LTS shown in Figure 1.1 to τ-actions. The new LTS is shown in Figure 1.2. In Example 1.6 we showed that $s_1 \not\sim t_1$, and clearly the relabelling does not change this fact. To prove that $s_1 \approx t_1$ we show a weak bisimulation $\mathcal{R}$ such that $(s_1, t_1) \in \mathcal{R}$:

$$\mathcal{R} = \{(s_1, t_1), (s_2, t_2), (s_2, t_3), (s_3, t_2), (s_3, t_3)\}.$$

We have now shown that $s_1 \approx t_1$.

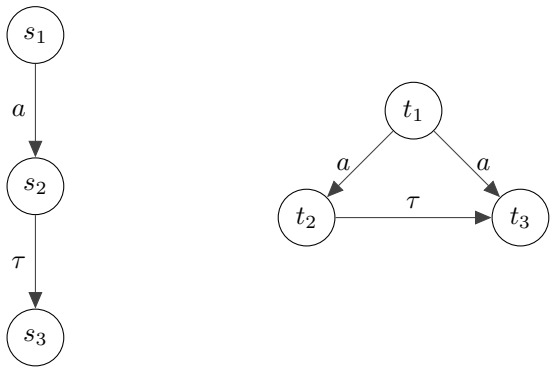


Figure 1.2: Two weakly bisimilar states s_1 and t_1.

Another way to look at process behavior is by considering their traces. Traces are defined in Definition 1.8.

Definition 1.8 - *Traces*

A strong trace from a state s is a sequence of actions $\alpha_1 \cdots \alpha_n \in \mathtt{Act}^*$ where $n \geq 0$ such that there exists a sequence of strong transitions

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s_n,$$

for some states $s_1, \ldots, s_n$.

A weak trace from a state s is a sequence of actions $\alpha_1 \cdots \alpha_n \in (\mathtt{Act} \setminus \{\tau\})^*$ where $n \geq 0$ such that there exists a sequence of weak transitions

$$s_0 \overset{\alpha_1}{\Longrightarrow} s_1 \overset{\alpha_2}{\Longrightarrow} \ldots \overset{\alpha_{n-1}}{\Longrightarrow} s_{n-1} \overset{\alpha_n}{\Longrightarrow} s_n,$$

for some states $s_1, \ldots, s_n$.

We write $\mathit{Traces}_{\rightarrow}(s)$ for the collection of all strong traces of s, and $\mathit{Traces}_{\Rightarrow}(s)$ for the collection of all weak traces of s.

Having defined strong and weak traces we can now define trace inclusion.

Definition 1.9 - *Trace Inclusion*

Let s and t be states in an LTS. We say that s is a *trace inclusion* of t if and only if

$$\mathit{Traces}_{\hookrightarrow}(s) \subseteq \mathit{Traces}_{\hookrightarrow}(t).$$

From now on the relation $\subsetneq$ will be referred to as *strong trace inclusion* when $\hookrightarrow = \rightarrow$, and $\lessapprox$ will be referred to as *weak trace inclusion* when $\hookrightarrow = \Rightarrow$.

We now define trace equivalence, which is the symmetric case of trace inclusion.

Definition 1.10 - *Trace Equivalence*

Let s and t be states in an LTS. We say that s and t are *trace equivalent* if and only if

$$\mathit{Traces}_{\hookrightarrow}(s) = \mathit{Traces}_{\hookrightarrow}(t).$$

From now on the relation $\simeq_T$ will be referred to as *strong trace equivalence* when $\hookrightarrow = \rightarrow$, and $\approx_T$ will be referred to as *weak trace equivalence* when $\hookrightarrow = \Rightarrow$.

We now show an example of strong trace equivalence.

Example 1.11

Consider the two states s_1 and t_1 shown in Figure 1.3. To show that $s_1 \simeq_T t_1$ we must show that $\mathit{Traces}_{\rightarrow}(s_1) = \mathit{Traces}_{\rightarrow}(t_1)$. We show the traces of s_1 and the traces of t_1:

$$\mathit{Traces}_{\rightarrow}(s_1) = \{\varepsilon, a, ab, ac\}$$
$$\mathit{Traces}_{\rightarrow}(t_1) = \{\varepsilon, a, ab, ac\}$$

As we can see, s_1 and t_1 are indeed strong trace equivalent. As an aside, s_1 and t_1 are not strong simulation equivalent since $s_1 \not\simeq t_1$.

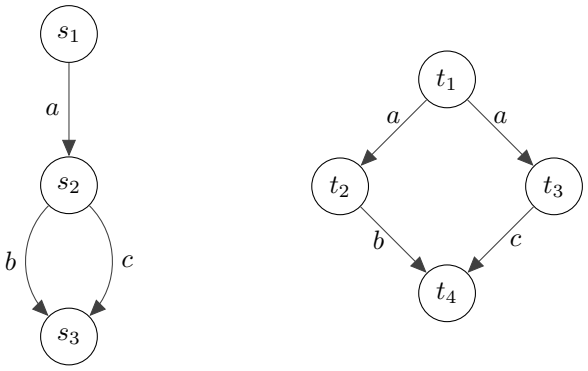


Figure 1.3: Two trace equivalent states s_1 and t_1.

1.1.2 CCS

CCS is a process algebra for modelling concurrent systems introduced by Robin Milner in 1980 [8], and was considered a major breakthrough at the time. We now give an informal introduction to the basic operators of CCS, and show how these operators can be combined to model complex systems.

The most basic operator in CCS is the *action prefixing* operator denoted by a period. With this operator we can model processes such as

$$a.0\ ,$$

which means that after performing the action a, the process becomes the 0 (*nil*) process. The 0 process is a special process which cannot perform any action whatsoever. Actions can be thought of as inputs (e.g. a) and outputs (e.g. $\overline{a}$).

Now let us look at a more interesting example. We model a person driving a car, and give it the name Driver:

$$\text{Driver} \stackrel{\text{def}}{=} \text{drive}.0 \ .$$

This process can perform the action drive and become the 0 process. With process names we can also make recursive definitions. Consider the process

$$\text{Driver} \stackrel{\text{def}}{=} \text{drive}.\text{Driver} \ ,$$

which can perform the action drive and become the Driver process once again.

At some unfortunate point in time the driver might crash, modelled with the *choice* operator + as

$$\text{Driver} \stackrel{\text{def}}{=} \text{drive}.\text{Driver} + \text{drive}.\overline{\text{crash}}.0 \ ,$$

which means that the driver can nondeterministically continue driving or crash and become the 0 process. In order to protect the driver, we create an airbag:

$$\text{Airbag} \stackrel{\text{def}}{=} \text{crash}.\overline{\text{inflate}}.\text{Airbag} \ .$$

For the sake of example, we assume that the driver can continue driving after the airbag has been inflated, which we model as

$$\text{Driver} \stackrel{\text{def}}{=} \text{drive}.\text{Driver} + \text{drive}.\overline{\text{crash}}.\text{inflate}.\text{Driver} \ .$$

The driver and the airbag are still separate entities, performing actions independent of each other. To have the processes run concurrently we can combine them using the *parallel composition* operator | as

$$\text{Car} \stackrel{\text{def}}{=} \text{Driver} \mid \text{Airbag} \ .$$

The driver and the airbag can now communicate over their shared communication channels, crash and inflate, resulting in only a τ-action being visible to outside observers. However, they may also proceed to execute independently. We can force the driver and the airbag to communicate by restricting communication on the crash and inflate channels:

$$\text{Car} \stackrel{\text{def}}{=} (\text{Driver} \mid \text{Airbag}) \setminus \{\text{crash}, \text{inflate}\} \ .$$

This means that the only available action will be τ, effectively forcing the Driver and the Airbag processes to communicate.

1.1.2.1 Syntax and Semantics

We now formally define the syntax and semantics of CCS. We assume a finite collection of *input names* $\mathcal{A}$, and the set of *output names* $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$. Let $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ be the set of *labels* and let $\texttt{Act} = \mathcal{L} \cup \{\tau\}$ be the set of actions. We assume a finite collection $\mathcal{K}$ of *process names* or *process constants*.

Definition 1.12 - *CCS Syntax*

The collection of CCS expressions $\mathcal{P}$ is given by the following grammar:

$$P ::= K \;\Big|\; \alpha.P \;\Big|\; \Sigma_{i\in I} P_i \;\Big|\; \big|_{i\in I} P_i \;\Big|\; P[f] \;\Big|\; P \setminus L \;\Big|\; 0$$

where $K \in \mathcal{K}$ is a process name, $\alpha \in \texttt{Act}$ is an action, I is a finite index set, $L \in \mathcal{A}$ is a set of labels, 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 behavior of each process name is given by its 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.

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.

Definition 1.13 - *CCS Semantics*

The semantics of CCS is given by the following SOS rules:

$$\text{SUM}\ \frac{P_j \xrightarrow{\alpha} P'_j}{\left(\sum_{i\in I} P_i\right) \xrightarrow{\alpha} P'_j} \text{ where } j \in I \qquad \text{ACT}\ \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{RES}\ \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \overline{\alpha} \notin L \qquad \text{CON}\ \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K \stackrel{\text{def}}{=} P$$

$$\text{COM1}\ \frac{P_j \xrightarrow{\alpha} P'_j}{\left(\Big|_{i\in I} P_i\right) \xrightarrow{\alpha} \left(\Big|_{i\in I\setminus\{j\}} P_i \mid P'_j\right)} \text{ where } j \in I \qquad \text{REL}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{COM2}\ \frac{P_j \xrightarrow{a} P'_j \quad P_k \xrightarrow{\overline{a}} P'_k}{\left(\Big|_{i\in I} P_i\right) \xrightarrow{\tau} \left(\Big|_{i\in I\setminus\{j,k\}} P_i \mid P'_j \mid P'_k\right)} \text{ where } j, k \in I \text{ and } j \neq k$$

where $\alpha \in \texttt{Act}$, $a \in \mathcal{L}$, and I is a finite index set.

1.1.2.2 Unguarded Recursion

A desired property is that any CCS expression has finitely many derivations that are all finite. First we define what it means for a process to be guarded.

Definition 1.14 - *Guarded Process Constant*

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

If a process constant is not guarded, we say it is *unguarded*. We provide an alternative definition in terms of a *reference graph*.

Definition 1.15 - *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).

Consider the following process definitions:

$$P \stackrel{\text{def}}{=} Q + a.R \ ,$$
$$Q \stackrel{\text{def}}{=} R + b.0 \ ,$$
$$R \stackrel{\text{def}}{=} P + c.R + R \ .$$

Figure 1.4 shows the corresponding reference graph.

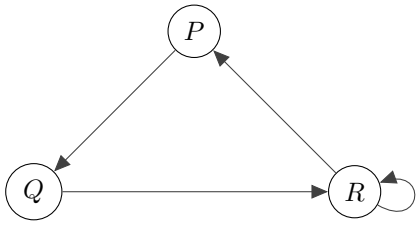


Figure 1.4: Cycles representing unguarded recursion.

The edges in the reference graph illustrate that P depends on Q, which depends on R, which depends on P and itself, leading to a cycle in the reference graph. Cycles in the reference graph represent unguarded recursion.

Definition 1.16 - *Weak Guardedness*

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

1.1.2.3 Structural Congruence

We now present a number of structural congruence rules which preserve behavior (bisimulation equivalence). The parallel operator | and the choice operator + are commutative and associative, and the choice operator is idempotent. In the following P, R, and Q are processes, L is a set of labels, and f is a relabelling function. Table 1.1 shows rules for associativity, Table 1.2 shows rules for process equivalence, and Table 1.3 shows congruence rules for CCS.

$$(P_1^1 + P_2^1 + \cdots + P_{n_1}^1) + (P_1^2 + P_2^2 + \cdots + P_{n_2}^2) + \cdots + (P_1^m + P_2^m + \cdots + P_{n_m}^m) \equiv (P_1^1 + P_2^1 + \cdots + P_{n_1}^1 + P_1^2 + P_2^2 + \cdots + P_{n_2}^2 + \cdots + P_1^m + P_2^m + \cdots + P_{n_m}^m)$$

$$(P_1^1 \mid P_2^1 \mid \cdots \mid P_{n_1}^1) \mid (P_1^2 \mid P_2^2 \mid \cdots \mid P_{n_2}^2) \mid \cdots \mid (P_1^m \mid P_2^m \mid \cdots \mid P_{n_m}^m) \equiv (P_1^1 \mid P_2^1 \mid \cdots \mid P_{n_1}^1 \mid P_1^2 \mid P_2^2 \mid \cdots \mid P_{n_2}^2 \mid \cdots \mid P_1^m \mid P_2^m \mid \cdots \mid P_{n_m}^m)$$

Table 1.1: Associativity rules.

$$(P \setminus L_1) \setminus L_2 \equiv P \setminus (L_1 \cup L_2)$$

$$(P\ [f_1])\ [f_2] \equiv P\ [f_2 \circ f_1]$$

$$\sum_{i \in I} P_i \equiv \sum_{i \in I \setminus Z} P_i \quad \text{where } Z = \{i \in I \mid P_i \equiv 0\}$$

$$\Big|_{i \in I} P_i \equiv \Big|_{i \in I \setminus Z} P_i \quad \text{where } Z = \{i \in I \mid P_i \equiv 0\}$$

$$0 \setminus L \equiv 0$$

$$0\ [f] \equiv 0$$

$$P \setminus \emptyset \equiv P$$

Table 1.2: Equivalence rules.

$$\frac{P \equiv Q}{\alpha.P \equiv \alpha.Q}$$

$$\frac{P \equiv Q}{P[f] \equiv Q[f]}$$

$$\frac{P \equiv Q}{P \setminus L \equiv Q \setminus L}$$

$$\frac{P_i \equiv Q_i \text{ for each } i \in I}{\sum_{i \in I} P_i \equiv \sum_{i \in I} Q_i}$$

$$\frac{P_i \equiv Q_i \text{ for each } i \in I}{|_{i \in I} P_i \equiv |_{i \in I} Q_i}$$

Table 1.3: Congruence rules.

1.1.3 Recursive HML

In the following we present the property language known as Hennessy-Milner Logic (HML), which was introduced by Matthew Hennessy and Robin Milner

in 1985 [4]. We will present an extension of HML with a single recursively defined variable [5].

Definition 1.17 - *Recursive HML Syntax*

Let $(\texttt{Proc}, \texttt{Act}, \rightarrow)$ be an LTS. The set $\mathcal{M}_X$ of HML formulas with a single recursively defined variable X is given by the following abstract syntax:

$$F, G ::= tt \;\Big|\; ff \;\Big|\; F \wedge G \;\Big|\; F \vee G \;\Big|\; \langle\alpha\rangle\, F \;\Big|\; [\alpha]\, F \;\Big|\; \langle\langle\alpha\rangle\rangle\, F \;\Big|\; [[\alpha]]\, F \;\Big|\; X,$$

where $\alpha \in \texttt{Act}$ is an action. We use tt and ff to denote "true" and "false", respectively. The variable X is defined by an equation of the form

$$X \stackrel{min}{=} F_X \quad \text{or} \quad X \stackrel{max}{=} F_X,$$

where F_X is a formula which may contain occurrences of X. If $A = \{a_1, \ldots, a_n\} \subseteq \texttt{Act}$ and $n \geq 0$ we use the abbreviations:

$\langle A\rangle\, F$ for the formula $\langle a_1\rangle\, F \vee \ldots \vee \langle a_n\rangle\, F$,

$\langle\langle A\rangle\rangle\, F$ for the formula $\langle\langle a_1\rangle\rangle\, F \vee \ldots \vee \langle\langle a_n\rangle\rangle\, F$,

$[A]\, F$ for the formula $[a_1]\, F \wedge \ldots \wedge [a_n]\, F$,

$[[A]]\, F$ for the formula $[[a_1]]\, F \wedge \ldots \wedge [[a_n]]\, F$.

The semantics of a formula F (which may contain a single variable X) is interpreted as a function

$$\mathcal{O}_F : \mathcal{P}(\texttt{Proc}) \rightarrow \mathcal{P}(\texttt{Proc}),$$

which returns the set of states that satisfy F when given the set of states that are assumed to satisfy X.

Definition 1.18 - *Recursive HML Semantics*

Let $(\texttt{Proc}, \texttt{Act}, \rightarrow)$ be an LTS. For each $S \subseteq \texttt{Proc}$ and formula F we define $\mathcal{O}_F(S)$ inductively as follows:

$$\begin{aligned}
\mathcal{O}_X(S) &= S, \\
\mathcal{O}_{tt}(S) &= \texttt{Proc}, \\
\mathcal{O}_{ff}(S) &= \emptyset, \\
\mathcal{O}_{F_1 \wedge F_2}(S) &= \mathcal{O}_{F_1}(S) \cap \mathcal{O}_{F_2}(S), \\
\mathcal{O}_{F_1 \vee F_2}(S) &= \mathcal{O}_{F_1}(S) \cup \mathcal{O}_{F_2}(S), \\
\mathcal{O}_{\langle a\rangle F}(S) &= \langle\cdot a\cdot\rangle\, \mathcal{O}_F(S), \\
\mathcal{O}_{[a]F}(S) &= [\cdot a\cdot]\, \mathcal{O}_F(S), \\
\mathcal{O}_{\langle\langle a\rangle\rangle F}(S) &= \langle\langle\cdot a\cdot\rangle\rangle\, \mathcal{O}_F(S), \\
\mathcal{O}_{[[a]]F}(S) &= [[\cdot a\cdot]]\, \mathcal{O}_F(S),
\end{aligned}$$

where we use the functions $\langle\cdot a\cdot\rangle, [\cdot a\cdot], \langle\langle\cdot a\cdot\rangle\rangle, [[\cdot a\cdot]] : \mathcal{P}(\texttt{Proc}) \to \mathcal{P}(\texttt{Proc})$ which we define as:

$$\begin{aligned}
\langle\cdot a\cdot\rangle S &= \{p \in \texttt{Proc} \mid p \xrightarrow{a} p' \text{ and } p' \in S \text{ for some } p'\},\\
[\cdot a\cdot] S &= \{p \in \texttt{Proc} \mid p \xrightarrow{a} p' \text{ implies } p' \in S \text{ for each } p'\},\\
\langle\langle\cdot a\cdot\rangle\rangle S &= \{p \in \texttt{Proc} \mid p \stackrel{a}{\Rightarrow} p' \text{ and } p' \in S \text{ for some } p'\},\\
[[\cdot a\cdot]] S &= \{p \in \texttt{Proc} \mid p \stackrel{a}{\Rightarrow} p' \text{ implies } p' \in S \text{ for each } p'\}.
\end{aligned}$$

Let the set $[\![X]\!] \subseteq \texttt{Proc}$ be the set of states that satisfy X. The set $[\![X]\!]$ can be interpreted as a solution to the recursive equation

$$[\![X]\!] = \mathcal{O}_{F_X}([\![X]\!]).$$

Since the function $\mathcal{O}_{F_X}$ is a monotonic function over the complete lattice $(\mathcal{P}(\texttt{Proc}), \subseteq)$ we have by Tarski's fixed-point theorem [10] that it has unique minimum and maximum fixed-points given by:

$$\begin{aligned}
\min \mathcal{O}_{F_X} &= \bigcap \{S \subseteq \texttt{Proc} \mid \mathcal{O}_{F_X}(S) \subseteq S\},\\
\max \mathcal{O}_{F_X} &= \bigcup \{S \subseteq \texttt{Proc} \mid S \subseteq \mathcal{O}_{F_X}(S)\}.
\end{aligned}$$

Definition 1.19 - *Satisfaction Relation*

Let s be a state in an LTS and let F_X be an HML formula with a single recursively defined variable X. We say that "s satisfies F_X" written as $s \models F_X$ if and only if

$$s \in \min \mathcal{O}_{F_X}$$

when X is defined as $X \stackrel{min}{=} F_X$, and

$$s \in \max \mathcal{O}_{F_X}$$

when X is defined as $X \stackrel{max}{=} F_X$.

Example 1.20

Consider the process shown in Figure 1.5. We have the recursively defined property

$$X \stackrel{min}{=} [\texttt{Act}]\mathit{ff} \vee \langle\texttt{Act}\rangle X,$$

which expresses the property of possible deadlock. We start by applying the function $\mathcal{O}_{F_X}$ to the empty set in order to find the minimum fixed-point:

$$\begin{aligned}
\mathcal{O}_{F_X}(\emptyset) &= ([\cdot a\cdot]\,\emptyset \cap [\cdot b\cdot]\,\emptyset) \cup (\langle\cdot a\cdot\rangle\,\emptyset \cup \langle\cdot b\cdot\rangle\,\emptyset)\\
&= (\{s_2, s_3\} \cap \{s_1, s_3\}) \cup (\emptyset \cup \emptyset)\\
&= \{s_3\}.
\end{aligned}$$

We now apply $\mathcal{O}_{F_X}$ to the set $\{s_3\}$:

$$\begin{aligned}\mathcal{O}_{F_X}(\{s_3\}) &= ([\cdot a\cdot]\{s_3\} \cap [\cdot b\cdot]\{s_3\}) \cup (\langle\cdot a\cdot\rangle\{s_3\} \cup \langle\cdot b\cdot\rangle\{s_3\}) \\ &= (\{s_2, s_3\} \cap \{s_1, s_3\}) \cup (\{s_1\} \cup \emptyset) \\ &= \{s_1, s_3\}.\end{aligned}$$

In this iteration the state s_1 was added to the candidate solution. We apply $\mathcal{O}_{F_X}$ to the set $\{s_1, s_3\}$:

$$\begin{aligned}\mathcal{O}_{F_X}(\{s_1, s_3\}) &= ([\cdot a\cdot]\{s_1, s_3\} \cap [\cdot b\cdot]\{s_1, s_3\}) \cup (\langle\cdot a\cdot\rangle\{s_1, s_3\} \cup \langle\cdot b\cdot\rangle\{s_1, s_3\}) \\ &= (\{s_2, s_3\} \cap \{s_1, s_3\}) \cup (\{s_1\} \cup \emptyset) \\ &= \{s_1, s_3\}.\end{aligned}$$

We have now found that the set $\{s_1, s_3\}$ is the minimum fixed-point of the function $\mathcal{O}_{F_X}$. This follows our intuition that s_2 is the only state that can never deadlock since it can always take the transition $s_2 \xrightarrow{b} s_2$.

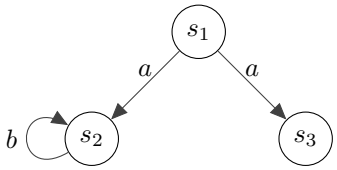


Figure 1.5: A process.

In Example 1.20 the maximum fixed-point is the set $\{s_1, s_2, s_3\}$, which is clearly not the solution we are looking for since the state s_2 can never deadlock. It turns out that minimum fixed-points are useful for expressing properties that hold if there is a finite computation proving the property (also called *liveness* properties), and maximum fixed-points are useful for expressing properties that hold unless there is a finite computation disproving the property (called *safety* properties).

1.2 Outline of the Report

The structure of the remainder of this report is as follows: In Chapter 2 we introduce the notion of dependency graphs and related concepts, which is the foundation of the verification engine in CAAL. In Chapter 3 we present a general equivalence relation between states and show reductions from such relations to dependency graphs. In Chapter 4 we introduce the process algebra Timed CCS and the logic Timed HML. We also show a number of timed equivalences and preorders and extend our general equivalence relation with time. In Chapter 5 we provide selected implementation details, and in Chapter 6 we present CAAL in the form of an informal tutorial. Chapter 7 contains the conclusion and possible directions for future work.

1.3 Bibliographical Remarks

Chapter 1 The equivalences and preorders defined in Section 1.1.1 are refinements of those found in [14] where they were originally adopted from [1]. However, Definition 1.4 and Definition 1.9 are new. The examples in Section 1.1.1 are inspired by [14].

The syntax and semantics of CCS in Section 1.1.2.1 is from [1]. Section 1.1.2.2 on unguarded recursion is reused from [14]. The structural congruence rules for CCS are refinements of [14]. The introduction to recursive HML in Section 1.1.3 is based on [1].

Chapter 2 The definitions and examples are refinements of those found in [14], except Example 2.9 which is new.

Chapter 3 Definition 3.1 is adopted from [1].

Chapter 4 The syntax and semantics of Timed CCS are based on [1], but the semantics are slightly modified. The equivalences and preorders defined in Section 4.3 are similar to those found in Section 1.1.1, but uses our own notion of untimed transitions. The examples in Section 4.3 are new. The syntax and semantics of Timed HML are extensions of the syntax and semantics of recursive HML, and the examples are new.

Chapter 6 The tutorial is an extended and modified version of the tutorial from [14] written in collaboration with [15]. Our main contributions to this chapter are Section 6.2, Section 6.4.3, Section 6.5.2, and Section 6.5.5.

Verification Using Dependency Graphs

2

In this chapter we introduce the notion of *dependency graphs* which was originally presented by Xinxin Liu and Scott A. Smolka [6]. Dependency graphs are the basis for verification in CAAL. We describe what assignments on dependency graphs are, 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 [10]. Finally, we introduce an on-the-fly algorithm for computing fixed-points on dependency graphs.

2.1 Dependency Graphs

A dependency graph is a directed graph which is used to represent dependencies between boolean variables. Dependency graphs are defined in Definition 2.1.

Definition 2.1 - *Dependency Graph*

A dependency graph 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.

We now give an example of a dependency graph.

Example 2.2

Let $G = (V, E)$ be a dependency graph where $V = \{s_1, s_2, s_3, s_4\}$ and $E = \{(s_4, \emptyset), (s_3, \{s_4\}), (s_1, \{s_3, s_4\}), (s_1, \{s_2\}), (s_2, \{s_2\})\}$. The dependency graph G can be seen in Figure 2.1.

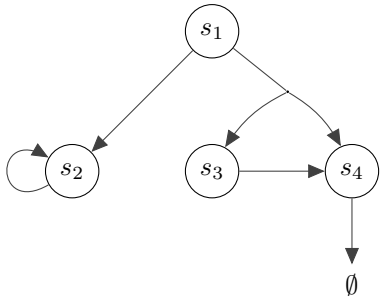


Figure 2.1: A simple dependency graph.

2.2 Assignments

To find fixed-points on dependency graphs we introduce the notion of *assignments*.

Definition 2.3 - *Assignment*

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

We can now use the assignment function A to define the pre-fixed-point assignment of a dependency graph.

Definition 2.4 - *Pre-Fixed-Point Assignment*

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

And the post-fixed-point assignment.

Definition 2.5 - *Post-Fixed-Point Assignment*

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

The partial order $\sqsubseteq$ between assignments is defined as $A \sqsubseteq A'$ if for all $v \in V$ it is the case that $A(v) \leq A(v')$. Then by Tarski's fixed-point theorem we know that there must exist a unique minimum pre-fixed-point assignment, A_{min}, and a unique maximum post-fixed-point assignment, A_{max} [10].

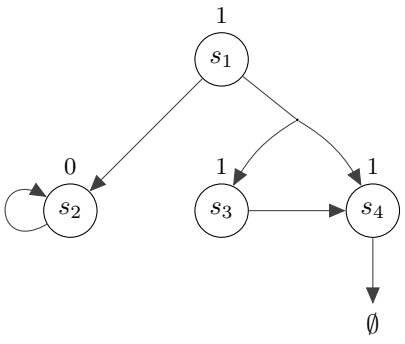


Figure 2.2: Minimum pre-fixed-point assignment.

Now we define F as a function mapping assignments to assignments.

Definition 2.6 - *Assignment Function*

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

Example 2.7

We compute the minimum fixed-point assignment of the dependency graph shown in Figure 2.2.

1. Initially, all vertices have the assignment 0. However, for the hyperedge $(s_4, \emptyset)$ it is trivial that for all targets $s'_4 \in \emptyset$, it is the case that $A_{min}(s'_4) = 1$. Thus $A_{min}(s_4) = 1$.
2. Since $A_{min}(s_4) = 1$, and due to the hyperedge $(s_3, \{s_4\}) \in E$, we have that $A_{min}(s_3) = 1$.
3. Since $A_{min}(s_3) = 1$ and $A_{min}(s_4) = 1$, then due to the hyperedge $(s_1, \{s_3, s_4\}) \in E$, we have that $A_{min}(s_1) = 1$.

We now introduce the notion of *level* on dependency graphs. The level indicates in which iteration of the function F the assignment was improved.

Definition 2.8 - *Dependency Graph Level*

Let (V, E) be a dependency graph. The level of vertex $v \in V$ is $L(v) = n$ if $F^n(v) = 1$ and $F^{n-1}(v) = 0$. By convention, if the assignment of the vertex never changes, then $L(v) = \infty$.

Example 2.9

We show how the level is computed on the dependency graph shown in Figure 2.3. Table 2.1 shows the assignment and level of each vertex in each iteration of F.

F^i	$L(s_1)$	$L(s_2)$	$L(s_3)$	$L(s_4)$	$A(s_1)$	$A(s_2)$	$A(s_3)$	$A(s_4)$
1	0	0	0	1	0	0	0	1
2	0	0	2	1	0	0	1	1
3	3	0	2	1	1	0	1	1
4	3	0	2	1	1	0	1	1

Table 2.1: Assignments and levels.

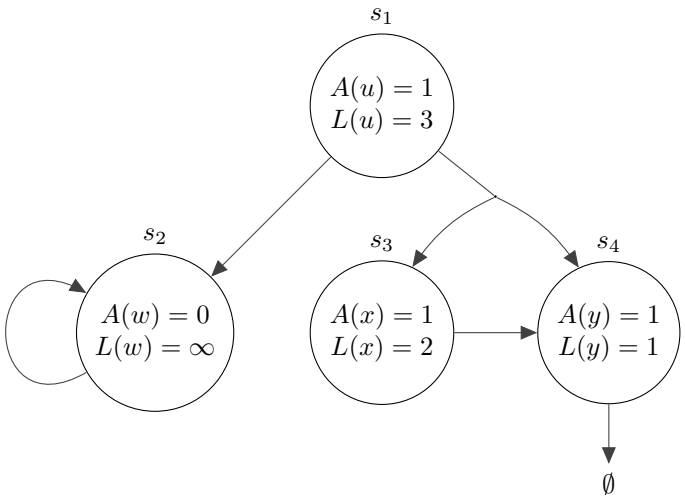


Figure 2.3: Simple dependency graph with level.

2.3 Fixed-Point Algorithm

We now present an on-the-fly algorithm by Xinxin Liu and Scott A.Smolka for computing the minimum pre-fixed-point assignment for a given vertex on a dependency graph [6].

Algorithm 2.1 takes a dependency graph $G = (V, E)$ and a vertex $v \in V$ as input, and computes the minimum pre-fixed-point assignment $A_{min}(v)$.

The algorithm maintains a set W of hyperedges waiting to be processed, which initially contains the hyperedges that have v_0 as the source, and will be expanded as needed. For each $v \in V$ the hyperedges which were processed under the assumption that $A(v) = 0$ is recorded in $D(v)$. The symbol $\perp$ denotes that a vertex has not yet been visited.

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 $A_{min}(v_0)$.

```
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)
```

In each iteration of the while-loop a hyperedge $e = (v, T)$ is selected and removed from W. The algorithm terminates when there are no more hyperedges waiting to be processed (i.e. $W = \emptyset$). There are three cases:

- If $A(v') = 1$ for every vertex v' in the target set T of e, then $A(v) = 1$. Each hyperedge $e' \in D(v)$ must be re-processed since the assumption that $A(v) = 0$ no longer holds. 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 $D(v')$.
- If $A(v') = \bot$ for every vertex v' in the target set T of e, then e is added to $D(v')$ and every hyperedge which has v' as the source is added to W.

Table 2.2 shows the internal state of the algorithm before the i'th iteration of the while-loop when executed on the dependency graph shown in Figure 2.2 and the vertex s_2.

i	W	$A(s_2)$	$A(s_3)$	$D(s_2)$	$D(s_3)$
1	$\{(s_2, \{s_3\})\}$	0	$\bot$	$\emptyset$	$\emptyset$
2	$\{(s_3, \emptyset)\}$	0	0	$\emptyset$	$(s_2, \{s_3\})$
3	$\{(s_2, \{s_3\})\}$	0	1	$\emptyset$	$(s_2, \{s_3\})$
4	$\emptyset$	1	1	$\emptyset$	$(s_2, \{s_3\})$

Table 2.2: Execution of Algorithm 2.1 on Figure 2.2.

Generalized Equivalences and Preorders 3

In this chapter we introduce the notion of a generalized parametric semantic relation inspired by [2] which can be used to define a wide range of equivalences and preorders. We provide examples of concrete instantiations of the parameters for the equivalences and preorders currently supported by CAAL, which are:

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

We then reduce the problem of determining for a pair of states if there exists a generalized parametric semantic relation that relates them, to that of computing the minimum pre-fixed-point assignment on a dependency graph. The minimum pre-fixed-point assignment is computed using the on-the-fly algorithm shown in Algorithm 2.1.

3.1 Game Characterizations

To prove that $s \sim t$, it suffices to find a single binary relation containing the pair (s, t) and proving that it is a strong bisimulation. However, in order to prove the negative case (e.g. $s \not\sim t$) one would have to prove that every distinct binary relation on $\texttt{Proc}$ containing the pair (s, t) is not a strong bisimulation. This approach quickly becomes infeasible as the number of distinct binary relations on an n-element set is 2^{n^2}. To overcome this problem, we provide an alternative definition of strong bisimilarity in terms of a set of rules for a game, which we call a *game characterization*. This not only has the advantage of allowing us to prove the negative case in an easier way, but viewing an equivalence checking problem as a game with a winner and a loser can also provide a better intuitive understanding of the problem, which is beneficial in an educational context.

We now define the rules for the game characterization of strong bisimilarity.

Definition 3.1 - *Strong Bisimulation Game*

Let $(\texttt{Proc}, \texttt{Act}, \rightarrow)$ be an LTS and let $s, t \in \texttt{Proc}$ be states. The game consists of an "attacker" whose goal is to show that $s \not\sim t$, and a "defender" whose goal is to show that $s \sim t$. The game is played over a number of rounds,

where each round starts in a pair of states from $\texttt{Proc} \times \texttt{Proc}$ called the *current configuration*. Initially, the pair (s, t) will be the current configuration.

Each round is played according to the following rules where (s, t) is the current configuration:

1. The attacker must perform a transition $s \xrightarrow{\alpha} s'$ or $t \xrightarrow{\alpha} t'$ where $s', t' \in \texttt{Proc}$ and $\alpha \in \texttt{Act}$. If both $s \not\xrightarrow{}$ and $t \not\xrightarrow{}$ then the defender wins.

2. The defender must now respond with a transition.

   - If the attacker played $s \xrightarrow{\alpha} s'$ then the defender must perform a transition $t \xrightarrow{\alpha} t'$ for some $t' \in \texttt{Proc}$. If $t \not\xrightarrow{\alpha}$ then the attacker wins.
   - If the attacker played $t \xrightarrow{\alpha} t'$ then the defender must perform a transition $s \xrightarrow{\alpha} s'$ for some $s' \in \texttt{Proc}$. If $s \not\xrightarrow{\alpha}$ then the attacker wins.

3. The game continues for another round with the pair (s', t') as the current configuration. If the pair (s', t') has previously been the current configuration the defender wins.

We say that $s \sim t$ if and only if the defender has a *universal winning strategy*. That is, a strategy where the defender always wins, regardless of how the attacker plays. Conversely, we say that $s \not\sim t$ if and only if the attacker has a universal winning strategy.

It should be noted that the defender wins in case of an infinite play. The reasoning behind this is that the attacker has not been able to demonstrate a difference in the two systems.

The above definition can be modified to obtain game characterizations of other equivalences and preorders. For example, in a weak bisimulation game the defender is allowed to respond with weak transitions, and in a strong simulation game the attacker is only allowed to play on one side, while the defender plays on the other side. In a trace equivalence game there is only one round where the attacker starts by playing a sequence of actions on one side, which the defender must then match with the same sequence of actions on the other side.

3.2 Generalized Parametric Semantic Relation

In Section 3.1 we saw that the game characterizations of different equivalences and preorders only differ slightly. We now take advantage of this observation and define a parameterized relation, where the parameters correspond to the rules of the game characterization of the equivalence or preorder that we wish to capture. We will refer to this relation as a *generalized parametric semantic relation*.

The relation has three variable parameters: d, k, and m, where d denotes which side the attacker is allowed to start playing from (left, right, or both), k is the maximum number of rounds in the game, and m is the number of times the attacker is allowed to switch sides. We also have a special fixed parameter E, which is another relation that every pair must also be included in.

Definition 3.2 - *Generalized Parametric Semantic Relation*

Let $(\texttt{Proc}, \texttt{Act}, \rightarrow)$ be an LTS. We fix a set of parameters for a parameterized relation:

- $\texttt{Type} \in \{\mathit{strong}, \mathit{weak}\}$,
- $\texttt{Moves} \in \{\mathit{long}, \mathit{short}\}$,
- $E \subseteq \texttt{Proc} \times \texttt{Proc}$.

We have a set of variables:

- $d \in \{L, R, LR\}$,
- $k \in \mathbb{N}_0^\infty$,
- $m \in \mathbb{N}_0^\infty$.

We define $\overset{w}{\hookrightarrow}$ as

$$\overset{w}{\hookrightarrow} \overset{\text{def}}{=} \begin{cases} \overset{w}{\longrightarrow} & \text{where } |w| = 1 \text{ if } \quad \texttt{Type} = \mathit{strong} \text{ and } \quad \texttt{Moves} = \mathit{short}, \\ \overset{w}{\Rightarrow} & \text{where } |w| = 1 \text{ if } \quad \texttt{Type} = \mathit{weak} \text{ and } \quad \texttt{Moves} = \mathit{short}, \\ \overset{w}{\longrightarrow} & \text{where } |w| \geq 1 \text{ if } \quad \texttt{Type} = \mathit{strong} \text{ and } \quad \texttt{Moves} = \mathit{long}, \\ \overset{w}{\Rightarrow} & \text{where } |w| \geq 1 \text{ if } \quad \texttt{Type} = \mathit{weak} \text{ and } \quad \texttt{Moves} = \mathit{long}. \end{cases}$$

A collection of binary relations over the set of states in an LTS is given by the function

$$\mathcal{R} : \{L, R, LR\} \times \mathbb{N}_0^\infty \times \mathbb{N}_0^\infty \rightarrow \mathcal{P}(\texttt{Proc} \times \texttt{Proc})$$

and is called a *generalized parametric semantic relation* with respect to Type, Moves, and E if for all d, k, m we have that

1. $\mathcal{R}(d, k, m) \subseteq E$

and for all $(s, t) \in \mathcal{R}(d, k, m)$ where $k > 0$ we have that

2. if $d = L$ then
   a) if $s \overset{w}{\hookrightarrow} s'$ then there is a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s', t') \in \mathcal{R}(L, k-1, m)$, and
   b) if $t \overset{w}{\hookrightarrow} t'$ and $m > 0$ then there is a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m-1)$, or
3. if $d = R$ then
   a) if $t \overset{w}{\hookrightarrow} t'$ then there is a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m)$, and
   b) if $s \overset{w}{\hookrightarrow} s'$ and $m > 0$ then there is a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s', t') \in \mathcal{R}(L, k-1, m-1)$, or
4. if $d = LR$ then
   a) if $s \overset{w}{\hookrightarrow} s'$ then there is a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s', t') \in \mathcal{R}(L, k-1, m)$, and
   b) if $t \overset{w}{\hookrightarrow} t'$ then there is a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m)$.

Two states s and t are semantically related written $(s,t) \in \Delta(\texttt{Type}, \texttt{Moves}, E, d, k, m)$ if and only if there is a generalized parametric semantic relation with respect to `Type`, `Moves`, and E such that $(s,t) \in \mathcal{R}(d,k,m)$.

We now show a number of concrete instantiations of the generalized parametric semantic relation for different strong and weak preorders and equivalences from the *linear time - branching time spectrum* by Rob van Glabbeek [13], as shown in Figure 3.1.

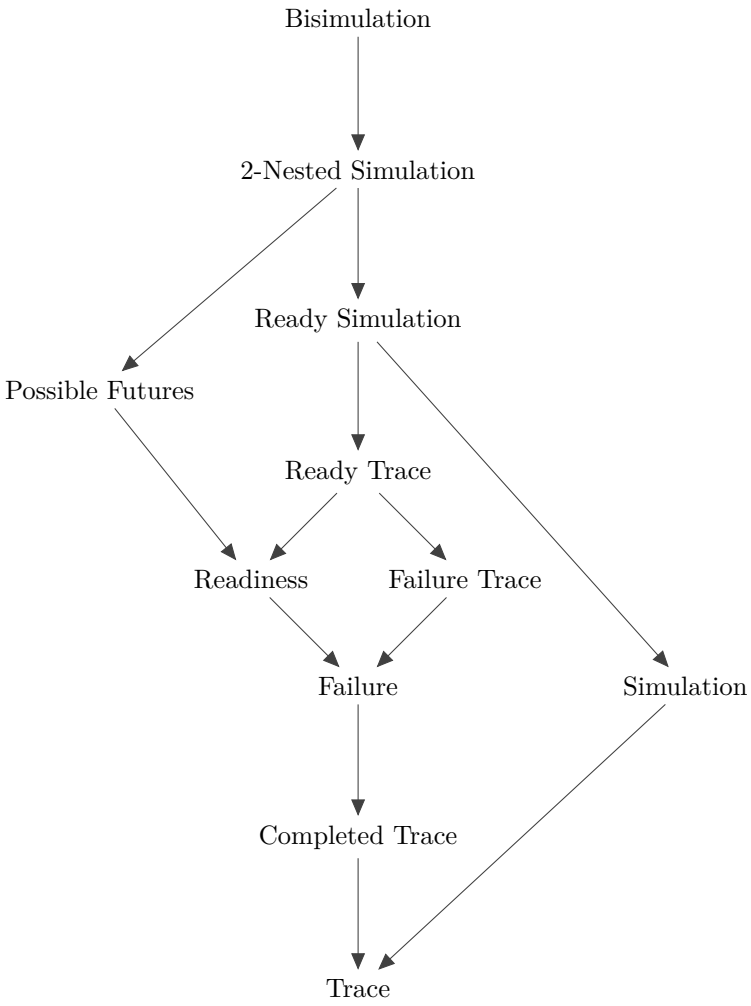


Figure 3.1: The linear time - branching time spectrum.

Simulation
$\Delta(\texttt{Type}, \mathit{short}, \texttt{Proc} \times \texttt{Proc}, L, \infty, 0)$

Simulation Equivalence
$\Delta(\texttt{Type}, \mathit{short}, \texttt{Proc} \times \texttt{Proc}, LR, \infty, 0)$

Bisimulation
$\Delta(\texttt{Type}, \mathit{short}, \texttt{Proc} \times \texttt{Proc}, LR, \infty, \infty)$

Trace Inclusion
$\Delta(\texttt{Type}, \mathit{long}, \texttt{Proc} \times \texttt{Proc}, L, 1, 0)$

Trace Equivalence
$\Delta(\texttt{Type}, \mathit{long}, \texttt{Proc} \times \texttt{Proc}, LR, 1, 0)$

Ready-Trace Equivalence
$\Delta(\texttt{Type}, \mathit{long}, \{(s,t) \mid s \xrightarrow{\alpha} \text{ iff } t \xrightarrow{\alpha} \text{ for all } \alpha \in \texttt{Act}\}, LR, 1, 0)$

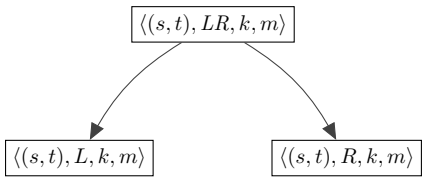
2-Nested Simulation
$\Delta(\texttt{Type}, \mathit{short}, \texttt{Proc} \times \texttt{Proc}, L, 2, 0)$

3.3 Reductions to Dependency Graphs

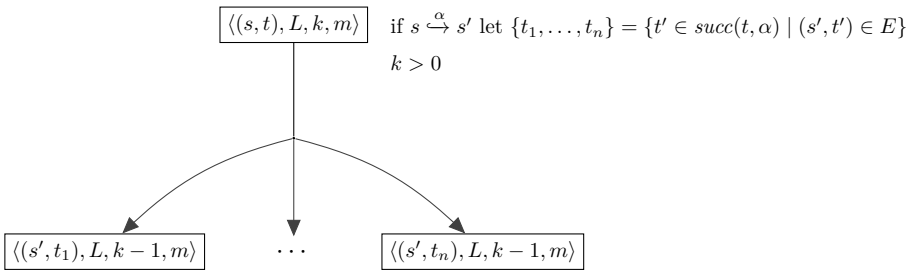
In this section we show reductions from the problem of determining if a pair of states are semantically related to that of computing the minimum pre-fixed-point assignment on a dependency graph. The reductions for *short* and *long* moves are shown in Figures 3.2 and 3.3, respectively.

Definition 3.3 - *Successor Generator*

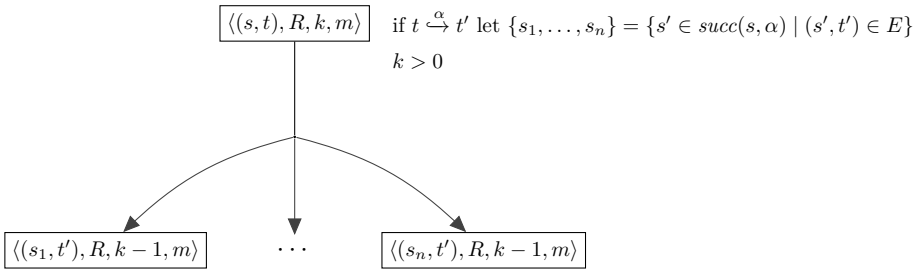
We define the successor generator $\mathit{succ}(s, \alpha) = \{s' \mid s \overset{\alpha}{\hookrightarrow} s'\}$ for a process s and action α.



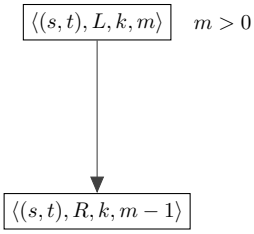
(a) Initial state, both sides.



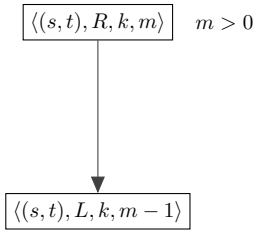
(b) Move on left side.



(c) Move on right side.

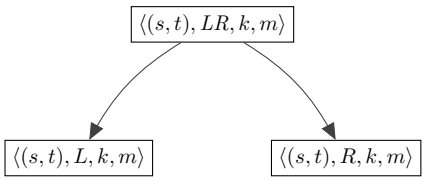


(d) Switch side from left to right.

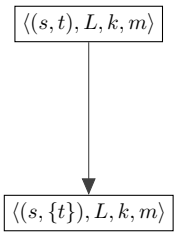


(e) Switch side from right to left.

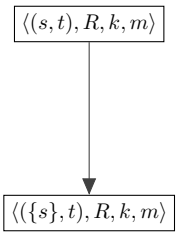
Figure 3.2: Dependency graph construction for *short* moves.



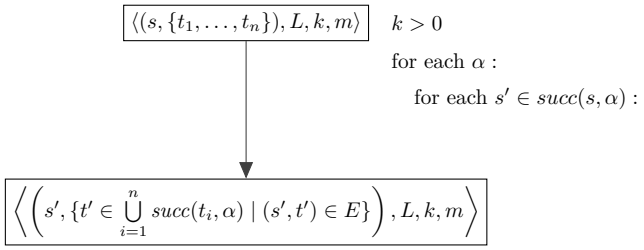
(a) Initial state, both sides.



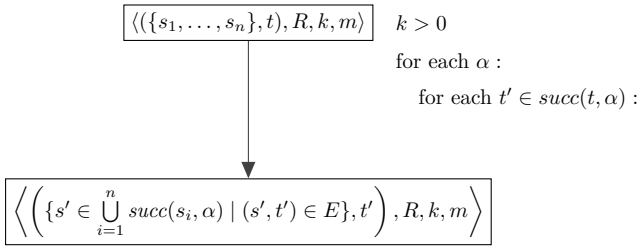
(b) Initial state, left.



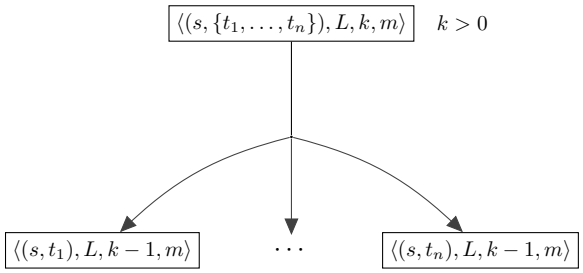
(c) Initial state, right.



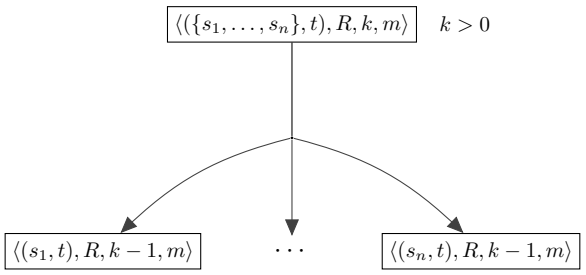
(d) Move on left side.



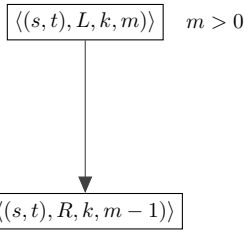
(e) Move on right side.



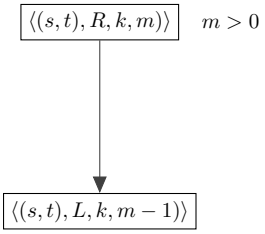
(f) End of round, left side.



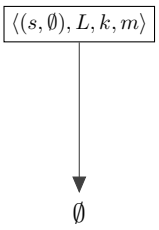
(g) End of round, right side.



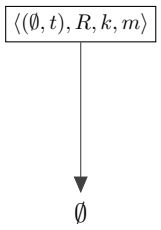
(h) Switch side from left to right.



(i) Switch side from right to left.



(j) Empty set.



(k) Empty set.

Figure 3.3: Dependency graph construction for *long* moves.

Theorem 3.4

We have that $A_{min}(\langle(s,t),d,k,m\rangle) = 0$ on the dependency graphs from Figure 3.2 if and only if $(s,t) \in \Delta(\texttt{Type}, \mathit{short}, E, d, k, m)$.

Proof. We prove both directions of the above statement.

"⇒": We construct a family of semantic relations

$$\mathcal{R}(d,k,m) = \{(s,t) \mid A_{min}(\langle(s,t),d,k,m\rangle) = 0\}.$$

We want to prove that the family of relations $\mathcal{R}(d,k,m)$ is a generalized parametric semantic relation. The conditions in Definition 3.2 must be satisfied. Case 1 must always be satisfied and either Case 2, Case 3, or Case 4 must also be satisfied.

Case 1

From the construction rules in Figure 3.2, we can see that for any new pair of states (s', t') it holds that $(s', t') \in E$.

Case 2

a Let $d = L, k > 0$, and $s \overset{\alpha}{\hookrightarrow} s'$. We have that $A_{min}(\langle (s,t), L, k, m\rangle) = 0$. From Figure 3.2b this implies, that $t \overset{\alpha}{\hookrightarrow} t'$ such that $A_{min}(\langle (s',t'), L, k-1, m\rangle) = 0$. Hence $(s', t') \in \mathcal{R}(d, k-1, m)$.

b Let $d = L, k > 0, m > 0$, and $t \overset{\alpha}{\hookrightarrow} t'$. A transition $t \overset{\alpha}{\hookrightarrow} t'$ is considered a move on the right side, hence a side switch must be made. We have that $A_{min}(\langle (s,t), L, k, m\rangle) = 0$. From Figure 3.2d this implies that $A_{min}(\langle (s,t), R, k, m-1\rangle) = 0$. Next, from Figure 3.2c this implies that $s \overset{\alpha}{\hookrightarrow} s'$ for some s' such that $A_{min}(\langle (s',t'), R, k-1, m-1\rangle) = 0$. Hence $(s', t') \in \mathcal{R}(d, k-1, m-1)$.

Case 3

a Symmetrical to Case 2a.

b Symmetrical to Case 2b.

Case 4

a Let $d = LR, k > 0$, and $s \overset{\alpha}{\hookrightarrow} s'$. We have that $A_{min}(\langle (s,t), LR, k, m\rangle) = 0$. From Figure 3.2a this implies that $A_{min}(\langle (s,t), L, k, m\rangle) = 0$ and $A_{min}(\langle (s,t), R, k, m\rangle) = 0$. Since $A_{min}(\langle (s,t), L, k, m\rangle) = 0$, we have from Figure 3.2b that $t \overset{\alpha}{\hookrightarrow} t'$ such that $A_{min}(\langle (s',t'), L, k-1, m\rangle) = 0$. Hence $(s', t') \in \mathcal{R}(d, k-1, m)$.

b Symmetrical to Case 4a.

"$\Leftarrow$": From Definition 3.2 we have that if $(s,t) \in \Delta(\texttt{Type}, \mathit{short}, E, d, k, m)$ then there exists a generalized parametric semantic relation $\mathcal{R}(d, k, m)$ such that $(s,t) \in \mathcal{R}(d, k, m)$. We define an assignment

$$A(\langle (s,t), d, k, m\rangle) = \begin{cases} 0 & \text{if } (s,t) \in \mathcal{R}(d,k,m) \\ 1 & \text{otherwise.} \end{cases}$$

We prove that $A(\langle (s,t), d, k, m\rangle)$ is a pre-fixed-point assignment by showing that none of the rules in Figure 3.2 can improve the assignment. For each rule we show that every hyperedge from the root has at least one vertex in its target set where $A(\langle (s,t), d, k, m\rangle) = 0$.

Figure 3.2a

Assume that $A(\langle (s,t), LR, k, m\rangle) = 0$ which implies that $(s,t) \in \mathcal{R}(LR, k, m)$. We want to show that $(s,t) \in \mathcal{R}(L, k, m)$ and $(s,t) \in \mathcal{R}(R, k, m)$, which implies that $A(\langle (s,t), L, k, m\rangle) = 0$ and $A(\langle (s,t), R, k, m\rangle) = 0$. To show this, case 2 and case 3 from Definition 3.2 must be satisfied.

Case 2a
Let $s \overset{w}{\hookrightarrow} s'$. We want to find a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s', t') \in \mathcal{R}(L, k-1, m)$. Because $(s, t) \in \mathcal{R}(LR, k, m)$, then due to case 4a there is a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s', t') \in \mathcal{R}(L, k-1, m)$.

Case 2b
Let $t \overset{w}{\hookrightarrow} t'$. We want to find a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m-1)$. Because $(s, t) \in \mathcal{R}(LR, k, m)$, then due to case 4b there is a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m)$. This trivially implies that $(s', t') \in \mathcal{R}(R, k-1, m-1)$.

Case 3a
Symmetrical to Case 2a.

Case 3b
Symmetrical to Case 2b.

Since both cases are satisfied we have that $(s, t) \in \mathcal{R}(L, k, m)$ and $(s, t) \in \mathcal{R}(R, k, m)$, which implies that $A(\langle (s, t), L, k, m \rangle) = 0$ and $A(\langle (s, t), R, k, m \rangle) = 0$. Hence the assignment $A(\langle (s, t), LR, k, m \rangle) = 0$ cannot be improved.

Figure 3.2b
Assume that $A(\langle (s, t), L, k, m \rangle) = 0$ which implies that $(s, t) \in \mathcal{R}(L, k, m)$. From Definition 3.2 we have that if $d = L$ then whenever $s \overset{w}{\hookrightarrow} s'$ there must exist a state t' such that $t \overset{w}{\hookrightarrow} t'$ and $(s', t') \in \mathcal{R}(L, k-1, m)$ which implies that $A(\langle (s', t'), L, k-1, m \rangle) = 0$. Hence the assignment $A(\langle (s, t), L, k, m \rangle) = 0$ cannot be improved.

Figure 3.2c
Symmetrical to Figure 3.2b.

Figure 3.2d
Assume that $A(\langle (s, t), L, k, m \rangle) = 0$ which implies that $(s, t) \in \mathcal{R}(L, k, m)$. We want to show that $(s, t) \in \mathcal{R}(R, k, m-1)$ which implies that $A(\langle (s, t), R, k, m-1 \rangle) = 0$. To show this, case 3a and case 3b from Definition 3.2 must be satisfied.

Case 3a
Let $t \overset{w}{\hookrightarrow} t'$, we want to find a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m-1)$. Because $(s, t) \in \mathcal{R}(L, k, m)$, then due to case 2b there is a transition $s \overset{w}{\hookrightarrow} s'$ such that $(s', t') \in \mathcal{R}(R, k-1, m-1)$.

Case 3b
Let $s \overset{w}{\hookrightarrow} s'$, we want to find a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s', t') \in \mathcal{R}(L, k-1, m-2)$. Because $(s, t) \in \mathcal{R}(L, k, m)$, then due to case 2a there is a transition $t \overset{w}{\hookrightarrow} t'$ such that $(s, t) \in \mathcal{R}(L, k-1, m)$. This trivially implies that $(s, t) \in \mathcal{R}(L, k-1, m-2)$.

Since both cases are satisfied we have that $(s, t) \in \mathcal{R}(R, k, m-1)$ which implies that $A(\langle (s, t), R, k, m-1 \rangle) = 0$. Hence the assignment $A(\langle (s, t), L, k, m \rangle) = 0$ cannot be improved.

Figure 3.2e
Symmetrical to Figure 3.2d.

□

Theorem 3.5

We have that $A_{min}(\langle(s,t),d,k,m\rangle) = 0$ on the dependency graphs from Figure 3.3 if and only if $(s,t) \in \Delta(\texttt{Type}, long, E, d, k, m)$.

Proof. We prove both directions of the above statement.

"⇒": We construct a family of semantic relations

$$\mathcal{R}(d,k,m) = \{(s,t) \mid A_{min}(\langle(s,t),d,k,m\rangle) = 0\}.$$

We want to prove that the family of relations $\mathcal{R}(d,k,m)$ is a generalized parametric semantic relation. The conditions in Definition 3.2 must be satisfied. Case 1 must always be satisfied and either Case 2, Case 3, or Case 4 must also be satisfied.

Case 1
From the construction rules in Figure 3.3, we can see that for any new pair of states (s',t') it holds that $(s',t') \in E$.

Case 2

a Let $d = L, k > 0$, and $s \overset{w}{\hookrightarrow} s'$. We have that $A_{min}(\langle(s,t),L,k,m\rangle) = 0$. We start by applying the rule shown in Figure 3.3b. Since $A_{min}(\langle(s,t),L,k,m\rangle) = 0$ we now have that $A_{min}(\langle(s,\{t\}),L,k,m\rangle) = 0$. Next, we make $|w|$ number of moves on the left side using the rule shown in Figure 3.2b such that the chosen successors form the sequence w ending up in the node $\langle(s',\{t_1,\ldots,t_n\}),L,k,m\rangle$. Because we must have $A_{min}(\langle(s',\{t_1,\ldots,t_n\}),L,k,m\rangle) = 0$ it is implied that there is at least one transition $t \overset{w}{\hookrightarrow} t_i$ such that $A_{min}(\langle(s',t_i),L,k-1,m\rangle) = 0$. Hence $(s',t_i) \in \mathcal{R}(L,k-1,m)$.

b Let $d = L, k > 0, m > 0$, and $t \overset{w}{\hookrightarrow} t'$. A transition $t \overset{w}{\hookrightarrow} t'$ is considered a move on the right side, hence a side switch must be made. We have that $A_{min}(\langle(s,t),L,k,m\rangle) = 0$, and from the rule shown in Figure 3.3h it is implied that $A_{min}(\langle(s,t),R,k,m-1\rangle) = 0$. The rest of the proof is symmetrical to that of Case 2a.

Case 3

a Symmetrical to Case 2a.

b Symmetrical to Case 2b.

Case 4

a Let $d = LR, k > 0$, and $s \overset{w}{\hookrightarrow} s'$. We have that $A_{min}(\langle(s,t), LR, k, m\rangle) = 0$. From the rule shown in Figure 3.3a it is implied that $A_{min}(\langle(s,t), L, k, m\rangle) = 0$ and $A_{min}(\langle(s,t), R, k, m\rangle) = 0$. The rest of the proof is identical to that of Case 2a.

b Symmetrical to Case 4a.

"$\Leftarrow$": From Definition 3.2 we have that if $(s,t) \in \Delta(\texttt{Type}, long, E, d, k, m)$ then there exists a generalized parametric semantic relation $\mathcal{R}(d,k,m)$ such that $(s,t) \in \mathcal{R}(d,k,m)$. We define an assignment

$$A(\langle(s,t), d, k, m\rangle) = \begin{cases} 0 & \text{if } (s,t) \in \mathcal{R}(d,k,m) \\ 1 & \text{otherwise} \end{cases}$$

$$A(\langle(\{s_1, \ldots, s_n\}, t), d, k, m\rangle) = \begin{cases} 0 & \text{if } (s_i,t) \in \mathcal{R}(d,k,m) \text{ for some } 1 \leq i \leq n \\ 1 & \text{otherwise} \end{cases}$$

$$A(\langle(s, \{t_1, \ldots, t_n\}), d, k, m\rangle) = \begin{cases} 0 & \text{if } (s,t_i) \in \mathcal{R}(d,k,m) \text{ for some } 1 \leq i \leq n \\ 1 & \text{otherwise.} \end{cases}$$

We prove that $A(\langle(s,t), d, k, m\rangle)$ is a pre-fixed-point assignment by showing that none of the rules in Figure 3.3 can improve the assignment. For each rule we show that every hyperedge from the root has at least one vertex in its target set where $A(\langle(s,t), d, k, m\rangle) = 0$.

Figure 3.3a
Identical to the proof of Figure 3.2a in Theorem 3.4.

Figures 3.3b, 3.3d, and 3.3f
We combine several rules in order to perform a single *long* move on the left side. We start with a single application of Figure 3.3b, followed by a sequence of applications of Figure 3.3d, where each such sequence can only end with a single application of either Figure 3.3f or Figure 3.3j.

Assume that $A(\langle(s,t), L, k, m\rangle) = 0$ which implies that $(s,t) \in \mathcal{R}(L,k,m)$. From Definition 3.2 we have that because $(s,t) \in \mathcal{R}(L, k, m)$ then whenever $s \overset{w}{\hookrightarrow} s'$ there must exist a state t' such that $t \overset{w}{\hookrightarrow} t'$ and $(s',t') \in \mathcal{R}(L, k-1, m)$. This means that for each application of Figure 3.3d the set $\{t_1, \ldots, t_n\}$ will always be non-empty, and thus every sequence of applications of Figure 3.3d will eventually be followed by a single application of Figure 3.3f (i.e Figure 3.3j will never be applied). This implies that the set $\{t_1, \ldots, t_n\}$ in Figure 3.3f must contain a state t_i such that $(s', t_i) \in \mathcal{R}(L, k-1, m)$ and thereby $A(\langle(s', t_i), L, k-1, m\rangle) = 0$. Hence the assignment $A(\langle(s,t), L, k, m\rangle) = 0$ cannot be improved.

Figures 3.3c, 3.3e, and 3.3g
Symmetrical to Figures 3.3b, 3.3d, and 3.3f.

Figure 3.3h
Identical to the proof of Figure 3.2d in Theorem 3.4.

Figure 3.3i
Symmetrical to Figure 3.3h.

Figure 3.3j
Assume that $A(\langle(s,\emptyset),L,k,m\rangle) = 1$ which implies that $(s,t) \notin \mathcal{R}(L,k,m)$. Clearly the assignment $A(\langle(s,\emptyset),L,k,m\rangle) = 1$ will never change since there is only a single hyperedge going to the empty set.

Figure 3.3k
Symmetrical to Figure 3.3j.

□

As a result of Theorem 3.4 and Theorem 3.5, we are now able to state the main result.

Corollary 3.6

We have that $A_{min}(\langle(s,t),d,k,m\rangle) = 0$ if and only if $(s,t) \in \Delta(\texttt{Type},\texttt{Move},E,d,k,m)$.

We now demonstrate how to use the rules from Figure 3.2 and Figure 3.3 on the LTS given in Figure 3.4. We give examples of strong bisimulation and strong trace equivalence between the two states s and t. When the dependency graph has been created it can be determined if the two states are semantically related by computing the minimum pre-fixed-point assignment.

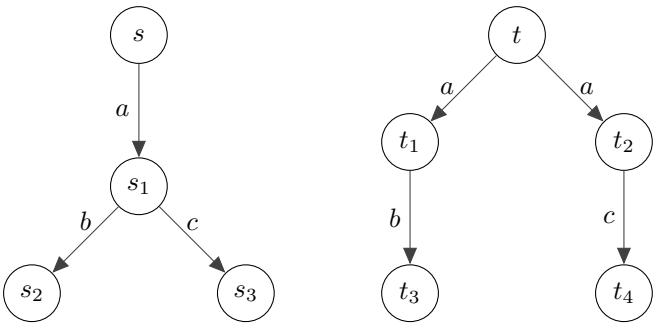


Figure 3.4: Two states s and t.

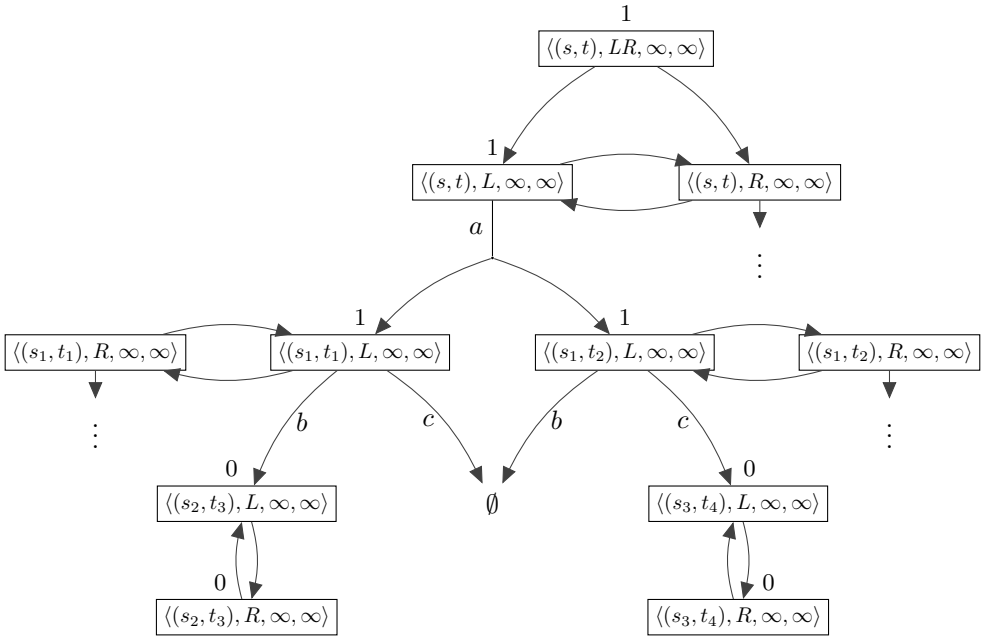


Figure 3.5: Dependency graph for bisimulation.

Example 3.7

We construct a partial dependency graph in Figure 3.5 from the procsseses s and t in Figure 3.4, using the rules from Figure 3.2 with the generalized parametric semantic relation for strong bisimulation, $\Delta(\mathit{strong}, \mathit{short}, \texttt{Proc} \times \texttt{Proc}, LR, \infty, \infty)$. We can establish that the two processes are not strongly bisimilar since the root node has the minimum pre-fixed-point assignment 1.

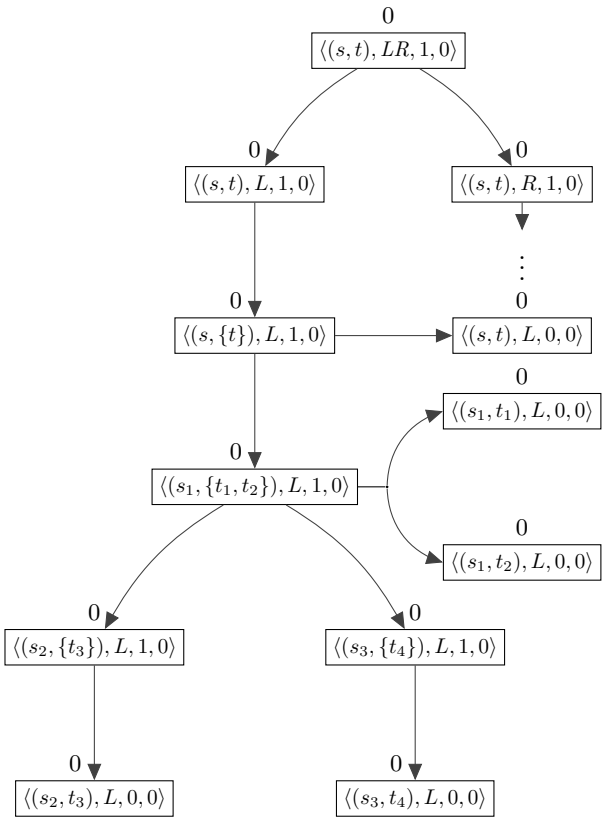


Figure 3.6: Dependency graph for trace equivalence.

Example 3.8

We construct a partial dependency graph in Figure 3.6 from the processes s and t in Figure 3.4, using the rules from Figure 3.3 with the generalized parametric semantic relation for strong trace equivalence, $\Delta(\mathit{strong}, \mathit{long}, \texttt{Proc} \times \texttt{Proc}, LR, 1, 0)$. We can establish that the two processes are strong trace equivalent since the root node has the minimum pre-fixed-point assignment 0.

Timed CCS

4

In this chapter we introduce the timed process algebra Timed CCS (TCCS) introduced by Wang Yi [16]. TCCS is an extension of CCS with only one new syntactic element, the *delay prefixing* operator.

There are two different ways of viewing the flow of time. There is continuous time with delays in the set $\mathbb{R}_{\leq 0}$ of non-negative real numbers as the time domain, and there is discrete time with delays in the set $\mathbb{N}$ of natural numbers as the time domain.

Time values in the discrete time domain are distinct points in time. A discrete time value jumps from one value to another without distinguishing what happened in between the values. Continuous time is a more natural way of viewing the flow of time; between any two time values, there are an infinite number of other time values.

We will restrict ourselves to the discrete time domain since discrete time is easier to implement and visualize. Moreover, we will define our semantics to use delays of 1 time unit only.

4.1 The Language TCCS

We still have all the syntactical elements as CCS, but we add the delay prefix operator $\varepsilon(d)$ to the syntax. With delays we can model processes like

$$\varepsilon(5).a.0\,,$$

which means that after 5 time units we can perform a, and then become 0.

Recall the example in Section 1.1.2 where we had a process Driver which we modelled as

$$\text{Driver} \stackrel{\text{def}}{=} \text{drive.Driver}\ .$$

At some unfortunate point in time the driver might crash his car, which we now model as

$$\text{Driver} \stackrel{\text{def}}{=} \text{drive.Driver} + \text{drive.Crash}\ .$$

Before we model Crash, let us define an airbag to protect the driver from the crash:

$$\text{Airbag} \stackrel{\text{def}}{=} \text{crash}.\varepsilon(1).\overline{\text{inflate}}.\text{Airbag}\ ,$$

which means that after a crash, the airbag can inflate after 1 unit of time has passed, but nothing is forcing it do so. Let us now model the remaining processes, Crash and Impl:

$$\text{Crash} \stackrel{\text{def}}{=} \overline{\text{crash}}.(\text{inflate.Driver} + \varepsilon(2).\tau.0)\ ,$$

$$\text{Impl} \stackrel{\text{def}}{=} (\text{Driver} \mid \text{Airbag}) \setminus \{\text{crash}, \text{inflate}\}\ .$$

The Driver and the Airbag processes are now running in parallel, and since the channels crash and inflate are restricted, they are forced to communicate. A TCCS process cannot delay when a τ-action is available, introducing a sense

of urgency. This means that 1 time unit after the crash has happened, the two processes are forced to communicate on the inflate channel, thus producing a τ-action which prevents further delaying. This prevention of delay makes it impossible for the Crash prosses to reach the 0 process, and hereby saving the driver.

It is of course also possible to make a bad airbag which will not inflate in time:

$$\text{BadAirbag} \stackrel{\text{def}}{=} \text{crash}.\varepsilon(3).\tau.\overline{\text{inflate}}.\text{Airbag}\ .$$

4.2 Syntax and Semantics

We model TCCS processes using Timed LTSs (TLTSs). A TLTS consists of a set of states (or processes), a set of labels, and a transition relation. If a process q can do a delay of 1 and become q' it is written as $q \xrightarrow{1} q'$.

Definition 4.1 - *Timed Labelled Transition System*

A TLTS is a triple $(\texttt{Proc}, \texttt{Lab}, \rightarrow)$ where $\texttt{Proc}$ is a set of states, $\texttt{Lab}$ is a set of actions and delays $\texttt{Act} \cup \{1\}$, and $\rightarrow \subseteq \texttt{Proc} \times \texttt{Lab} \times \texttt{Proc}$ is the transition relation.

We extend the syntax of CCS with the delay-prefixing operator $\varepsilon(d)$ and give the formal semantics of this operator. Our semantics uses discrete delays of 1 time unit only, however we allow delays of arbitrary length in the syntax.

Definition 4.2 - *TCCS Syntax*

The syntax of TCCS is a direct extension of Definition 1.12. The syntax is extended with a single operator:

$$\varepsilon(d).P$$

where $d \in \mathbb{N}_0$ is a time delay.

We extend the equivalence rules from Table 1.2 with the rules

$$\varepsilon(0).P \equiv P\,,$$
$$\varepsilon(d).\varepsilon(d').P \equiv \varepsilon(d+d').P\,.$$

We also extend the congruence rules from Table 1.3 with the rule

$$\frac{P \equiv Q}{\varepsilon(d).P \equiv \varepsilon(d).Q}$$

where P and Q are processes.

We now give the formal semantics of TCCS with only 1-delays. The SOS rules for CCS still apply.

Definition 4.3 - *TCCS Semantics*

The semantics of TCCS is a direct extension of Definition 1.13. The semantics is extended with the following SOS rules:

$$\text{DEL}_1 \ \frac{}{\varepsilon(d).P \xrightarrow{1} \varepsilon(d-1).P} \text{ if } d \geq 1 \qquad \text{DEL}_{\text{SUM}} \ \frac{P_i \xrightarrow{1} P_i' \text{ for each } i \in I}{\left(\sum_{i\in I} P_i\right) \xrightarrow{1} \left(\sum_{i\in I} P_i'\right)}$$

$$\text{DEL}_2 \ \frac{P \xrightarrow{1} P'}{K \xrightarrow{1} P'} \ K \stackrel{\text{def}}{=} P \qquad \text{DEL}_{\text{REL}} \ \frac{P \xrightarrow{1} P'}{P[f] \xrightarrow{1} P'[f]}$$

$$\text{DEL}_3 \ \frac{}{\alpha.P \xrightarrow{1} \alpha.P} \text{ for } \alpha \neq \tau \qquad \text{DEL}_{\text{REC}} \ \frac{P \xrightarrow{1} P'}{P \setminus L \xrightarrow{1} P' \setminus L}$$

$$\text{DEL}_{\text{COM}} \ \frac{P_i \xrightarrow{1} P_i' \text{ for each } i \in I}{\left(\Big|_{i\in I} P_i\right) \xrightarrow{1} \left(\Big|_{i\in I} P_i'\right)} \text{ if } \left(\Big|_{i\in I} P_i\right) \stackrel{\tau}{\not\to}$$

We now show some examples of the SOS rules in use. Consider the process

$$P \equiv a.0\,.$$

This process can perform the transitions $P \xrightarrow{a} 0$ by the ACT rule and $P \xrightarrow{1} a.0$ by the DEL_3 rule. Now consider the process

$$Q \equiv a.0 + \varepsilon(2).b.0$$

which can perform the transitions $Q \xrightarrow{a} 0$ by the SUM and ACT rules and $Q \xrightarrow{1} a.0 + \varepsilon(1).b.0$ by the DEL_{SUM} and DEL_1 rules. The process

$$R \equiv \tau.0 + \varepsilon(2).b.0$$

can only perform the transition $R \xrightarrow{\tau} 0$ since the DEL_3 rule prohibits delays when a τ-transition is available, so we are forced to commit to the left-hand side. The same applies for parallel composition where a composition cannot delay if it can perform a τ-transition. Consider the process

$$S \equiv \varepsilon(2).b.0 \mid a.0 \mid \overline{a}.0$$

which cannot delay since it is possible for the a and $\overline{a}$ to handshake and thereby perform a τ-transition by the COM_2 rule.

4.3 Equivalences and Preorders

This section describes notions of behavioral equivalences and preorders between processes in terms of their TLTSs.

Sometimes we want to abstract away from time. If we have the processes $\varepsilon(2).a.0$ and $a.0$ where we might not care about the time, we would expect these two processes to have the same behavior. In Definition 4.4 we define the untimed transition which abstracts away from time.

Definition 4.4 - *Untimed Transition*

Let s and t be two states in a TLTS. For each label $\iota \in \mathtt{Lab}$, we write $s \xrightarrow{\iota}_u t$ if and only if either

- $\iota \neq 1$ and there are processes s' and t' such that
$$s \left(\xrightarrow{1}\right)^* s' \xrightarrow{\iota} t' \left(\xrightarrow{1}\right)^* t$$
- or $\iota = 1$ and $s \left(\xrightarrow{1}\right)^* t$.

For CCS we have a weak transition. The weak transition from Definition 1.2 also applies to TCCS except that we use a TLTS instead of an LTS. This allows us to perform a $\stackrel{1}{\Rightarrow}$ transition which abstracts away from τ.

It can also be useful to abstract away from both τ and time. Naturally, we do this with a weak untimed transition defined in Definition 4.5.

Definition 4.5 - *Untimed Weak Transition*

Let s and t be two states in a TLTS. For each label $\iota \in \mathtt{Lab}$, we write $s \stackrel{\iota}{\Rightarrow}_u t$ if and only if either

- $\iota \notin \{\tau, 1\}$ and there are processes s' and t' such that
$$s \left(\xrightarrow{\tau} \cup \xrightarrow{1}\right)^* s' \xrightarrow{\iota} t' \left(\xrightarrow{\tau} \cup \xrightarrow{1}\right)^* t$$
- or $\iota \in \{\tau, 1\}$ and $s \left(\xrightarrow{\tau} \cup \xrightarrow{1}\right)^* t$.

We can now continue to define the different timed and untimed versions of the equivalences and preorders from Section 1.1.1.

In what follows, we use $\hookrightarrow$ to denote either the strong timed transition relation $\rightarrow$, the strong untimed transition relation $\rightarrow_u$, the weak timed transition relation $\Rightarrow$, or the weak untimed transition relation $\Rightarrow_u$.

Definition 4.6 - *Simulation*

A binary relation $\mathcal{R}$ over the set of states of a TLTS is a simulation if and only if whenever $(s_1, s_2) \in \mathcal{R}$, and $\iota \in \mathtt{Lab}$:

If $s_1 \xrightarrow{\iota} s_1'$ then there is a transition $s_2 \stackrel{\iota}{\hookrightarrow} s_2'$ such that $(s_1', s_2') \in \mathcal{R}$.

A state s is said to *simulate* a state t if and only if there is a simulation that relates them. From now on the relation $\sqsubseteq_{\sim t}$ will be referred to as *strong timed simulation* when $\hookrightarrow = \rightarrow$, $\sqsubseteq_{\sim u}$ will be referred to as *strong untimed simulation* when $\hookrightarrow = \rightarrow_u$, $\sqsubseteq_{\approx t}$ will be referred to as *weak timed simulation* when $\hookrightarrow = \Rightarrow$, and $\sqsubseteq_{\approx u}$ will be referred to as *weak untimed simulation* when $\hookrightarrow = \Rightarrow_u$.

Having defined simulation we can now use this to define simulation equivalence in Definition 4.7 which is the same as having simulation preorder in both directions.

Definition 4.7 - *Simulation Equivalence*

Two states s and t are *strong timed simulation equivalent* if and only if $s \sqsubseteq_{\sim t} t$ and $t \sqsubseteq_{\sim t} s$. s and t are *strong untimed simulation equivalent* if and only if $s \sqsubseteq_{\sim u} t$ and $t \sqsubseteq_{\sim u} s$. s and t are *weak timed simulation equivalent* if and only if $s \sqsubseteq_{\approx t} t$ and $t \sqsubseteq_{\approx t} s$. s and t are *weak untimed simulation equivalent* if and only if $s \sqsubseteq_{\approx u} t$ and $t \sqsubseteq_{\approx u} s$.

From now on the relation $\simeq_t$ will be referred to as *strong timed simulation equivalence*, $\simeq_u$ will be referred to as *strong untimed simulation equivalence*, $\approxeq_t$ will be referred to as *weak timed simulation equivalence*, and $\approxeq_u$ will be referred to as *weak untimed simulation equivalence*.

Consider untimed strong bisimulation in Definition 4.8 as it appears in the book Reactive Systems [1]. This definition for untimed strong bisimulation does not abstract away from time in the natural way that one might expect, because delays must be matched by delays and actions must be matched by actions.

Definition 4.8 - *Untimed Strong Bisimulation 1*

A binary relation $\mathcal{R}$ over the set of states of a TLTS is a strong untimed bisimulation if and only if whenever $(s_1, s_2) \in \mathcal{R}$, and α is an action and d is a delay:

if $s_1 \xrightarrow{\alpha} s_1'$, then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $(s_1', s_2') \in \mathcal{R}$,

if $s_1 \xrightarrow{1} s_1'$, then there is a transition $s_2 \left(\xrightarrow{1}\right)^* s_2'$ such that $(s_1', s_2') \in \mathcal{R}$,

if $s_2 \xrightarrow{\alpha} s_2'$, then there is a transition $s_1 \xrightarrow{\alpha} s_1'$ such that $(s_1', s_2') \in \mathcal{R}$,

if $s_2 \xrightarrow{1} s_2'$, then there is a transition $s_1 \left(\xrightarrow{1}\right)^* s_1'$ such that $(s_1', s_2') \in \mathcal{R}$.

Two states s and t are *strongly untimed bisimilar* if and only if there is a strong untimed bisimulation the relates them.

Instead, we define the strong/weak timed/untimed bisimulation in Definition 4.9 where the untimed bisimulations uses the untimed transitions from Definition 4.4 and Definition 4.5.

Definition 4.9 - *Bisimulation 2*

A binary relation $\mathcal{R}$ over the set of states of a TLTS is a bisimulation if and only if whenever $(s_1, s_2) \in \mathcal{R}$, and $\iota \in \mathtt{Lab}$:

if $s_1 \overset{\iota}{\hookrightarrow} s_1'$, then there is a transition $s_2 \overset{\iota}{\hookrightarrow} s_2'$ such that $(s_1', s_2') \in \mathcal{R}$,

if $s_2 \overset{\iota}{\hookrightarrow} s_2'$, then there is a transition $s_1 \overset{\iota}{\hookrightarrow} s_1'$ such that $(s_1', s_2') \in \mathcal{R}$.

Two states s and t are *bisimilar* if and only if there is a bisimulation that relates them. From now on the relation $\sim_t$ will be referred to as *strong timed simulation* when $\hookrightarrow = \rightarrow$, $\sim_u$ will be referred to as *strong untimed simulation* when $\hookrightarrow = \rightarrow_u$, $\approx_t$ will be referred to as *weak timed simulation* when $\hookrightarrow = \Rightarrow$, and $\approx_u$ will be referred to as *weak timed simulation* when $\hookrightarrow = \Rightarrow_u$.

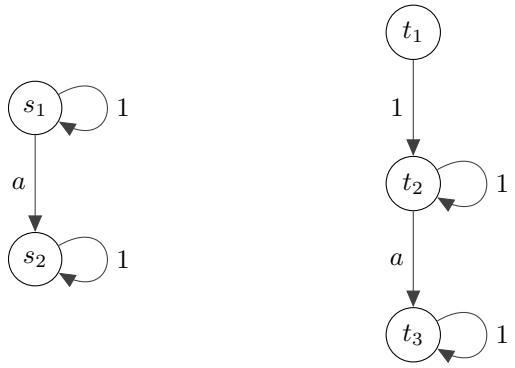


Figure 4.1: States s_1 and t_1 are untimed bisimilar.

Example 4.10

Figure 4.1 shows two TLTSs where the states s_1 and t_1 are untimed bisimilar.

States s_1 and t_1 are not timed bisimilar because of the transition $s_1 \overset{a}{\rightarrow} s_2$ and $t_1 \overset{a}{\not\rightarrow}$.

According to Definition 4.8 s_1 and t_1 are not strongly untimed bisimilar because $s_1 \overset{a}{\rightarrow} s_2$ and $t_1 \overset{a}{\not\rightarrow}$.

According to strong untimed bisimilarity from Definition 4.9 we have that $s_1 \sim_u t_1$ since we abstract away from delay. Both s_1 and t_1 can perform $\overset{a}{\rightarrow}_u$. To show that $s_1 \sim_u t_1$, here is an untimed bisimulation $\mathcal{R}$:

$$\mathcal{R} = \{(s_1, t_1), (s_1, t_2), (s_2, t_3)\}$$

The definition for timed traces and untimed traces are separated, because we can define the untimed traces more explicit than timed traces. By Definition 4.11 we say that timed traces consist of sequences of actions and 1-delays,

whereas by Definition 4.12 we say that untimed traces consists only of sequences of actions because we abstract away from time with the $\rightarrow_u$ transitions.

Definition 4.11 - *Timed Traces*

A strong timed trace from a state s is a sequence of $\iota_1 \cdots \iota_n \in \mathtt{Lab}^*$ where $n \geq 0$ such that there exists a sequence of strong transitions

$$s_0 \xrightarrow{\iota_1} s_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_{n-1}} s_{n-1} \xrightarrow{\iota_n} s_n,$$

for some states $s_1, \ldots, s_n$.

A weak timed trace from a state s is a sequence of $\iota_1 \cdots \iota_n \in (\mathtt{Lab} \setminus \{\tau\})^*$ where $n \geq 0$ such that there exists a sequence of weak transitions

$$s_0 \stackrel{\iota_1}{\Rightarrow} s_1 \stackrel{\iota_2}{\Rightarrow} \ldots \stackrel{\iota_{n-1}}{\Longrightarrow} s_{n-1} \stackrel{\iota_n}{\Rightarrow} s_n,$$

for some states $s_1, \ldots, s_n$.

We write $\mathit{Traces}_{\rightarrow}(s)$ for the collection of all strong timed traces of s, and $\mathit{Traces}_{\Rightarrow}(s)$ for the collection of all weak timed traces of s.

Definition 4.12 - *Untimed Traces*

A strong untimed trace from a state s is a sequence of $\alpha_1 \cdots \alpha_n \in \mathtt{Act}^*$ where $n \geq 0$ such that there exists a sequence of strong untimed transitions

$$s_0 \xrightarrow{\alpha_1}_u s_1 \xrightarrow{\alpha_2}_u \ldots \xrightarrow{\alpha_{n-1}}_u s_{n-1} \xrightarrow{\alpha_n}_u s_n,$$

for some states $s_1, \ldots, s_n$.

A weak untimed trace from a state s is a sequence of $\alpha_1 \cdots \alpha_n \in (\mathtt{Act} \setminus \{\tau\})^*$ where $n \geq 0$ such that there exists a sequence of strong untimed transitions

$$s_0 \stackrel{\alpha_1}{\Rightarrow}_u s_1 \stackrel{\alpha_2}{\Rightarrow}_u \ldots \stackrel{\alpha_{n-1}}{\Longrightarrow}_u s_{n-1} \stackrel{\alpha_n}{\Rightarrow}_u s_n,$$

for some states $s_1, \ldots, s_n$.

We write $\mathit{Traces}_{\rightarrow_u}(s)$ for the collection of all strong untimed traces of s, and $\mathit{Traces}_{\Rightarrow_u}(s)$ for the collection of all weak untimed traces of s.

Having defined timed and untimed traces we can now define trace inclusion in Definition 4.13.

Definition 4.13 - *Trace Inclusion*

Process s is said to include the traces of process t if and only if

$$\mathit{Traces}_{\hookrightarrow}(s) \subseteq \mathit{Traces}_{\hookrightarrow}(t).$$

From now on the relation $\subseteq_t$ will be referred to as *strong timed trace inclusion* when $\hookrightarrow = \rightarrow$, $\subseteq_u$ will be referred to as *strong untimed trace inclusion* when $\hookrightarrow = \rightarrow_u$, $\subseteq_t$ will be referred to as *weak timed trace inclusion* when $\hookrightarrow = \Rightarrow$, and $\subseteq_u$ will be referred to as *weak untimed trace inclusion* when $\hookrightarrow = \Rightarrow_u$.

With trace inclusion defined we can now use this to define trace equivalence in Definition 4.14 which is the same as having trace inclusion preorder in both directions.

Definition 4.14 - *Trace Equivalence*

Process s is said to be trace equivalent to process t if and only if

$$\mathit{Traces}_{\hookrightarrow}(s) = \mathit{Traces}_{\hookrightarrow}(t)$$

From now on the relation $\simeq_{T_t}$ will be referred to as *strong timed trace equivalence* when $\hookrightarrow = \rightarrow$, $\cong_{T_u}$ will be referred to as *strong untimed trace equivalence* when $\hookrightarrow = \rightarrow_u$, $\simeq_{T_t}$ will be referred to as *weak timed trace equivalence* when $\hookrightarrow = \Rightarrow$, and $\cong_{T_u}$ will be referred to as *weak untimed trace equivalence* when $\hookrightarrow = \Rightarrow_u$.

Example 4.15

Figure 4.1 shows two TLTSs where the timed traces of t_1 is included in s_1, i.e. $t_1 \subsetneq s_1$. Some examples of traces of s_1:

$$\{a, a1, 1a, 1a1, \cdots\}$$

Some examples of traces of t_1:

$$\{1, 11, 1a, 1a1, \cdots\}$$

The traces of s_1 is not included in t_1 because $s_1 \xrightarrow{a} s_2$ and $t_1 \not\xrightarrow{a}$. However, if we abstract away from time then $s_1 \simeq_{T_u} t_1$ because they both afford the same untimed trace, namely (a) and the empty trace.

4.4 Timed Generalized Parametric Semantic Relation

In this section we extend the generalized parametric semantic relation from Definition 3.2 to support both timed and untimed behavioral equivalences and preorders. We call this a *timed generalized parametric semantic relation*.

Definition 4.16 - *Timed Generalized Parametric Semantic Relation*

The timed generalized parametric semantic relation is a direct extension of Definition 3.2. The relation is extended to take one additional fixed parameter $\texttt{Time} \in \{\mathit{timed}, \mathit{untimed}\}$. The other parameters $\texttt{Type}$, $\texttt{Moves}$, E, d, k, and m are the same. Let $(\texttt{Proc}, \texttt{Lab}, \rightarrow)$ be a TLTS. We redefine $\overset{w}{\hookrightarrow}$ where $w \in \texttt{Lab}^*$ in the following way:

$$\overset{w}{\hookrightarrow} \overset{\text{def}}{=} \begin{cases} \overset{w}{\rightarrow} & \text{where } |w| = 1 \text{ if } \texttt{Type} = \textit{strong}, \quad \texttt{Moves} = \textit{short}, \quad \texttt{Time} = \textit{timed}, \\ \overset{w}{\rightarrow} & \text{where } |w| \geq 1 \text{ if } \texttt{Type} = \textit{strong}, \quad \texttt{Moves} = \textit{long}, \quad \texttt{Time} = \textit{timed}, \\ \overset{w}{\rightarrow}_u & \text{where } |w| = 1 \text{ if } \texttt{Type} = \textit{strong}, \quad \texttt{Moves} = \textit{short}, \quad \texttt{Time} = \textit{untimed}, \\ \overset{w}{\rightarrow}_u & \text{where } |w| \geq 1 \text{ if } \texttt{Type} = \textit{strong}, \quad \texttt{Moves} = \textit{long}, \quad \texttt{Time} = \textit{untimed}, \\ \overset{w}{\Rightarrow} & \text{where } |w| = 1 \text{ if } \texttt{Type} = \textit{weak}, \quad \texttt{Moves} = \textit{short}, \quad \texttt{Time} = \textit{timed}, \\ \overset{w}{\Rightarrow} & \text{where } |w| \geq 1 \text{ if } \texttt{Type} = \textit{weak}, \quad \texttt{Moves} = \textit{long}, \quad \texttt{Time} = \textit{timed}, \\ \overset{w}{\Rightarrow}_u & \text{where } |w| = 1 \text{ if } \texttt{Type} = \textit{weak}, \quad \texttt{Moves} = \textit{short}, \quad \texttt{Time} = \textit{untimed}, \\ \overset{w}{\Rightarrow}_u & \text{where } |w| \geq 1 \text{ if } \texttt{Type} = \textit{weak}, \quad \texttt{Moves} = \textit{long}, \quad \texttt{Time} = \textit{untimed}. \end{cases}$$

We now have that two states s and t are semantically related written $(s,t) \in \Delta(\texttt{Type}, \texttt{Moves}, \texttt{Time}, E, d, k, m)$ if and only if there is a timed generalized parametric semantic relation with respect to Type, Moves, Time, and E such that $(s,t) \in \mathcal{R}(d,k,m)$.

We reuse the rules from Section 3.3 to construct the dependency graph, and we have the following theorem:

Theorem 4.17

We have that $A_{min}(\langle (s,t), d, k, m \rangle) = 0$ on the dependency graph constructed using the rules shown in Figure 3.2 and Figure 3.3 if and only if $(s,t) \in \Delta(\texttt{Type}, \texttt{Moves}, \texttt{Time}, E, d, k, m)$.

Proof. The proof follows from Corollary 3.6. □

4.5 Timed HML

In real-time systems we are often interested in verifying that certain properties will always be satisfied before a specific amount of time has passed. Returning to our previous example, we might wish to verify that anytime the crash action is performed the inflate action becomes available within 1 time unit. Modelling such a property as a specification and then testing that specification against an implementation using some notion of equivalence often feels unnatural, especially if we are only interested in a small part of the system. In this section we present an extension of HML with timed modalities called Timed HML (THML), which will allow us to us express the aforementioned property and similar properties in a more natural way.

Definition 4.18 - *THML Syntax*

The set $\mathcal{M}_T$ of THML formulas is obtained by extending the set $\mathcal{M}_X$ of HML formulas given in Definition 1.17 with the following timed modalities:

$$\langle d \rangle F \;\Big|\; [d] F \;\Big|\; \langle\langle d \rangle\rangle F \;\Big|\; [[d]] F,$$

where $d \in \mathbb{N}_0$ is a time delay.

We can now intuitively express the property that anytime the crash action is performed the inflate action becomes available within 1 time unit as the recursive formula

$$X \overset{max}{=} [crash]\left(\overline{\langle inflate \rangle}\, tt \vee \langle 1 \rangle \overline{\langle inflate \rangle}\, tt\right) \wedge [\mathtt{Act}]\, X.$$

With this intuition let us now formally define the meaning of the timed modalities.

Definition 4.19 - *THML Semantics*

The semantics of THML is obtained by extending the semantics of recursive HML given in Definition 1.18 in the following way:

$$\begin{aligned}
\mathcal{O}_{\langle 1 \rangle F}(S) &= \langle \cdot 1 \cdot \rangle\, \mathcal{O}_F(S), \\
\mathcal{O}_{[1]F}(S) &= [\cdot 1 \cdot]\, \mathcal{O}_F(S), \\
\mathcal{O}_{\langle\langle 1 \rangle\rangle F}(S) &= \langle\langle \cdot 1 \cdot \rangle\rangle\, \mathcal{O}_F(S), \\
\mathcal{O}_{[[1]]F}(S) &= [[\cdot 1 \cdot]]\, \mathcal{O}_F(S),
\end{aligned}$$

where use the set operators $\langle \cdot 1 \cdot \rangle, [\cdot 1 \cdot], \langle\langle \cdot 1 \cdot \rangle\rangle, [[\cdot 1 \cdot]] : \mathcal{P}(\mathtt{Proc}) \to \mathcal{P}(\mathtt{Proc})$ which we define as:

$$\begin{aligned}
\langle \cdot 1 \cdot \rangle\, S &= \{p \in \mathtt{Proc} \mid p \xrightarrow{1} p' \text{ and } p' \in S \text{ for some } p'\}, \\
[\cdot 1 \cdot]\, S &= \{p \in \mathtt{Proc} \mid p \xrightarrow{1} p' \text{ implies } p' \in S \text{ for each } p'\}, \\
\langle\langle \cdot 1 \cdot \rangle\rangle\, S &= \{p \in \mathtt{Proc} \mid p \overset{1}{\Rightarrow} p' \text{ and } p' \in S \text{ for some } p'\}, \\
[[\cdot 1 \cdot]]\, S &= \{p \in \mathtt{Proc} \mid p \overset{1}{\Rightarrow} p' \text{ implies } p' \in S \text{ for each } p'\}.
\end{aligned}$$

It should be noted that syntactically delays of arbitrary length are allowed, whereas the semantics is only defined for 1-delays. This is due to the fact that any THML formula in $\mathcal{M}_T$ can be expanded into an equivalent formula also in $\mathcal{M}_T$, but where $d = 1$. For example, we have that

$$\langle 3 \rangle\, F \equiv \langle 1 \rangle \langle 1 \rangle \langle 1 \rangle\, F.$$

Example 4.20

Consider the process shown in Figure 4.2. We have the recursively defined property

$$X \overset{min}{=} [1]\, ff \vee \langle \mathtt{Lab} \rangle\, X$$

which is satisfied by a process that cannot delay. We start by applying the function $\mathcal{O}_{F_X}$ to the empty set since we are looking for the minimum fixed-point:

$$\begin{aligned}
\mathcal{O}_{F_X}(\emptyset) &= [\cdot 1 \cdot]\, \emptyset \cup ([\cdot 1 \cdot]\, \emptyset \cap [\cdot a \cdot]\, \emptyset \cap [\cdot \tau \cdot]\, \emptyset) \\
&= \{s_2\} \cup (\{s_2\} \cap \{s_2\} \cap \{s_1\}) \\
&= \{s_2\}
\end{aligned}$$

We now apply $\mathcal{O}_{F_X}$ to the set $\{s_2\}$:

$$\begin{aligned}\mathcal{O}_{F_X}(\{s_2\}) &= [\cdot 1\cdot]\{s_2\} \cup ([\cdot 1\cdot]\{s_2\} \cap [\cdot a\cdot]\{s_2\} \cap [\cdot \tau\cdot]\{s_2\}) \\ &= \{s_2\} \cup (\{s_2\} \cap \{s_1, s_2\} \cap \{s_1, s_2\}) \\ &= \{s_2\}\end{aligned}$$

We now have that the set $\{s_2\}$ is the minimum fixed-point of $\mathcal{O}_{F_X}$. Intuitively we can see that this is correct since s_2 can perform a τ-action, and thus cannot delay, whereas s_1 can delay indefinitely. The maximum fixed-point of $\mathcal{O}_{F_X}$ is the set $\{s_1, s_2\}$, which clearly is not the solution that we are looking for in this case.

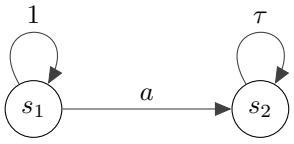


Figure 4.2: A process.

Implementation 5

We started the development of Caal as part of our pre-specialization where we created the visualizer for CCS processes, a verifier for bisimulation and model checking with HML. We now extend our implementation with more equivalences and preorders as well as games for them, and we add the language TCCS to the tool including timed and untimed equivalences and preorders.

Caal is built as a web application in order to avoid the hassle of installation and also to make it easy to access, effectively reducing the time spent on getting started during valuable lecture time, since it is designed for educational purposes. The tool is available at

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

Caal is built with TypeScript, a typed superset of JavaScript that compiles into plain JavaScript [12]. We welcome any contributions to Caal and the tool is licensed under the MIT License.

5.1 Successor Generation

We already have strong and weak successor generators for CCS, and the successor generator for TCCS is extended directly from the successor generator for CCS. We reuse the CCS successor generator to generate all action transitions and then add delay transitions to them. By the SOS rules in Definition 4.3 we see that delay transitions do not resolve nondeterministic choices unlike action transitions, and delay transitions delay an entire parallel system. This means that any TCCS process can perform at most one delay transition. If the current TCCS state can perform a τ-transition we do not add any delay transition to the result, if the current TCCS state cannot perform a τ-transition we compute the 1 time unit delay successor.

For all the delay prefixes in a current process state s, the delay successor s' from the transition $s \xrightarrow{1} s'$ will subtract 1 from all the currently guarding delay prefixes in s. If the state does not contain any delay prefixes, then the delay successor is a self-loop $s \xrightarrow{1} s$.

We can use the strong timed successor generator to form the basis of the weak version and also the untimed successor generator. We had previously implemented a weak successor generator for CCS. The weak successor generator uses a strong successor generator to generate the reflexive and transitive closure of τ transitions. We change this successor generator to a more generic version which we will refer to as an abstracting successor generator. The abstracting successor generator has two parameters, another successor generator to create abstracting successors with, and one or more transitions that should be abstracted away from.

Weak timed generator To create the weak timed successor generator, we give the abstracting successor generator the timed successor generator and tell it to abstract away from $\xrightarrow{\tau}$ transitions. The result will thus

contain weak action successors and also weak delay successors that all abstract away from $\xrightarrow{\tau}$ transitions.

Strong untimed generator To create the strong untimed successor generator, we give the abstracting successor generator the timed successor generator and tell it to abstract away from $\xrightarrow{1}$ transitions. The result will thus contain strong untimed action successors that all abstract away from $\xrightarrow{1}$ transitions.

Weak untimed generator To create the weak untimed successor generator, we give the abstracting successor generator the timed successor generator and tell it to abstract away from $\xrightarrow{\tau}$ and $\xrightarrow{1}$ transitions. The result will thus contain weak action successors that all abstract away from $\xrightarrow{\tau}$ and $\xrightarrow{1}$ transitions.

5.2 Visualization

The visualizer allows exploration of the LTS or the TLTS of a given process with different successor generators. The visualizer can already display the strong and weak transition relations. We add the four new transition relations discussed in Section 5.1 to the visualizer. Figure 5.1 shows the visualizer using the strong timed successor generator to display the TLTS for the Impl process of the airbag example from Section 4.1.

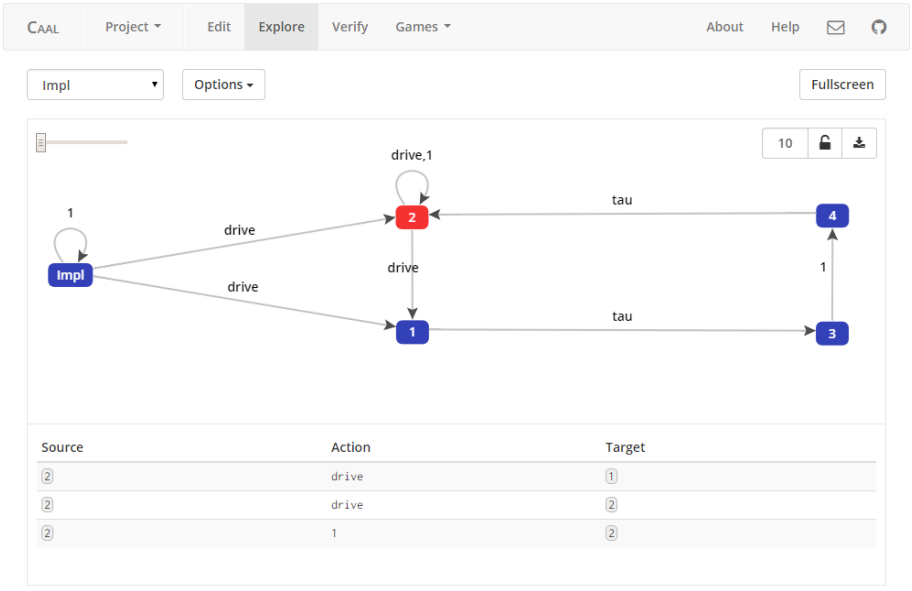


Figure 5.1: Visualization of a TLTS.

The states in the TLTS can be selected by clicking them. The selected state is highlighted in red and the transitions available from that state will be displayed in the table below the TLTS. By clicking the “Options”-button

it is possible to select which transition relation should be used, as shown in Figure 5.2.

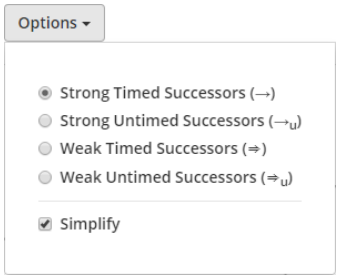


Figure 5.2: Transitions relations for visualization.

We have some different options for the displayed TLTS:

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

Expand Depth The number at the top right is the number of states to expand the TLTS 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 TLTS. The states in the TLTS are automatically positioned, but may sometimes become cluttered if there are too many states or transitions. Locking the TLTS makes it possible to manually rearrange the states in the TLTS.

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

5.3 Verification

The verifier in Caal already has the ability to verify strong and weak bisimulation between CCS processes as well as model checking with HML [14]. We extend the verifier with additional equivalences and preorders for CCS and timed/untimed versions for TCCS as well as THML. The verifier can be seen inFigure 5.3.

5.3.1 Equivalences and Preorders

We add the following equivalences and preorders to the verifier:

- simulation,
- simulation equivalence,
- trace inclusion, and

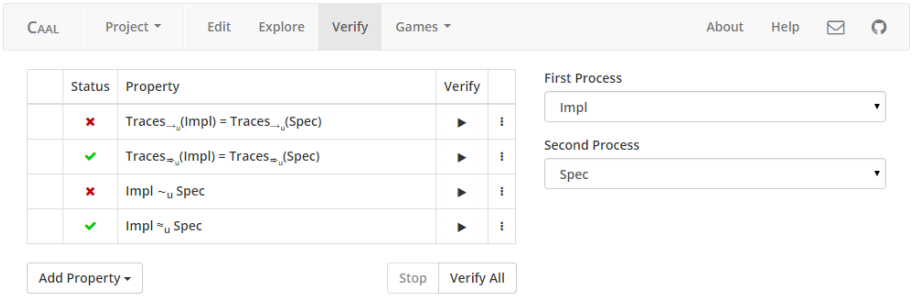


Figure 5.3: Verification of untimed trace equivalence and bisimulation.

- trace equivalence.

The implementation of simulation consists of reusing parts of the implementation for bisimulation. Instead of allowing moves on either side, that can be matched with a move on the opposite side, we only allow moves on the left side that can be matched by a move on the right side. For simulation equivalence we apply the simulation preorder in both directions as defined in Definition 1.4.

The dependency graph for trace inclusion is made using a simplified version of the generalized parametric semantic relation. Trace inclusion only uses a few of the construction rules from Figure 3.3 since its game characterization consists of a single round where the attacker plays a long move on the left. For trace equivalence we apply trace inclusion preorder in both directions as defined in Definition 1.10.

All the verifications of CCS equivalences and preorders can be reused for TCCS. The definitions for equivalences and preorders on TLTSs are very similar to the definitions on LTSs. In fact we only add one more case to each of them, namely the 1-delay transition. Because of this we say that delay is a subclass of action with a unique action name not in `Act`. To make all the different equivalences and preorders we only need to provide the verifier with the correct successor generator effectively adding strong/weak and timed/untimed equivalences and preorders for TCCS.

5.3.2 Timed HML

Verification of THML formulas is performed using the dependency graph construction rules for HML formulas described in [14]. The implementation has been reused, but the dependency graph is constructed using the successor generators for TCCS.

All formulas containing timed modalities are expanded into equivalent formulas using only 1-delays during parsing. For example, the formula $\langle 3 \rangle F$ is expanded into the formula $\langle 1 \rangle \langle 1 \rangle \langle 1 \rangle F$. Timed modalities may also contain intervals using the syntax $\langle d_{min}, d_{max} \rangle$ (analogous for the other timed modalities) where $d_{min}, d_{max} \in \mathbb{N}_0$ and $d_{min} \leq d_{max}$. A formula such as $\langle 0, 2 \rangle F$ is expanded into the formula $F \vee \langle 1 \rangle F \vee \langle 1 \rangle \langle 1 \rangle F$. The modalities $[d] F$ and $[[d]] F$ are expanded using a conjunction instead of a disjunction.

5.4 Game Implementation

We have implemented the following games:

- strong/weak simulation,
- strong/weak bisimulation,
- strong/weak trace inclusion.

In Section 6.5.2 we go into detail with the design and all the features of the strong bisimulation game by giving an example of a game. The weak bisimulation game is very similar to the strong version in terms of design. The simulation game is also very similar to the bisimulation game, but for simulation we only allow the attacker to make moves on the left LTS and the defender is only allowed to make moves on the right LTS.

The rules for the trace inclusion game which we showed in the form of a generalized parametric semantic relation in Section 3.2 specify that the attacker plays a trace i.e. a long move, and the defender has to respond with the same trace in the other LTS. The game only runs for one round. Instead of creating the game with two LTSs, we generate a trace in the form of an HML formula which only the left LTS satisfies, and then play the HML game for the right LTS on the formula. If only one process satisfies the trace ab, then the HML formula trace looks like $\langle a \rangle \langle b \rangle tt$.

5.4.1 Computer Strategy

We give the player the option to assume the role of either the attacker or the defender and the computer will assume the opposite role. At all times during the game, we know which role has a universal winning strategy. We use this information for the computer to make clever moves.

Winning attack The computer has a universal winning attack strategy, meaning the current node has the assignment 1. The computer can then pick any edge leading to a node with the assignment 1 and would still have a universal winning strategy. However it can be possible for the dependency graph to have an edge going back to a previously visited node, i.e. a loop.

The computer finds and picks the edge which leads to a node with the lowest possible level, in order to win quickly.

Losing attack The computer does not currently have a universal winning strategy, which means levels on nodes are not available. It will instead try to confuse its opponent.

The computer looks ahead on the options for the opponent by following the potential edges the computer can pick. The computer will then find the play which yields the highest ratio of 1-assignments for the defender to pick between. If there are multiple plays with the same ratio, the computer will pick a random one of these; this will make the computer randomly try something new.

Winning defend The nodes will not have a level in this case. The computer will then pick any of the nodes which has a 0-assignment which will yield a winning play at some point.

Losing defend The computer does not have a universal winning strategy, but will attempt to keep the game going for as long as possible.

The computer finds and picks the node with the highest possible level, in order to keep the game going for as long as possible. If there are multiple nodes with the same level, the computer will pick a random one of these.

CAAL *Tutorial*

6

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 CCS (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`.

6.1 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:

$$\texttt{Boy} = \texttt{shake}.\texttt{walk}.0 \ . \tag{6.1}$$

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:

$$\texttt{ColorTree} = \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{ColorTree} + \overline{\texttt{'redapple}}.\texttt{ColorTree}) \ . \tag{6.2}$$

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 6.1, 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:

$$\texttt{Girl} = \overline{\texttt{'shake}}.\texttt{apple}.\overline{\texttt{'walk}}.\texttt{Girl} \ . \tag{6.3}$$

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 6.3 and expand the example to accept red or green apples:

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{Man1 + Man2}) \ , \\ \texttt{Man1} &= \texttt{redapple}.\overline{\texttt{'walk}}.\texttt{Man} \ , \\ \texttt{Man2} &= \texttt{greenapple}.\overline{\texttt{'throw}}.\texttt{Man} \ . \end{aligned} \tag{6.4}$$

Now we can define the Orchard using the ColorTree from Equation 6.2 and the refined Man from Equation 6.4:

$$\texttt{Orchard} = (\texttt{ColorTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ . \tag{6.5}$$

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.

6.2 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 6.2 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 6.4 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.

6.3 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 6.2 and Equation 6.4 as shown in Figure 6.1.

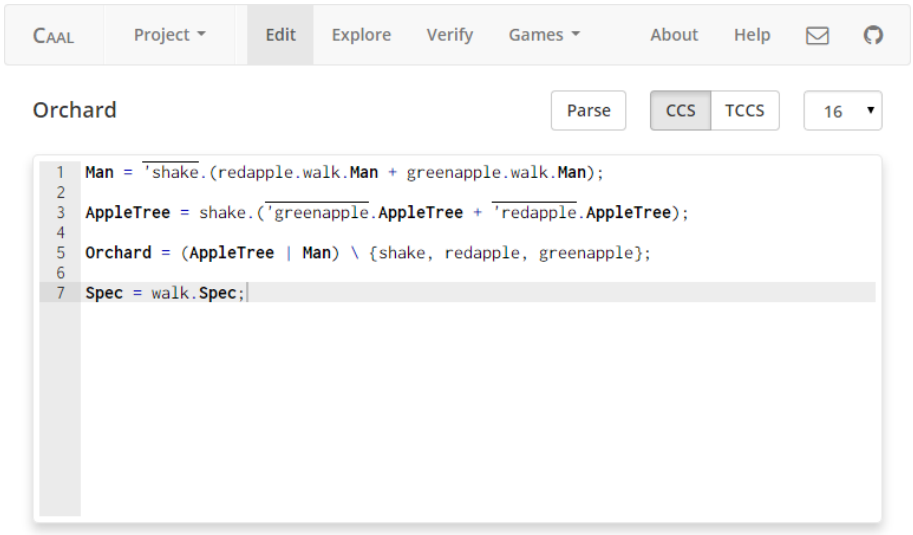


Figure 6.1: Editor.

6.4 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.

6.4.1 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 6.1

We have the CCS processes:

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

We want to check if the processes Orchard and Spec are strongly or weakly bisimilar. Figure 6.2 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.

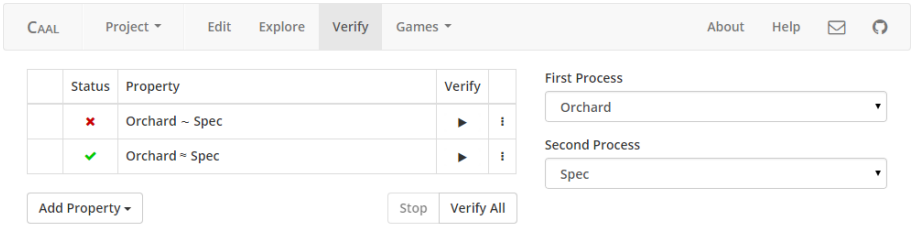


Figure 6.2: Verification of bisimulation.

6.4.2 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
```
(6.6)

Equation 6.6 is an example of a supported HML formula.

```
X min= <a>X or Y
Y max= [b]Y and X
```
(6.7)

Equation 6.7 is an example of an HML formula that is not allowed because Y refers back to X.

Example 6.2

We have the CCS processes

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

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 6.3 shows the result of the verification. As we can see, this property is not satisfied, as indicated by the red cross.

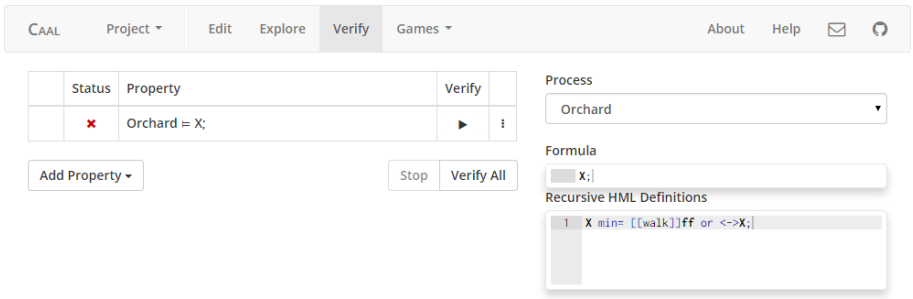


Figure 6.3: Verification of a recursive HML formula.

6.4.3 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 6.3

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 6.4.

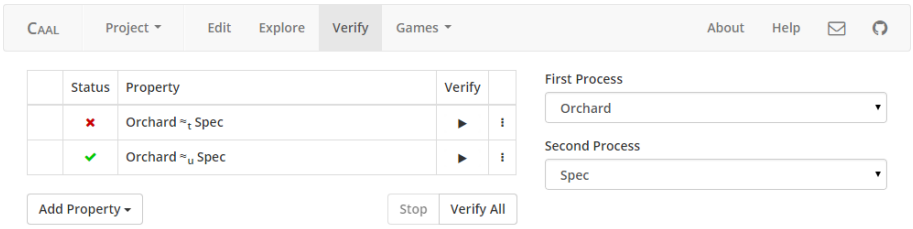


Figure 6.4: Verification of bisimulation with time.

As seen in Example 6.2, 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 6.4

We want to verify that the Man from Example 6.3 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 6.5 the property is satisfied.

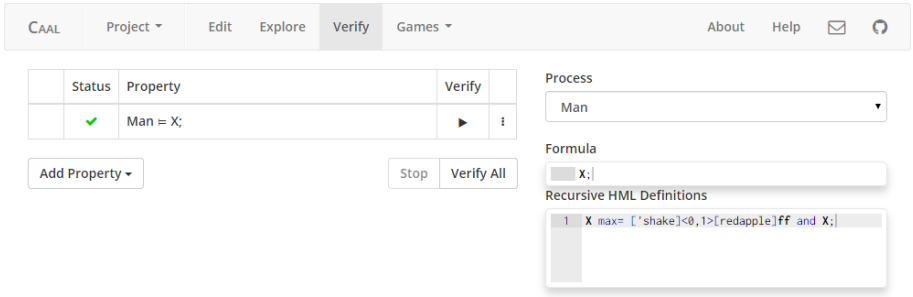


Figure 6.5: Verification of a recursive THML formula.

6.5 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.

6.5.1 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 6.6.

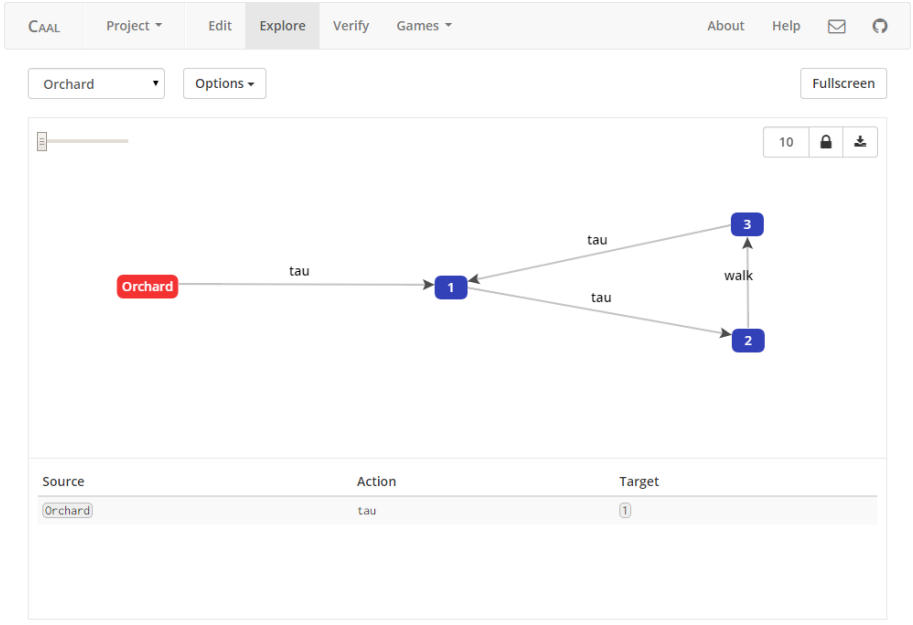


Figure 6.6: Explorer.

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 6.7 shows the Orchard with strong bisimulation collapse, and Figure 6.8 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.

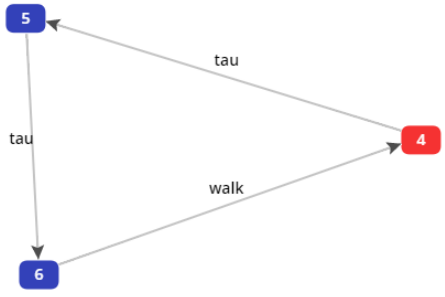


Figure 6.7: Orchard with strong bisimulation collapse.

Figure 6.8: Orchard with weak bisimulation collapse.

6.5.2 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 6.2:

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

Figure 6.9 shows a strong bisimulation game where the player is playing as attacker against the computer in the defender role.

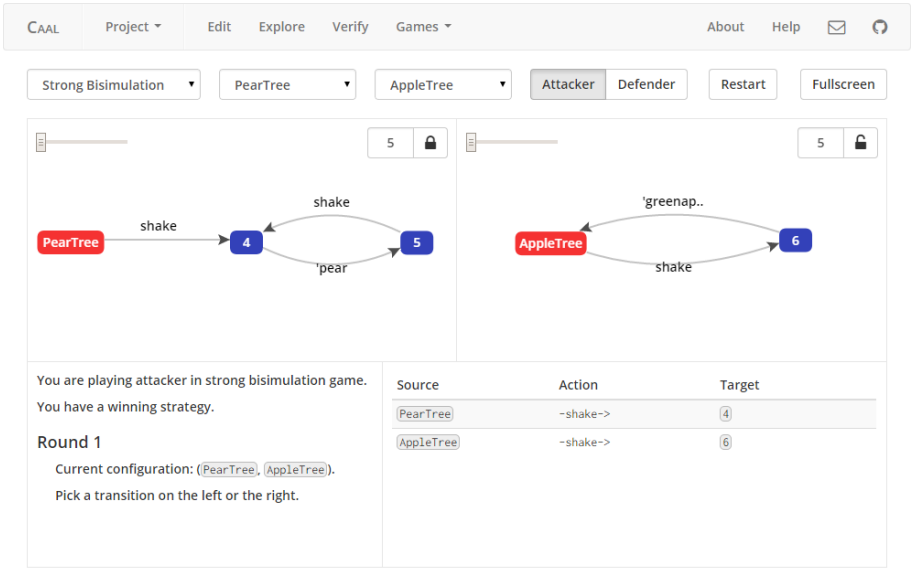


Figure 6.9: Screenshot of the strong bisimulation game.

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 6.10, 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.

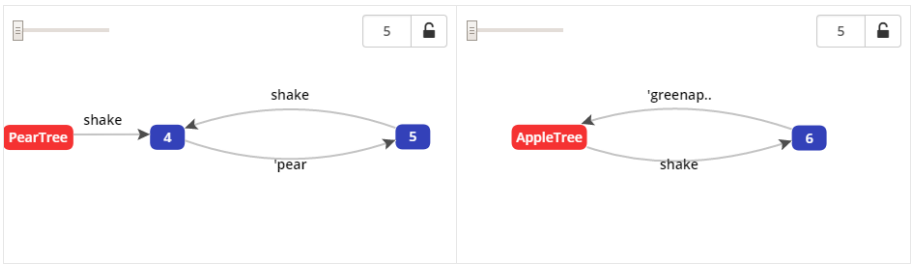


Figure 6.10: Game LTSs.

Figure 6.11 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.

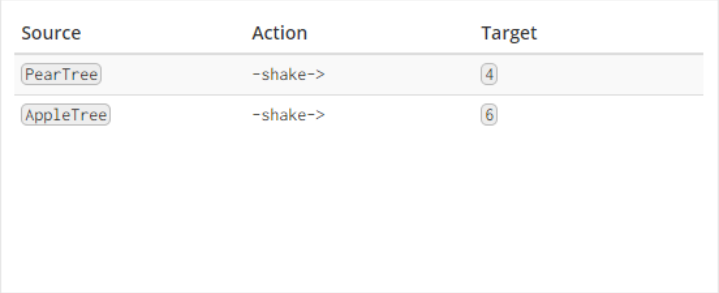


Figure 6.11: Available transitions.

Figure 6.12 shows the different steps of a full game in the game log. The initial state of the game log is shown in Figure 6.12a, 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 6.12b 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 6.12c 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 6.12d 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 6.12: The game log.

6.5.3 HML Game

An HML game consists of an "attacker", a "defender", a process s, and a formula F. 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 $(s, F_1 \wedge F_2)$, or when the choices are either $(s, [\alpha]\, F)$ or $(s, [[\alpha]]F)$.
- The defender picks a configuration when the formula is of the form $(s, F_1 \vee F_2)$, or when the choices are $(s, \langle\alpha\rangle\, F)$ or $(s, \langle\langle\alpha\rangle\rangle F)$.

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: $X \overset{\min}{=} F$. The defender is the winner in every infinite play provided that X is defined as a maximum fixed-point: $X \overset{\max}{=} F$.

Figure 6.13 shows an HML game where the user is playing as defender against the computer.

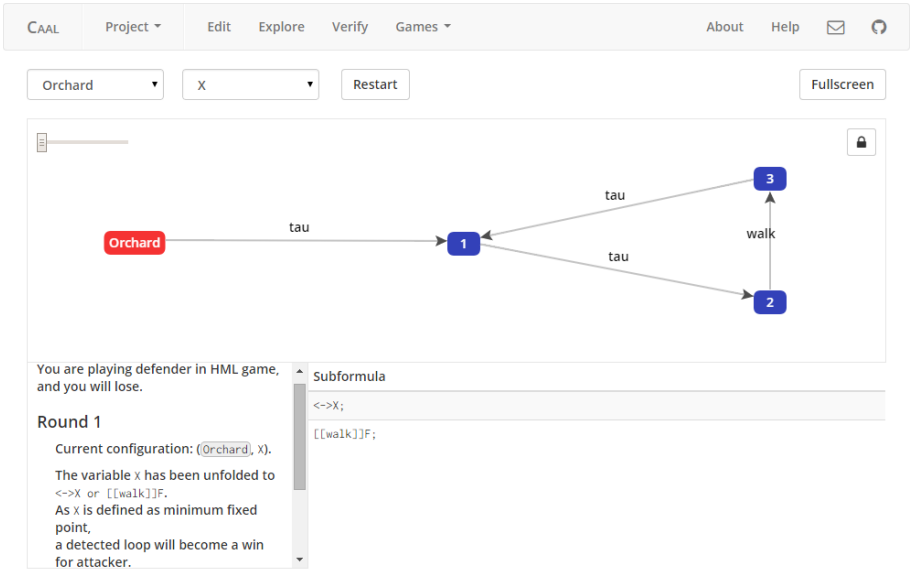


Figure 6.13: HML Game.

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 6.2 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

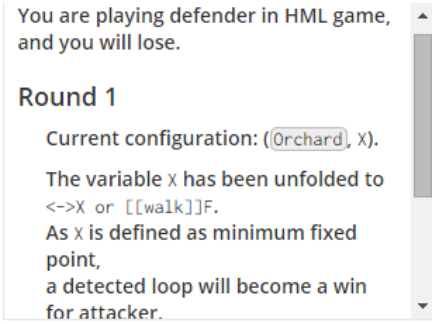
$$\texttt{X min= [[walk]]ff or <->X} \ .$$

The initial game log can be seen in Figure 6.14a. As it can be seen, we are playing as defender and we are going to lose, matching the claim from Figure 6.3 that the formula is not satisfied.

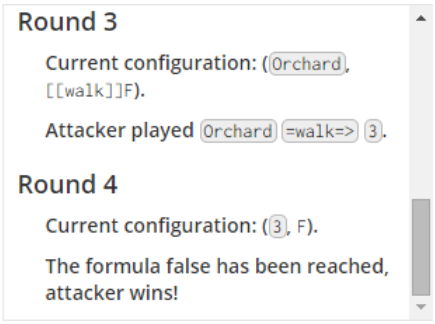
As defender we have to choose which of the disjunctions we want to continue from. We can pick between two subformulas:

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

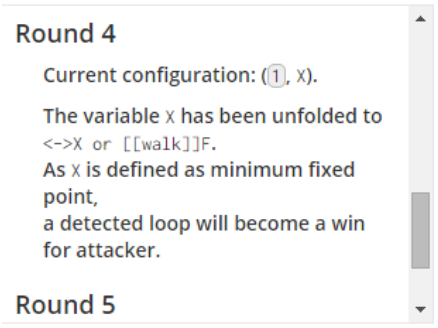
Taking case 1 will result in a loss in the next round because the attacker picks a transition so we reach a false formula as shown in Figure 6.14b. Instead the defender picks case 2 and picks a transition, resulting in `X` being unfolded as it can be seen on Figure 6.14c. The game continues with the defender picking one of the two cases, each time unfolding `X` and having to pick a transition as seen on Figure 6.14d. Figure 6.15 shows the transition table for the defender. Eventually the game will detect a cycle as it can be seen in Figure 6.14e, which means the defender loses because we played in a minimum fix-point game.



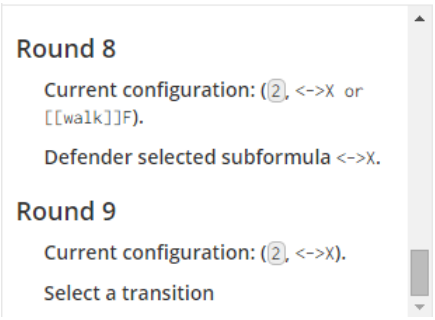
(a) Game log with introduction.



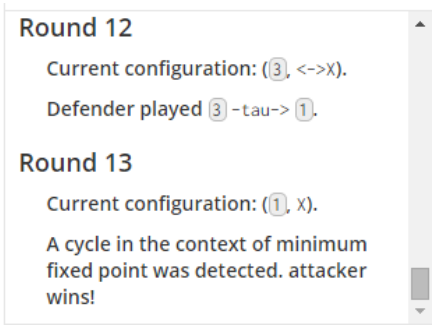
(b) Reaching false formula.



(c) Unfold X.



(d) Select a transition.



(e) Game log with a winner.

Figure 6.14: Game log for HML Game.

Source	Action	Target
2	walk	3

Figure 6.15: Transition table.

6.5.4 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 6.5

We have the processes

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{FastMan} &= \overline{\texttt{'shake}}.(\texttt{redapple.}(\texttt{walk.FastMan + FastMan}) \ + \\ &\qquad \texttt{greenapple.}(\texttt{walk.FastMan + FastMan}) \ . \end{aligned}$$

CAAL is able to generate a distinguishing formula for two processes. Figure 6.16 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'.

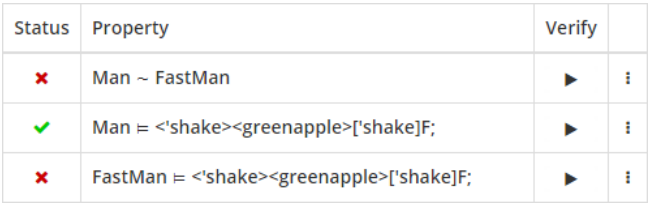


Figure 6.16: Distinguishing formula.

6.5.5 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 6.17 shows a distinguishing trace for the processes Man and FastMan from Example 6.5.

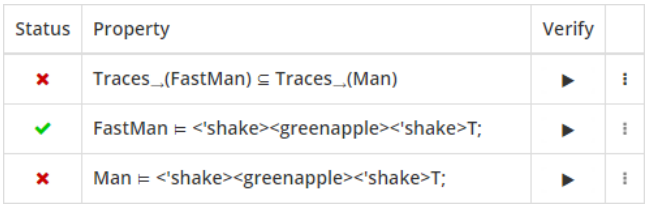


Figure 6.17: Distinguishing trace.

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.

6.6 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 and Future Work 7

In this project we have extended Caal with additional equivalences and preorders, namely strong and weak variants of simulation, simulation equivalence, trace inclusion, and trace equivalence. For all of these equivalences and preorders we have implemented interactive games where the user plays a game against the computer in order to prove or disprove the result of an equivalence checking problem.

Real-time systems often have strict timing constraints which cannot be modelled using CCS alone. We extend Caal with support for the timed process algebra TCCS with discrete time delays. We suggest a special semantics using delays of 1 time unit only in order to simplify the implementation. However, we still allow delays of arbitrary length in the syntax. We define timed and untimed variants of the existing equivalences and preorders in Caal and implement these. We also extend Caal with games for all of the aforementioned timed and untimed equivalences and preorders. Furthermore, we implement Timed HML in Caal, which is a timed variant of recursive HML.

We take advantage of the fact that the game characterizations of different equivalences and preorders are similar in nature by defining a parameterized relation, the *generalized parametric semantic relation*, where the parameters correspond to the rules of the game characterization of the equivalence or preorder that we wish to express. We also show how the problem of determining if a pair of states are related by a generalized parametric semantic relation can be reduced to the problem of computing the minimum pre-fixed-point assignment on a dependency graph. We compute the minimum pre-fixed-point assignment using an effective on-the-fly algorithm. Finally, we extend the generalized parametric semantic relation with an additional parameter, allowing it to also express timed and untimed equivalences and preorders.

Caal was used in the first half of the course Semantics and Verification at Aalborg University as a replacement for Edinburgh Concurrency Workbench (CWB), and was generally very well received. At the end of the first part the students were asked to rate their experience working with Caal on a scale from 1 to 9, where 1 is very bad and 9 is very good. The games received an average score of 7.0, and overall Caal received an average score of 7.3.

In the future we should like to extend the time domain of our TCCS implementation from the set of non-negative natural numbers to the set of non-negative real numbers. Viewing the flow of time as being continuous is more natural, but there are several challenges that must be dealt with. Even the simplest of processes generate uncountably many reachable states, and thus our current verification engine using dependency graphs cannot be used.

A full implementation of the generalized parametric semantic relation would also be a valuable extension of Caal, as it would allow us to implement additional equivalences and preorders along with their respective games in a more dynamic fashion.

Finally, there are certain areas of Caal where the user experience needs to be improved. In particular, the user interface of the verification module needs

to be redesigned in order to increase its user-friendliness and to make it more aesthetically pleasing. There are also certain features, such as importing and exporting projects using files, which do not work across all browsers.

Bibliography

[1] 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.

[2] Rance Cleaveland and Oleg Sokolsky. Equivalence and Preorder Checking for Finite-State Systems. *Handbook of Process Algebra*, pages 391–424, 2001.

[3] Concurrency Workbench of the New Century. `http://www3.cs.stonybrook.edu/~cwb`.

[4] Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[5] Kim G. Larsen. Proof Systems for Hennessy-Milner Logic with Recursion. In *CAAP '88*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin Heidelberg, 1988. ISBN 978-3-540-19021-9.

[6] 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.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0131149849.

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

[9] David Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. ISBN 3-540-10576-X.

[10] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math*, 5(2):285–309, 1955.

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

[12] TypeScript. `http://typescriptlang.org`.

[13] R.J. van Glabbeek. The Linear Time - Branching Time Spectrum. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer Berlin Heidelberg, 1990. ISBN 978-3-540-53048-0.

[14] Jacob Karstensen Wortmann, Jesper Riemer Andersen, Mathias Munk Hansen, Nicklas Andersen, Simon Reedtz Olesen, and Søren Enevoldsen. CAAL, 2015.

[15] Jacob Karstensen Wortmann, Simon Reedtz Olesen, and Søren Enevoldsen. CAAL 2.0 - Recursive HML, Distinguishing Formulae, Equivalence Collapses and Parallel Fixed-Point Computations, 2015.

[16] Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, ICALP '91, pages 217–228. Springer-Verlag, 1991. ISBN 3-540-54233-7.