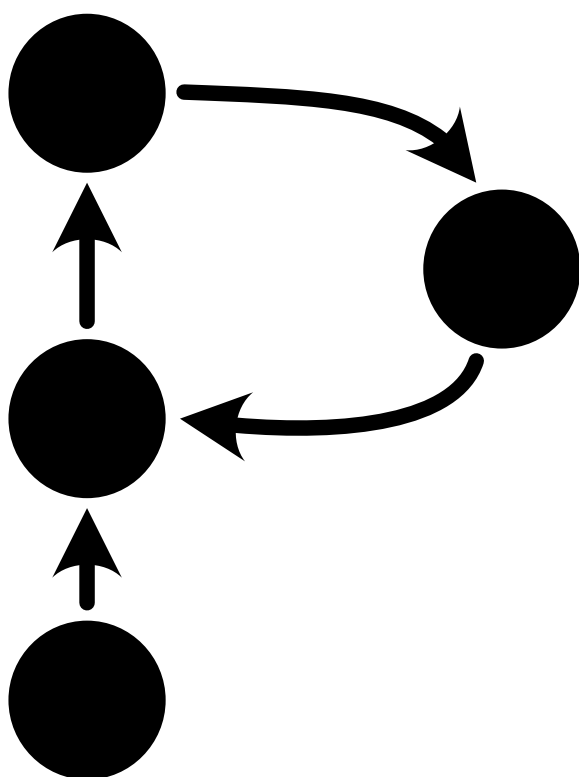




pseuCo Book

Contents

1	CCS	5
1.1	Introduction: Calculating with Models	5
1.1.1	Getting our Feet Wet: First Steps in CCS	5
1.2	Sequential CCS without Recursion	6
1.2.1	Syntax	6
1.2.2	Semantics: Breathing (Formal) Life into CCS	7
1.3	Sequential CCS (with Recursion)	11
1.3.1	Syntax	12
1.3.2	Semantics	12
1.3.3	The Power of Sequential CCS	13
1.3.4	Guardedness & Termination of CCS Semantics	14
1.4	CCS: The Full Monty	15
1.4.1	Parallel Composition and Communication	16
1.4.2	Syntax	18
1.4.3	Semantics	19
1.4.4	Operator Lifetime and Regularity	19
1.5	CCS with Values	22
1.5.1	Adding Values	23
1.5.2	Process Parameters	26
1.5.3	Conditional Transitions	27
1.5.4	Syntax	28
1.5.5	Semantics	29
1.5.6	Ranges for CCS in pseuCo.com	30
1.6	Workout	30
2	Equality	35
2.1	Introduction	35
2.2	Trivial Notions of Equality	35
2.2.1	Identity	35
2.2.2	Universal Equality	36
2.3	Isomorphism	37
2.3.1	Equivalence Relation	37
2.3.2	Isomorphism in Practice	37

2.3.3	An Unexpected Problem	38
2.3.4	Hierarchy	39
2.3.5	Summary	39
2.4	Trace Equivalence	40
2.4.1	Termination Sensitivity	41
2.4.2	Hierarchy	41
2.5	Bisimilarity	43
2.5.1	First Attempt	43
2.5.2	Second Attempt	44
2.5.3	Proving Bisimilarity	44
2.5.4	Disproving Bisimilarity	45
2.5.5	Properties of Bisimilarity	47
2.5.6	Termination Sensitivity	48
2.5.7	Hierarchy	49
2.5.8	Minimization	49
2.5.9	The perfect equality?	50
2.6	Weak Bisimilarity	50
2.6.1	A Problem Returns	53
2.6.2	Hierarchy	54
2.7	Observation Congruence	54
2.7.1	Proving Observation Congruence	55
2.7.2	Disproving Observation Congruence	55
2.7.3	Hierarchy	56
2.7.4	Properties of Observation Congruence	56
2.7.5	Minimization	58
2.8	Workout	59
2.8.1	Bisimilarity	60
2.8.2	Weak Bisimilarity	61
2.8.3	Observation Congruence	61
3	Programming with pseuCo	63
3.1	A Gentle Introduction to pseuCo	63
3.1.1	About this Chapter	63
3.1.2	Sequential pseuCo	63
3.1.3	Starting Agents	64
3.1.4	Intermezzo: The Dangers of Uncoordinated Access to Shared Memory	65
3.2	Message Passing	68
3.2.1	Introducing Channels	68
3.2.2	Select-Case	70
3.2.3	Working with Message Passing	71
3.3	Shared Memory	75
3.3.1	Introducing Locks	75
3.3.2	Locking Locked Locks	76
3.3.3	Arrays, Structures, and Sneaky Data Races	77
3.3.4	Protecting Structured Data with Monitors	79

Version master [340ff82a7738edb3b362c93e4a260d62ccd86583, #73663]

Chapter 1

CCS

1.1 Introduction: Calculating with Models

At this point, you should be familiar with the idea of modelling systems using labelled transition systems. While these models can be insightful, they can also be a bit unwieldy: Drawing or reading an LTS with 10, 20, or maybe even a hundred states certainly isn't fun, and surely, no-one is going to claim that something like $(\{a, b, c\}, \{(a, \textit{strike}, b), (b, \textit{burn}, b), (b, \textit{extinguish}, c)\}, a)$ is a succinct, easy-to-parse model of a match.

This becomes even more apparent when *creating* models of systems: Except for parallel execution, there isn't a way to systematically compose parts of systems, so if you want to model a large, complicated system, you basically have to do it in one go.

In this chapter, we will tackle this problem by introducing a new way of creating models using a *calculus*. We will develop a new syntax to write down concurrent systems, one that is given life by a translation into the labelled transition systems you know already.

Our ultimate goal will be to not only ease the process of modelling systems (and understanding such models), but also to deepen our understanding of how concurrent systems operate and communicate. And, as an aside, we will gain a new way of comparing processes by looking at them on a syntactic level.

1.1.1 Getting our Feet Wet: First Steps in CCS

The calculus we are using will be based on Robin Milner's *Calculus of Communicating Systems*, called *CCS* for short. Before delving into a proper definition of CCS, let's explore a few motivating examples.

First, let's look at the simplest structure a model can have: Models that simply perform a sequence of actions one after another. An example of such a model is shown in Figure 1.1.

Figure 1.1: A simple labelled transition system modelling a match that performs a sequence of actions

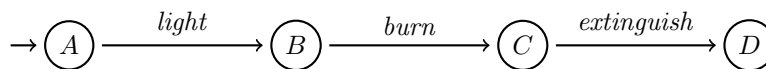
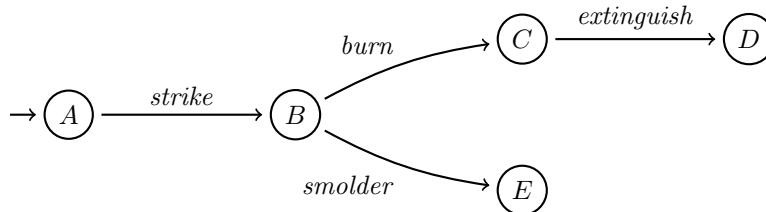


Figure 1.2: A model of a wet match that might burn properly, but also might not



In essence, the system shown there can be described as “first *strike*, then *burn*, then *extinguish*, then nothing more”. In CCS, we will write this down as *strike.burn.extinguish.0*. There, the dot ($.$), called *prefix*, separates an action from what happens afterwards, and the 0 , called “stop”, describes the process that can no longer do anything.

You may have noticed that we are not specifying the names of the states A , B , C , and D . That is intentional: CCS will (for now) not give us control over the names of the states. This is no problem for us: The interesting part of a system is the (observable) actions it can perform and *when* it can perform them; the state names are more of a technicality, assigned merely because we need some way to refer to a state. We will formalize these thoughts later on; for the moment, we’ll simply blissfully ignore the names of states.

Systems that merely perform a single sequence of actions are, of course, not the most interesting processes to study. To get to something more interesting, we must add at least one more capability to CCS: The ability to model processes that can branch.

We will do this by adding a single operator to CCS, the *choice* operator $+$. It allows a process to choose between multiple options. For example, *strike.(burn.extinguish.0 + smolder.0)* might describe a wet match that, after lighting it, might burn and extinguish, but it might also merely smolder and never properly burn. The resulting system is sketched in Figure 1.2.¹

Note that the choice operator can only occur on the right side of a prefix operator, never on the left.

CCS has a lot more to offer than these operators, but before we venture any deeper, it is time that we formalize what we have seen so far.

1.2 Sequential CCS without Recursion

In the previous section, you have seen your first CCS terms, and gained an intuitive understanding of their semantics. In this section, we will nail down the syntax and semantics of CCS formally. We will begin with the subset of CCS you know already, called CCS_0 .

1.2.1 Syntax

Let Act be a set of actions. When working with CCS, we impose slightly more structure on this set, by assuming there is only a single action representing internal steps, called τ (“tau”), so $Act = Com \cup \{\tau\}$. This is not a major restriction: Since internal actions are representing something that should not be observable or even distinguishable for an external observer, conflating all internal actions makes no difference to this external observer. For the communication actions Com , we typically use letters or words.

¹One detail is not correct in this sketch. Can you find it? We will look at this example more closely once we have properly defined CCS.

When using variables that range over actions, by convention, we use $\alpha, \beta, \dots$ to range over all actions (Act), and $a, b, \dots$ to range over communication actions (Com).

Definition 1 (Sequential CCS without Recursion (CCS_0)). Let Act be a set of actions. The set of expressions in CCS_0 is defined as

$$P ::= 0 \quad | \quad P + P \quad | \quad \alpha.P$$

provided that $\alpha \in Act$.

As seen in the definition, we typically use capital letters like P or Q to refer to CCS terms.

To reduce the number of parentheses, we use the following precedence rules:

- $P + Q + R \rightsquigarrow (P + Q) + R$, i.e. the choice operator is left-associative
- $\alpha.\beta.P \rightsquigarrow \alpha.(\beta.P)$, i.e. prefix is right-associative²
- $\alpha.P + Q \rightsquigarrow (\alpha.P) + Q$, i.e. prefix has higher precedence than choice

Let's try out this syntax for a bit.

🔗 Toying with the Syntax of CCS

book.pseuCo.com

Which of the following are valid CCS_0 terms?

- (a) 0
- (b) $a.b.0$
- (c) $(a.b).0$
- (d) $a.(b.0)$
- (e) $a.b$
- (f) $0.a.b$
- (g) $a.(b.0) + 0$
- (h) $0 + 0$

Solve this quiz and check your solutions online:
<https://book.pseuco.com/#/quiz/ccs-0-syntax-basics>

1.2.2 Semantics: Breathing (Formal) Life into CCS

The next step is to assign meaning to CCS_0 , by capturing the intuition you learned in the previous section, and filling in the blanks as far as formal details go.

The most natural way to turn CCS_0 processes into labelled transition systems involves a rather neat trick: We use CCS_0 terms themselves as the states in the resulting LTS. So, for example, let's reconsider the simple match we investigated before, described by the CCS_0 term $light.burn.extinguish.0$.

Instead of somehow coming up with fresh state names, we put the CCS term $light.burn.extinguish.0$ itself into the LTS as its initial state. It can perform a transition with the action $light$, resulting in a state that should still do $burn$ and $extinguish$. We now simply call this new state $burn.extinguish.0$, i.e. again identify it with the CCS term describing its remaining behavior.

This results in the LTS shown in Figure 1.3.

²Note that prefix being left-associative would not make sense: The resulting terms would not be syntactically valid.

Figure 1.3: A simple match, now properly modelled using CCS_0

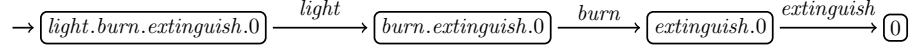


Figure 1.4: An example of a minimal inference rule

$$\frac{A \quad B}{C} \text{ example}$$

Provided that we are somehow able to describe a relation $\rightarrow \subseteq CCS_0 \times Act \times CCS_0$ that describes which transitions any CCS_0 terms has (and which CCS_0 terms result from executing it), we can then simply set the semantics $\llbracket P \rrbracket$ of a CCS_0 term P to be:

$$\llbracket P \rrbracket := (CCS_0, \rightarrow, P)$$

You may have noticed that this definition is *not* restricting the sets of states and transitions to what is reachable from the initial state. This slightly simplifies the definition and does not matter in practice as unreachable state do not influence the behavior of a model.

The open question that remains, of course, is defining the transition relation $\rightarrow$. Our goal is to find a flexible definition that is easy to apply and can be extended when we add new operators to CCS. To this end, we use *structural operational semantics (SOS)*, a way of describing the behavior of a term using inference rules.

As a reminder, inference rules are just a structured way of writing implications. For example, the inference rule shown in Figure 1.4, named *example*, states that if A and B hold, then C must hold.

For the semantics of CCS_0 , we only need three such rules, shown in Figure 1.5.

Definition 2 (Semantics of CCS_0). Let $\llbracket \cdot \rrbracket : CCS_0 \rightarrow LTS$ be defined as $\llbracket P \rrbracket := (CCS_0, \rightarrow, P)$, where $\rightarrow$ is the smallest relation $\rightarrow \subseteq CCS_0 \times Act \times CCS_0$ that adheres to all inference rules shown in Figure 1.5.

We call $\llbracket P \rrbracket$ the semantics of P .

Working with CCS Inference Rules

How do these rules now work in practice? Let us look at our first branching CCS process we had seen before, the wet match:

$$\text{strike}.\text{(burn.extinguish.0 + smolder.0)}$$

You have seen a sketch of the resulting LTS already in Figure 1.2, but now, we will systematically construct it together, using the structural operational semantics.

As you can easily see from the definition of the semantics function, $\llbracket \text{strike}.\text{(burn.extinguish.0 + smolder.0)} \rrbracket$ is an LTS with an initial state called $\text{strike}.\text{(burn.extinguish.0 + smolder.0)}$. The interesting question is which transitions and states we can reach from this state.

To determine this, we try to apply all our SOS rules to this CCS term. This works by looking at the main operator in the CCS term (the one that would be at the root node when drawing a

Figure 1.5: The inference rules for CCS_0

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \text{ prefix} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \text{ choice.l} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \text{ choice.r}$$

Figure 1.6: The LTS for our wet match, under construction

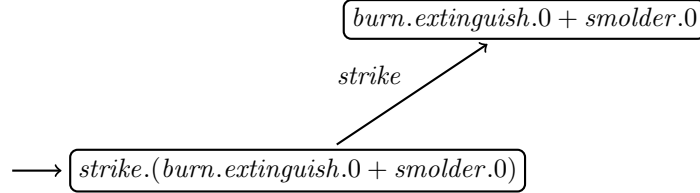
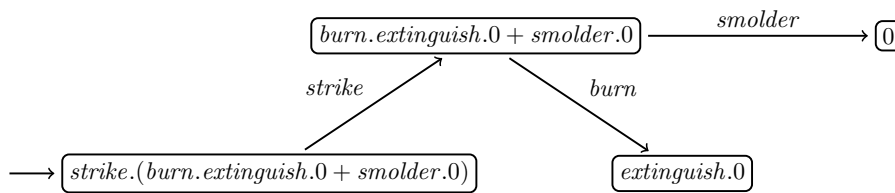


Figure 1.7: The LTS for our wet match, under construction



syntax tree), then trying to match all SOS rules that apply to this operator. If successful, the rule tells us which premises we have to check (if any), and which transition we get (possibly depending on the premises). Let's try this!

🔗 Interactive Exercise

book.pseuCo.com

In this interactive exercise, you will build your very first (and very simple) inference tree using the SOS rules of CCS.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-first-example-1>

There was only a single way to solve this, so we have found a single outgoing transition from the initial state. We can sketch what we know so far, shown in Figure 1.6.

We have discovered a new state, *burn.extinguish.0 + smolder.0*. This means we are not done yet – we must also look for outgoing transitions from this state!

It's your turn again – find two different ways to complete this inference tree:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, you will encounter the first time where two different SOS rules can be applied.

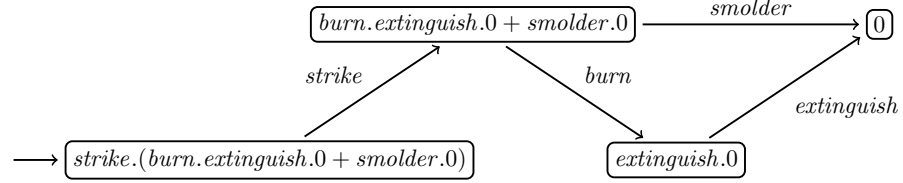
Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-first-example-2>

We can now add the two new transitions we have discovered to our LTS, shown in Figure 1.7.

We have discovered two more states, so we must look for further outgoing transitions. First, let us check *extinguish.0*:

Figure 1.8: The LTS for our wet match, still under construction?



🔗 Interactive Exercise

book.pseuCo.com

In this exercise, you will continue discovering transitions in our running example.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-first-example-3>

When we add the resulting transition to the LTS, we have to be careful: It's target state, 0, is already in our LTS, so we should not add a new state, but connect to it, as shown in Figure 1.8. (This is a difference to the sketch we had shown in Figure 1.2, but this difference does not matter in practice.³)

For our next trick, we'll look at the last state we have not analyzed yet, 0:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, you will look at the last state in our running example.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-first-example-4>

As you will quickly have noticed, none of the rules apply to the process 0! Is this a mistake, an oversight?

No, this behavior is simply as designed. The process 0 (which is called *stop*, as you may remember) has no outgoing transitions, so no rule is required for it. (This may sound confusing, but think about it like this: Just as there is no rule that specifies that $a.0$ does *not* have an outgoing transition with the action b , there is no need to specify that 0 has no outgoing transition with action b , or any outgoing transition, for that matter. As per our definition, no transition exists, unless we can justify it using the SOS rules!)

So, we have now looked for outgoing transitions from all reachable states and added them to the LTS. Are we done? Is the LTS sketched in Figure 1.8 complete?

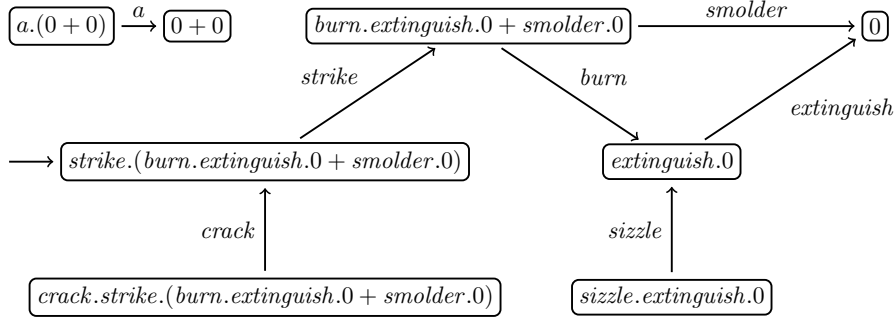
Formally, we are not done. Remember how we defined the formal semantics of CCS? The resulting LTS contains *all* CCS_0 terms and *all* transitions derivable by the SOS rules, as sketched in Figure 1.9.

These additional, unreachable states and transitions clearly do not contribute any meaning to the model and are quite distracting, so we will generally not draw them when representing an LTS graphically. Therefore, we consider the graphical representation shown in Figure 1.8 to be the canonical representation of $\llbracket \text{strike}.(burn.extinguish.0 + smolder.0) \rrbracket$.

To finish off this section, you'll apply the steps you have seen here to a new example:

³Later on, we will spend a whole chapter discussing which differences matter, and which do not, see Chapter 2.

Figure 1.9: The LTS for our wet match, with some of its whole, ungodly glory revealed



Interactive Exercise

book.pseuCo.com

In this exercise, you will construct a full LTS based on a CCS term yourself.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-lts-reconstruction>

In this exercise, you've also seen a neat trick: The state $0 + 0$ has no outgoing transitions, so it is, in a certain sense⁴, equivalent to 0 , but it is a different CCS_0 term and therefore a different state in the LTS. This trick allows us to create multiple states with identical behavior if desired and means we could now create a CCS_0 term that perfectly matches the shape of Figure 1.2 – although that is typically not needed.

1.3 Sequential CCS (with Recursion)

In CCS_0 , we are only able to model processes that consist of nothing but sequences of actions, possibly with nondeterministic branching. Everything else is out of reach so far. The most notable absence is that all processes described in CCS_0 are finite, and only have finite traces. This is easy to see, as every SOS rule of CCS_0 creates a transition that shortens the process, so no CCS_0 term can permit a trace that is longer than the number of operators in the term. Consequently, all LTS resulting from CCS_0 semantics are acyclic.

This is a severe restriction: Many real (concurrent) processes are cyclic; sometimes in small ways (e.g. because they might retry failed actions in some way), sometimes in large ways (e.g. because they are a reactive system that never terminates, interacting with its environment and regularly returning to its initial state, like a vending machine). Therefore, our first expansion of CCS will tackle this, by allowing us to use *Recursion*.

Diving Deeper

The way we add recursion here differs from Robin Milner's original approach: Instead of adding a recursion operator, we use bindings and recursion variables. In the end, both approaches are equally expressive, but our approach can be a bit more readable. If you want to see how Milner defined recursion, read his original paper and compare it with our definition!

⁴We will formalize this in a later chapter.

Figure 1.10: A more realistic match that can repeatedly burn things



Conceptually, (like in programming languages,) recursion requires two things: One must be able to assign a name to some sub-process (or procedure), and be able to refer to it, or rather jump into it, later on – notably also within said process itself. Let’s look at an example, namely our wet match from the previous section:

$$\text{strike}.\text{(burn.extinguish.0 + smolder.0)}$$

This model has a weakness: The match burning is modelled as a single action. As actions are instantaneous and timeless, this is not a good model; and this will become much more relevant when we later start to connect different processes to each other: In the real world, a burning match burns for some time, and can be used to light a (in principle) unbounded number of things until it extinguishes.

So, a more accurate model of our match would be able to perform the *burn* action repeatedly, until it eventually does *extinguish* and enters a state in which a *burn* is no longer possible, as sketched in Figure 1.10.

Now, let’s consider how recursion could be used to describe this system in CCS. As discussed previously, recursion allows us to “jump back” to a previously-defined CCS term, representing a state of the system. In this case, the action that can be seen as jumping back is the *burn* action, and it jumps to the state where the match can either burn (again) or extinguish. We can write this down using a recursive formula like this:

$$\text{BurningMatch} := \text{burn.BurningMatch} + \text{extinguish.0}$$

In other words, a burning match is a match that can either *burn* and is then still a burning match, or it can *extinguish* and has no more behavior afterwards. As before, we can now model our wet match using CCS:

$$\text{strike}.\text{(BurningMatch + smolder.0)}$$

Of course, what we have written above is not a valid CCS_0 term (and our semantics does not know how to incorporate a recursive formula at all). So, before we can analyze this process in more detail, we have to formalize the new CCS dialect we have created here.

1.3.1 Syntax

Definition 3 (Sequential CCS (CCS_0^ω)). Let Act be a set of actions and Var be a set of recursion variables. The set of expressions in CCS_0^ω is defined as

$$P ::= 0 \mid X \mid P + P \mid \alpha.P$$

provided that $\alpha \in Act$ and $X \in Var$.

For recursion variables, we always use names that start with capital letters, typically either combinations of words (like *BurningMatch*) or letters like X .

1.3.2 Semantics

To define the semantics of CCS_0^ω , we first need to add our recursive definitions into the mix. Formally, we simply collect all definitions that should be used into a partial function $\Gamma \in Var \rightarrow CCS_0^\omega$ (though we rarely write it down in set notation, typically using a list of defining equations instead). We call Γ the *binding*.

Figure 1.11: The SOS rules for CCS_0^ω

$$\begin{array}{c}
 \frac{}{\alpha.P \xrightarrow{\alpha} P} \text{prefix} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \text{choice.l} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \text{choice.r} \\
 \\
 \frac{P \xrightarrow{\alpha} P' \quad \Gamma(X) = P}{X \xrightarrow{\alpha} P'} \text{rec}
 \end{array}$$

Next, we need to add a new SOS rule that enables processes like X to have outgoing transitions. Intuitively, we want to say that X can perform a transition if X is bound to some CCS process and that CCS process can perform the transition in question. This new rule, along all other inference rule, is shown in Figure 1.11.

With the new rule in place, we can now define the semantics of CCS_0^ω . Our semantics function will now take the binding as another argument:

Definition 4 (Semantics of CCS_0^ω). Let $\llbracket \cdot \rrbracket : (Var \rightarrow CCS_0^\omega) \rightarrow CCS_0^\omega \rightarrow LTS$ be defined as $\llbracket P \rrbracket_\Gamma := (CCS_0^\omega, \rightarrow_\Gamma, P)$, where $\rightarrow_\Gamma$ is the smallest relation $\rightarrow_\Gamma \in CCS_0^\omega$ that adheres to all inference rules shown in Figure 1.11.

We call $\llbracket P \rrbracket_\Gamma$ the semantics of P under binding Γ .

Let's look at these rules in practice and reconsider our updated model of our wet match, given by the binding containing

$$BurningMatch := burn.BurningMatch + extinguish.0$$

and this process:

$$strike.(BurningMatch + smolder.0)$$

Exercise

Look at the CCS_0^ω model of a wet match given above.

Without considering the formal semantics, sketch an LTS showing how you think the wet match should behave. Once you have constructed the LTS formally in the next exercise, compare both results. Are there differences? If so, are they relevant? Which model seems to be more correct?

Interactive Exercise

book.pseuCo.com

Now, it is time to formally apply our new recursion rule for the first time.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-omega-lts-reconstruction>

Note that with CCS_0^ω , we can now also take control of the naming of our state in the LTS, as demonstrated in the LTS you constructed above which contained a state named *BurningMatch*.

1.3.3 The Power of Sequential CCS

CCS_0^ω is a powerful calculus. With the ability to encode sequences of actions, choice between multiple alternatives, and the ability to return to previous states, it seems possible to encode any structure a model might have. And indeed, this is true, at least in a certain sense:

Lemma 1. Let $LTS = (S, \rightarrow, s_0)$ be an arbitrary, finite LTS with $S \subseteq Var$. Then, there is a CCS_0^ω term P and a finite binding Γ such that LTS and $\llbracket P \rrbracket_\Gamma$ fully agree in their reachable fragments.

In other words, for any finite LTS where all state names are legal recursion variables, there is a CCS_0^ω term that corresponds exactly to this LTS, at least in the reachable parts. (And consequently, if you only care about the shape and transitions of the reachable part, you can describe any LTS with a CCS_0^ω term.⁵)

The finiteness of the LTS is key, for infinite LTS, making such a promise is more difficult. Conversely, finite bindings are guaranteed to yield a finite LTS:

Lemma 2. Let P be a finite CCS_0^ω term and Γ be a finite binding. Then, $Reach(\llbracket P \rrbracket_\Gamma)$ is finite.

1.3.4 Guardedness & Termination of CCS Semantics

With the great power of CCS_0^ω comes great responsibility, or rather a catch. So far, attempting to infer transitions in the interactive exercises might have been tedious, but it was also simple: If in doubt, mindlessly trying to apply the different rules available was guaranteed to eventually lead you to a complete inference tree. This observation gives rise to a naïve way of inferring *all* outgoing transitions of a CCS process:

To compute the outgoing transitions, you attempt to apply all available SOS rules to the process. For every rule that applies, try to recursively work on all premises, until you either complete the inference tree or fail. Using either depth- or breadth-first search, collect all possible inference trees.

This approach is called the *naïve implementation of CCS semantics*, and for CCS_0 , it is a perfectly valid approach. Now, however, an annoying problem can arise. As an example, consider the binding $X := X + a.X$, and the process X . As an experiment, let's try to derive a transition for X using only the *choice_l* rule:

Interactive Exercise

book.pseuCo.com

In this exercise, you'll experience the problem a naïve implementation of CCS semantics can run into.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-omega-infinite-recursion>

As you have seen, this process can diverge: You can create a forever-growing inference tree fragment, repeatedly looking up the process X and wittling it down to another recursion variable.

A CCS term having this problem does not make it invalid in any way. In fact, it is perfectly possible to apply our semantics to it, provided you take care not to get lost in an infinite loop. Let's construct the LTS for our example:

Interactive Exercise

book.pseuCo.com

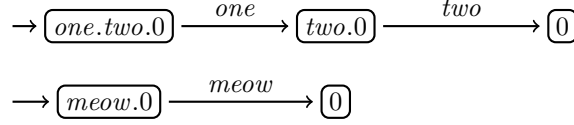
In this exercise, you'll construct an LTS for a CCS term that allows you to descend into eternal madness, so beware!

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-0-omega-infinite-recursion-lts-reconstruction>

⁵We will formally capture this in Section 2.3.

Figure 1.12: A counting human and a cat, isolated



As you'll have seen, constructing the LTS is rather easy, and the trick is essentially to just ignore the $X +$, which is *unproductive*: You can follow this path, and you may be able to create new, completed inference trees, but they all yield a transition that you can also derive by using fewer steps.

In general, cycles such as $X := X$ do not contribute transitions, but allow infinite inference trees. They can come alone (in which case no transitions are possible) or in conjunction with other CCS operators that permit a transition; and they can also come as larger cycles that span multiple recursion variables, like $X := Y$ and $Y := X$.

Either way, these cycles are a nuisance. What distinguishes them from “good”, unproblematic uses of recursion like $X := a.X$ is that the recursion variable occurs as the top element of a binding or within a choice operator, but not under a prefix term. Conversely, usage of recursive variables within a prefix term is harmless.

We call CCS terms that only use recursive variables safely *guarded*:

Definition 5 (Guardedness). A recursion variable X is called *guarded* in a CCS process P if every occurrence of X is within a term of the form $\alpha.Q$. Otherwise, X is called *unguarded*. A CCS term P is called *guarded* if all occurring recursion variables are guarded in P , otherwise, it is called *unguarded*. If a binding Γ only maps onto guarded CCS terms, we call it *guarded*, otherwise, we call it *unguarded*.

For guarded CCS_0^ω terms, the naïve implementation of CCS semantics terminates.

Similarly, if a finite binding Γ is guarded, the naïve implementation of CCS semantics can construct the whole reachable fragment of $\llbracket P \rrbracket_\Gamma$ for any $P \in CCS_0^\omega$.⁶

PseuCo.com internally uses the naïve implementation of CCS semantics. You can easily see this in action: If you attempt to open our previous example in pseuCo.com, it is unable to generate any transitions and throws an error, despite the LTS certainly having a transition!

1.4 CCS: The Full Monty

As we've seen previously, CCS_0^ω is quite powerful, and notably powerful enough to express any finite LTS. However, this does not mean that CCS_0^ω can model all finite systems *well*. The missing part is structural: CCS_0^ω does not have any way of expressing that multiple systems run concurrently.

On paper, this is not a problem. We have seen how we can handle concurrency before, on an LTS level, by parallel composition of LTS. As we can describe any finite LTS using CCS, we can, in a certain sense, model concurrency.

As an example, let's consider a minimal concurrent system, where there are two components, a human counting “one”, “two”; and a cat doing “meow”. We can easily model both these components using CCS or LTS, shown in Figure 1.12.

Using parallel composition of LTS and the interleaving model, we can also easily create an LTS describing a system consisting of the human and the cat, shown in Figure 1.13.

⁶Note that this even holds if P itself is unguarded.

Figure 1.13: A counting human and a cat, happily sharing an LTS

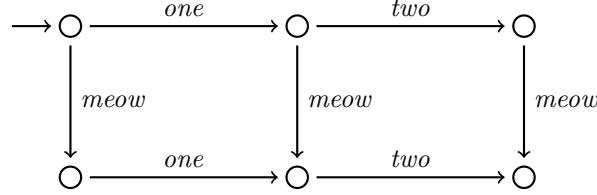


Figure 1.14: SOS rules for simple interleaving-based parallel composition

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{par.l} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \text{par.r}$$

Of course, as this LTS is finite, it can be expressed in CCS_0^ω . In fact, CCS_0 suffices:

🔗 Interactive Exercise

book.pseuco.com

In this exercise, you will manually write a CCS term that describes the result of parallel execution of LTS

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-parallel-execution-with-ccs-0-omega>

This model of describing concurrency in CCS has multiple drawbacks, some of which are more obvious than others:

- Creating such a model is cumbersome and effectively requires describing all traces through the combined system manually.
- This construction method is not compositional: If you have bindings *Human* and *Cat*, they cannot be used directly to construct the model describing parallel execution of both.
- Describing concurrency in this way does not bring us any closer to modelling systems where multiple components can communicate and interact.

We need to resolve these issues, especially the last one, to create the proper, full *CCS*. It is called the Calculus of *Communicating Systems*, after all!

1.4.1 Parallel Composition and Communication

So, our goal is to allow modelling parallel execution of CCS processes, directly on the CCS level. In a first step, this is quite simple: We just add a new operator, called the *parallel* operator, like this: When P and Q are CCS terms, then $P \mid Q$ is a CCS term describing their parallel execution.

To breathe life into this, we can use the interleaving model of concurrency and simply add two SOS rules allowing both sides of a parallel operator to move independently (without disturbing each other, unlike with the choice operator), shown in Figure 1.14.

With this approach, we make creating models of concurrent systems easy and compositional, for example, we can now model the human-and-cat-system shown in Figure 1.13 very easily as $\text{one.two.0} \mid \text{meow.0}$.

Figure 1.15: SOS rule for synchronization. Note that a only ranges over communication actions Com , so τ is not allowed in this rule.

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{ sync}$$

However, this approach still does not allow us to model systems where multiple components communicate or interact. So far, both sides of the parallel operator are completely disjoint, they are not aware of each other and simply run in isolation, with the resulting transitions interleaved.

To improve on this, we must create a way for multiple concurrently-running processes to talk to each other. There is a multitude of ways of doing so, we will follow Robin Milner and choose an interaction model called *handshaking*. Handshaking works by having two processes execute a *matching pair* of actions at the same time.

Therefore, we must change our set of actions Act to contain such matching pairs. In doing so, we will not touch our internal action τ , as internal actions are, well, internal, and thus not suitable for communicating with others. Our communication actions, however, will now be split up: We set $Com := \mathcal{A}^! \cup \mathcal{A}^?$ where $\mathcal{A}^!$ is a set of actions like $act!$ ending in an exclamation mark and $\mathcal{A}^?$ is the corresponding set such that $act? \in \mathcal{A}^?$ if and only if $act! \in \mathcal{A}^!$. For any $a \in Com$, we use $\bar{a}$ to denote the matching action (with $!$ and $?$ swapped). For completeness, we set $\bar{\tau} := \tau$.

So, from now on, all actions besides τ in our CCS processes will end with $!$ or $?$.⁷

In principle, the symbols $!$ and $?$ are chosen arbitrarily and can be exchanged if done consistently. In practice, we will typically use $!$ actions for things that are active, like “sending a signal”, and we use $?$ actions for things that are more passive, like “receiving a signal”.

We can now add a new inference rule that allows such matching pairs of actions to be executed at the same time. When this happens, the resulting transition is an internal one: Handshaking is something that happens between exactly two processes, and no-one else should be able to observe any details of the handshake. The new rule is shown in Figure 1.15.

This gives us all the tools we need to describe parallel execution of systems that interactively communicate with each other. Let’s try this with an example!

We’ll consider a very simple vending machine. It waits for a coin, then emits a coffee. We’ll pair it up with a user that purchases a coffee, then emits a greeting. We can easily model both in CCS:

$$\begin{aligned} User &:= coin!.coffee?.morning!.0 \\ Machine &:= coin?.coffee!.Machine \end{aligned}$$

This *almost* gives us what we want. If we look at the CCS term $User \mid Machine$, we can now use the new *sync* rule to derive transitions where user and vending machine interact; but unfortunately, both can still execute the actions on their own! In fact, this system has three transitions originating from the initial state:

- a transition derived using *sync*, describing the user inserting a coin into the vending machine
- a transition derived using *par_l*, describing the user throwing a coin away (with no involvement from the vending machine)
- a transition derived using *par_r*, describing the vending machine receiving a coin materialized out of thin air (with no involvement from the user)

⁷In the version of CCS implemented in `pseuCo.com`, simple actions without $!$ or $?$ can still be used, but they cannot participate in any handshaking whatsoever.

Figure 1.16: SOS rule for restriction

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin H}{P \setminus H \xrightarrow{\alpha} P' \setminus H} \text{res}$$

This problem is not trivially solvable: We need the *par_l* and *par_r* rules for the actions that are *not* intended to handshake, like the action *morning!* in our example. Thus, we need a way to *selectively* suppress the unwanted transitions. We do this by adding one more operator to CCS, the *restriction operator*: If P is a CCS process, we write $P \setminus H$ to describe the process that behaves like P , except that all actions in the set $H \subseteq \text{Com}$ are banned. Note that we do not allow banning the internal action τ ; and in addition, we will require that $a \in H \Leftrightarrow \bar{a} \in H$, i.e. actions must be banned pairwise. This gives rise to a handy shortcut: Instead of listing both actions $a!$ and $a?$, we will often simply write a .⁸

This new operator, of course, must be given meaning through a new inference rule, shown in Figure 1.16.

This now allows us to describe our example system as $(\text{User}|\text{Machine}) \setminus \{\text{coin!}, \text{coin?}, \text{coffee!}, \text{coffee?}\}$.

Let's see the new rules in action before we formally define our new variant of CCS. First, to test the effect of the *res* rule, try to derive a transition from this process where the user and machine do *not* synchronize, but try to execute *coin!* or *coin?* on their own:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we'll have a run-in with the CCS police.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-deriving-restricted-transitions>

With this in mind, we can now find all transitions that are not banned to construct the full LTS:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we will construct the full system consisting of our thirsty user and the vending machine.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-sync-vending-machine-user>

Note that while constructing the LTS, you had to derive transitions with *coin*- and *coffee*-based actions above the restriction operator. This is perfectly fine if we use the *sync* rule as it “hides” the action behind an internal τ action, preventing the restriction operator from blocking the transition.

We have now collected all the tools required to enable communication in CCS. The resulting CCS dialect is simply called *CCS*, and in the following sections, we'll formally collect the building blocks we discussed so far.

1.4.2 Syntax

Definition 6 (CCS (*CCS*)). Let Act be a set of actions with $\text{Act} = \text{Com} \cup \{\tau\}$, and $\text{Com} := \mathcal{A}^! \cup \mathcal{A}^?$ with $\text{act?} \in \mathcal{A}^? \Leftrightarrow \text{act!} \in \mathcal{A}^!$. Let Var be a set of recursion variables. The set

⁸In pseuCo.com syntax, using this shortcut is mandatory.

Figure 1.17: The SOS rules for *CCS*

$$\begin{array}{c}
 \frac{}{\alpha.P \xrightarrow{\alpha} P} \text{prefix} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \text{choice.l} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \text{choice.r} \\
 \\
 \frac{P \xrightarrow{\alpha} P' \quad \Gamma(X) = P}{X \xrightarrow{\alpha} P'} \text{rec} \quad \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin H}{P \setminus H \xrightarrow{\alpha} P' \setminus H} \text{res} \\
 \\
 \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{par.l} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \text{par.r} \\
 \\
 \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{sync}
 \end{array}$$

of expressions in *CCS* is defined as

$$P ::= 0 \mid X \mid P + P \mid \alpha.P \mid P \mid P \mid P \setminus H$$

provided that $\alpha \in \text{Act}$, $X \in \text{Var}$, and $H \subseteq \text{Com}$ with $a \in H \Leftrightarrow \bar{a} \in H$.

To reduce the number of parentheses, we use the following precedence rules:

- $P + Q + R \rightsquigarrow (P + Q) + R$, i.e. the choice operator is left-associative
- $\alpha.\beta.P \rightsquigarrow \alpha.(\beta.P)$, i.e. prefix is right-associative
- $P \mid Q \mid R \rightsquigarrow (P \mid Q) \mid R$, i.e. the parallel operator is left-associative
- from highest to lowest precedence, the operators are ordered as follows:
 1. prefix ($\cdot$)
 2. restriction ($\setminus$)⁹
 3. choice ($+$)
 4. parallel ($\mid$)

1.4.3 Semantics

Definition 7 (Semantics of *CCS*). Let $\llbracket \cdot \rrbracket : (\text{Var} \rightarrow \text{CCS}) \rightarrow \text{CCS} \rightarrow \text{LTS}$ be defined as $\llbracket P \rrbracket_{\Gamma} := (\text{CCS}, \rightarrow_{\Gamma}, P)$, where $\rightarrow_{\Gamma}$ is the smallest relation $\rightarrow_{\Gamma} \subseteq \text{CCS} \times \text{Act} \times \text{CCS}$ that adheres to all inference rules shown in Figure 1.17.

We call $\llbracket P \rrbracket_{\Gamma}$ the semantics of P under binding Γ .

You have seen all these operators and rules before, so nothing in this definition should come as a surprise.

In the next section, we will analyze some properties of *CCS*.

1.4.4 Operator Lifetime and Regularity

The newly added operators in *CCS* have a property that has potentially surprising side effects, ones that have a remarkable influence on the power of *CCS*.

Briefly return to Figure 1.17, and for each rule, look at the transition it creates. Check the right-hand side (the target process) – is the top-level operator from the source process still in the target process?

⁹The high precedence of the restriction operator may be surprising. It is a good habit to *always* enclose the left hand side of this operator in parantheses.

🔗 CCS SOS rules: Where did the operators go?

book.pseuCo.com

In which of the following CCS rules is the top-level operator of the source process still part of the target process?

- (a) prefix
- (b) choice_l
- (c) choice_r
- (d) rec
- (e) res
- (f) par_l
- (g) par_r
- (h) sync

Solve this quiz and check your solutions online:

<https://book.pseuco.com/#/quiz/ccs-static-operator-rules>

As you have seen, all the old operators do not occur in the target process, but the new ones do! This makes the CCS_0^ω operators ephemeral, in a sense: Each transition of a CCS process “burns” an operator, so each process eventually decays to stop (or some stop-like process like $0 + 0$). The only escape hatch is recursion, but recursion only allows you to return to a process from the binding. As we had already seen in Section 1.3.3, this makes the reachable part of CCS_0^ω finite: It only contains the initial process, plus the processes from the binding, plus a finite “appendix” of a length limited by the number of operators in the respective CCS term.

Our new operators are very different: They do not disappear when the process makes a transition, instead influencing the behavior of a process for its lifetime (by restricting its actions, or keeping it in parallel to another process).

First, let’s formalize the difference:

Definition 8 (Static and Dynamic Operators). A CCS operator is called *static* when it occurs in the target process of all SOS inference rules where it occurs in the source process.

A CCS operator is called *dynamic* when it never occurs in the target process of any SOS inference rule where it occurs in the source process.

Note that this means that not all CCS operators are either static or dynamic: 0 is both, because it does not enable any transition. We could also imagine operators that only disappear when performing some transitions. These would be neither static nor dynamic, but we have not seen such an operator yet.

The behavior of our new, static operators comes with remarkable consequences when used strategically in conjunction with recursion. First of all, let us analyze the process X with the following binding:

$$X := 0 \mid a!.X$$

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we will investigate how using recursion below a static operator affects CCS semantics.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-prefix-recursion-below-parallel>

As you have seen, the resulting LTS has surprisingly many reachable states: X , $0|X$, $0|(0|X)$, $0|(0|(0|X))$, and so on. The LTS has infinitely many reachable states – despite the binding being finite.

This phenomenon is possible only because we are using a non-dynamic operator, which allows the operator to “stack” when we go into recursion. If you try creating the same phenomenon with a dynamic operator, it is “used up” before we enter recursion, which creates an infinite loop, but does not create any fresh states:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we try to create infinitely many states using a dynamic operator.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-prefix-recursion-below-choice>

An even weirder phenomenon can also come up. Let’s look at a slightly different binding:

$$X := X \mid a!.0$$

Let’s try to find some transitions for it:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we try to find transition of our process without getting lost.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-infinite-branching>

As you have seen, this process behaves differently: It accumulates parallel operators *within a single derivation*, allowing transitions to an infinite number of target states from the single initial state. Our resulting LTS is, therefore, not only *infinite*, but also *infinitely branching*.

Infinitely branching LTS are quite unwieldy: Not only are they infinite, but we also know the naïve implementation diverges for the infinitely branching states. In fact, even a non-naïve implementation of CCS semantics diverges: The potential loop in the derivation tree is not an irrelevant artifact, but it is productive, giving rise to our infinite number of outgoing transitions.

Before, we have seen that recursive variables being used outside of prefix terms can cause the naïve implementation of CCS semantics to diverge, and coined the concept of *guarded* CCS terms for terms that are safe against this.

While guarded CCS terms are free from (reachable) infinitely branching states (as infinite branching requires a possible loop in the derivation, which is prevented by guardedness), guardedness does not guarantee finiteness – our first example of an infinite LTS shown above was guarded!

While infinite LTS can be quite useful, we want to have a mechanism for enforcing finiteness on a syntactic level if required. The key observation is that infinity can only be gained when recursion is used below an operator that is not dynamic, as this allows us to recurse into a process in a context where some operator will be added to the resulting transition’s target process.

Definition 9 (Regular CCS). We call a CCS term *regular* if it can be built using the following grammar:

$$\begin{aligned} P &::= 0 \mid X \mid P + P \mid \alpha.P \mid R \\ R &::= 0 \mid R + R \mid \alpha.R \mid R \mid R \mid R \setminus H \end{aligned}$$

We analogously extend this definition to regular CCS_0 and $CCS_0^{\omega 10}$.

This syntactically removes the ability to use any of our new, non-dynamic operators until we move to the syntactic category R , which irrevocably removes the option of using recursion for that part of the syntax tree.

Lemma 3. If Γ is a finite binding, and $\Gamma(X)$ is regular for every $X \in Var$, then $Reach(\llbracket P \rrbracket_\Gamma)$ is finite for any $P \in CCS$.

Note that the lemma above does not need to check whether P itself is regular.

1.5 CCS with Values

By this point, we have CCS , a powerful language for describing finite or infinite systems that are composed of communicating components.

Our communication model is both fundamentally basic and powerful: While CCS , by itself, only contains one basic type of communication (namely exactly two processes performing a matched pair of actions concurrently), many other forms of communication can be expressed by that. For example, it is possible to describe systems

- where three or more components must perform a handshake-like step,
- where a message is broadcast and can be received by any number of recipients (including zero) that are listening at that time,
- where a message is sent asynchronously, to be picked up by one or more receivers later on, or
- where a handshake transmits data from one component to another.

Some of these are easier to model than others, but generally, these concepts can be broken down into multiple communication steps. For example, a three-way “handshake” can be simulated by the three components first registering with a management component through a handshake, then waiting for the management component to report that all three components have arrived.

The last example, handshaking with data transmission, can be modelled using a slightly different approach.

Let’s explore that on a concrete example: Our coffee vending machine. Our earlier model, described simply by the binding $Machine := coin?.coffee!.Machine$, modelled payment in the most simplistic way possible, with the machine only requiring (and accepting) a single coin. Of course, reality is a lot more complex – we will now add a bit of this complexity to our model:

- Not all coins are equal – some are worth more than others.
- If the price of a coffee has not been paid yet, more coins are required before a coffee is dispensed.
- Once enough money has been paid, no more coins are accepted, and a coffee is dispensed.
- If the last inserted coin caused the coffee to be overpaid, some change is given.

How can we model this in CCS ? There are, of course, multiple options. One of them is to abstract away the concrete coin values, like so:

$$\begin{aligned} Machine &:= coin?.Machine + coin?.PaidMachine \\ PaidMachine &:= coffee?.(change!.Machine + \tau.Machine) \end{aligned}$$

¹⁰Plus any future CCS dialect we will encounter: P is a clone of the original syntax with all non-dynamic operators removed and the jump to R added, and R is a clone of the original syntax with recursion removed.

This model is not too bad! You can play around with it in `pseuCo.com` if you wish. It correctly models the state changes the machine can make, from coin acceptance to giving change.¹¹ But it is abstract in the data, and thus *overapproximates* the coffee machine: Instead of saying *whether* the coffee machine will require another coin or dispense a coffee, the model simply describes that both options exist.

Overapproximation and abstraction are powerful tools for simplification, and are indispensable when creating useful models!¹² Abstracting away data often is the right choice. But when the data is a crucial part of the model, because it influences the behavior of the system in ways that are critical for the properties we want to analyze, this is no longer an option. What do we do in this case?

In principle, *CCS* is powerful enough to still encode this. Let's say our vending machine expects to be paid 5 *money units*¹³, and there are coins worth 1, 2, and 5 money units. We can simply use three different actions to describe these three coins, *coin₁!*, *coin₂!*, and *coin₅!*, then describe how the machine behaves in each case:

$$\begin{aligned} \text{Machine}_0 &:= \text{coin}_1?.\text{Machine}_1 + \text{coin}_2?.\text{Machine}_2 + \text{coin}_5?.\text{Machine}_5 \\ \text{Machine}_1 &:= \text{coin}_1?.\text{Machine}_2 + \text{coin}_2?.\text{Machine}_3 + \text{coin}_5?.\text{Machine}_6 \\ \text{Machine}_2 &:= \text{coin}_1?.\text{Machine}_3 + \text{coin}_2?.\text{Machine}_4 + \text{coin}_5?.\text{Machine}_7 \\ \text{Machine}_3 &:= \text{coin}_1?.\text{Machine}_4 + \text{coin}_2?.\text{Machine}_5 + \text{coin}_5?.\text{Machine}_8 \\ \text{Machine}_4 &:= \text{coin}_1?.\text{Machine}_5 + \text{coin}_2?.\text{Machine}_6 + \text{coin}_5?.\text{Machine}_9 \\ \text{Machine}_5 &:= \text{coffee}?.\text{Machine}_0 \\ \text{Machine}_6 &:= \text{coffee}?.\text{change}_1!. \text{Machine}_0 \\ \text{Machine}_7 &:= \text{coffee}?.\text{change}_1!. \text{change}_1!. \text{Machine}_0 \\ \text{Machine}_8 &:= \text{coffee}?.\text{change}_1!. \text{change}_1!. \text{change}_1!. \text{Machine}_0 \\ \text{Machine}_9 &:= \text{coffee}?.\text{change}_1!. \text{change}_1!. \text{change}_1!. \text{change}_1!. \text{Machine}_0 \end{aligned}$$

While this option technically works, it produces extremely verbose models, typically affected by two problems:

- They often contain large sequences of choice operators branching between multiple version of essentially the same action, like our different coin insertion actions.
- The bindings are often full of essentially duplicate processes, like our different *Machine* processes above.

Seeing such models raises an important question: Can we somehow add variables to *CCS*, variables that allow us to store and transmit values from some set of values? This is indeed what we will do in this part of *pseuCo Book*.

Adding variables and values to *CCS* does *not* make *CCS* more powerful in a formal sense (at least as long as the set of acceptable values is finite), but it will make our models much more readable. Note that this only affects the *CCS* terms – the resulting LTS will not contain variables, but “unfold” everything, duplicating states in a similar way as we saw for our coin-counting vending machine above.

1.5.1 Adding Values

The core place in *CCS* where we want to add values is our communication model: We want to get from a dataless *a!-to-a?-handshake* to a handshake that can send a value. To do this,

¹¹An minor detail is the τ transition when no change is given. Without it, the machine would continue to be able to give change until a coin is inserted, while in reality, the machine takes an (internal) decision to not give change. We would consider this a flaw in our model. More details on this will finally emerge in Section 2.6.

¹²No-one wants a model of a vending machine down to its atomic level.

¹³Replace this with your favorite currency!

Figure 1.18: New SOS rules for the prefix operator

$$\frac{\alpha \in \{a!, a? \mid a \in \mathbb{C}\} \cup \{\tau\}}{\alpha.P \xrightarrow{\alpha} P} \text{ prefix}$$

$$\frac{e \Downarrow n}{a?.e.P \xrightarrow{a?n} P} \text{ value} \qquad \frac{e \Downarrow n}{a!.e.P \xrightarrow{a!n} P} \text{ output}$$

first of all, we will break the previously-existing (formal) symmetry between exclamation- and question-mark-clad actions, and clearly assign roles:

- Actions with exclamation marks, like $a!$, will be able to *send* values, and
- actions with question marks, like $a?$, will be able to *receive* values.

In our previous example, we hand-crafted actions like $\text{coin}_5!$ – now, we introduce new syntax, and will use actions like $\text{coin}!5$ or $\text{coin}?2$, with data written after the exclamation or question mark. This splits the action into three parts:

- the *channel* (in our example coin), which determines which actions can communicate with each other,
- the exclamation or question mark, determining the direction of communication, and
- the data that is transmitted.

We extend the definition of our bar operator $\bar{\cdot}$ accordingly, so for example $\overline{\text{coin}?5} = \text{coin}!5$. Our “old” actions without data will stay valid; and for simplicity, we’ll use the set of integers $\mathbb{Z}$ as the set of possible values¹⁴.

While this is not so relevant yet, we also want the ability to *compute*, especially to compute a value that will be sent. To enable this, we split our *prefix* rule into three rules, shown in Figure 1.18.

In these rules, $e \Downarrow n$ indicates that e is an arithmetic expression on the integers that evaluates to e . (If e contains a variable, it is not closed and cannot evaluate to anything, so no transition is possible.¹⁵)

In principle, this is already enough to pass some values. Notably, no modification to the sync rule is required (besides to our expansion of the $\bar{\cdot}$ operator). Let’s try it – ignoring the fact that the $a?2$ action is written with parentheses in CCS (this will make sense later):



Interactive Exercise

book.pseuCo.com

In this exercise, you’ll derive your first value passing transition.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-vp-first-passed-value>

Hooray – we’ve computed $1 + 1$ and sent the result to another CCS process!

Of course, this is still incredibly limited. Most importantly, the receiver already needed to know the result (so they could offer the action with the appropriate value)!

¹⁴In pseuCo.com, and the interactive exercises here in pseuCo Book, we will also allow Boolean values and strings.

¹⁵Formally, a CCS term like $a!(42 + x).0$ is perfectly valid and has no transitions. In pseuCo.com, such terms may not be written down, they are considered a syntax error.

The solution to this, of course, is to allow receiving *arbitrary* values. We do this by allowing prefix expressions like $a?x.0$. Note that $a?x$ is not a valid action as x is not an integer. Indeed, none of the SOS rules you have seen above apply to this term. Instead, we add a new rule that allows receiving arbitrary values, called *input*. This time, you will try to use the rule before you see it written down formally – try to find different transitions for $a?x.0$, using only the *input* rule:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, you'll have the opportunity to choose values to receive.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-vp-first-input>

As you have seen, you can choose arbitrary integers to receive. This makes the initial state of $\llbracket a?x.0 \rrbracket$ infinitely branching – and can be a headache when trying to derive a transition as it may require “guessing” the correct value to allow a synchronization to happen. Of course, in most cases, you'll be able to guess correctly – try it here:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, you will have to guess – or look ahead.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-vp-first-input-sync>

Before we can write down the *input* rule, there is one remaining question we have to tackle: How do we actually store the value that was received? So far, we dodged this question by having the receiving process simply be the stop process.

If you try to think of a solution for this, you may be tempted to come up with a complex system, maybe involving an environment that stores the value of all variables. But this would be exceptionally complicated, not only because we would need to track separate environments for separate parts of our CCS term, but also because it would not fit into the structure of our simple syntax-based SOS rules.

The solution we will choose instead is a lot simpler¹⁶: We store the received value *syntactically*. For example, consider the process $a?x.b!(x+1).0$, a simple process that tries to receive a value on channel a , increment it by one and pass the result onwards on channel b . If this process now executes the action $a?42$, we store this value right in the process – the transition yields the target process $b!(42+1).0$, to which the *output* rule now applies.

This approach moves all logic related to “handling values” into the *input* rule – when a value is read into a variable, the *input* rule is tasked with eliminating that variable. To write this down, we define a *replacement function*, written down as $P\{v/x\}$ (read as “P with v for x”) that handles the syntactical replacement of x with v .

This replacement function is therefore also responsible for defining the “scope” of variables. We will define it such that processes like $a?x.b!x.a?x.b!x.0$ are possible and behave identically to $a?x.b!x.a?y.b!y.0$, in other words, each use of a variable is tied to the syntactically closest receiving prefix term. We defer details to a later section where we formally define our new dialect of CCS.

With this replacement function, we can now easily write down our new SOS rule for receiving arbitrary values, shown in Figure 1.19.

¹⁶Some also call it a hack.

Figure 1.19: New SOS rule for receiving an arbitrary value

$$\frac{n \in \mathbb{Z}}{a?x.P \xrightarrow{a?n} P \{n/x\}} \text{input}$$

Figure 1.20: Updated version of the recursion SOS rule, now capable of passing data into the process

$$\frac{\Gamma(X[u_1, \dots, u_n]) = P \quad P \{e_1/u_1, \dots, e_n/u_n\} \xrightarrow{\alpha} P'}{X[e_1, \dots, e_n] \xrightarrow{\alpha} P'} \text{rec}$$

Even without a formal definition of replacement, you should be able to apply the *input* rule to cases where replacement comes into effect. Let's try it! Keep in mind that replacement *only* replaces variables, it does not simplify expressions, or change them in any other way.¹⁷



Interactive Exercise

book.pseuCo.com

In this exercise, you'll need to perform replacement by yourself for the first time.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-vp-input-with-replacement>

1.5.2 Process Parameters

Since we use replacement for our variables, so far, we have a problem with recursion. Imagine we want to encode our coin-counting vending machine from above using our new value-passing powers: We could happily accept a coin of any value with something like $Machine := coin?x.Machine$, but since our target process after receiving the coin is simply $Machine$, the value of the received coin is lost!

We need a way to carry values into recursion. We have already seen a way to handle this manually, by creating many variants of a process, like $Machine_5$. To integrate this idea with our new value-passing features of CCS, we introduce *process parameters*, i.e. the ability to define processes like this:

$$Machine[x] := coin?c.Machine[x + c]$$

This means that a binding Γ no longer maps recursion variables onto processes, but terms like $X[a, \dots, x]$ consisting of a process variable and any number of data variables.

Enabling this feature requires a change to our *rec* rule: It must now insert the arguments into the process it retrieved from the binding. This can easily be done using replacement, as shown in Figure 1.20.

While this rule may look intimidating at first, its behavior is very simple. It essentially is identical to the old *rec* rule, except after looking up a process in Γ , all its arguments are replaced with the parameters that were given in the CCS term. The new rule is also backwards compatible: For processes without process parameters, it derives the same transitions as the *rec* rule from *CCS*.

¹⁷PseuCo.com can try to simplify expressions. This behavior is on by default (and can be disabled using the cog wheel icon in upper right). It creates different, often smaller LTS. These represent the same behavior as the theoretically correct LTS, in a sense that we will formally define in Section 2.5.

Figure 1.21: A new SOS rule implementing conditions

$$\frac{P \xrightarrow{\alpha} P' \quad b \Downarrow \text{true}}{\text{when}(b) P \xrightarrow{\alpha} P'} \text{ cond}$$

Let's see this mechanism in action in a minimal example, the process $(a!2.0|a?x.\text{Sender}[x]) \setminus \{a\}$ with the binding $\text{Sender}[x] := b!x.0$:

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, you'll have to use the process-parameter-aware recursion rule for the first time.

Solve this exercise online:

<https://book.pseuCo.com/#/interactive/ccs-vp-process-parameter>

1.5.3 Conditional Transitions

We need to solve one last problem before we can make full use of our new value-passing features. So far, while we have values and can pass them around (both through handshakes and into recursively called processes), the values cannot really influence the behavior of the system!

Think of our coin-accepting vending machine: We can model the machine taking coins and remembering the balance, but the behavior of the versions of the machine with different balance is almost identical – they can differ in values (they could, for example, send the balance to someone else), but structurally, their behavior must be the same, offering transitions with identical channels, moving the states that once again only differ in values. What we lack is a mechanism that allows the behavior of a system to depend on a value, allowing us, for example, to model that our vending machine can brew a coffee if (and only if) its balance is above a certain amount.

So that is exactly what we add! You may expect this feature to be called **if**, like in most programming languages. However, our new CCS feature will work slightly differently than a prototypical **if**: Instead of implementing *branching*, we implement *conditions*, allowing us to disable (sub-)processes depending on a Boolean term.

It works like this: If P is a CCS process, then $\text{when}(b) P$ is a CCS process that behaves like P if b , a Boolean expression, evaluates to *true*, otherwise, it behaves like 0.

Most often, you will see a condition immediately preceding a prefix term, like $\text{when}(\text{balance} \geq 5) \text{coffee}!.0$, but they can also be used in front of all other CCS processes. Note that true branching can easily be obtained from conditions: $\text{when}(b) \text{yes}!.0 + \text{when}(!b) \text{no}!.0$

The SOS rule for conditions is quite simple and shown in Figure 1.21.

Note that $\text{when}(\cdot)$ is a dynamic operator, disappearing after performing a transition.

With conditions, we finally have all the tools we need to properly model our coin-counting coffee machine in CCS. Here, we model it together with a thirsty user that only has coins of value 2:

$$\begin{aligned} \text{Machine}[b] &:= \text{when}(b < 5) \text{coin}?c.\text{Machine}[b + c] + \text{when}(b \geq 5) \text{coffee}!. \text{ReturningMachine}[b - 5] \\ \text{ReturningMachine}[b] &:= \text{when}(b > 0) \text{change}!. \text{ReturningMachine}[b - 1] + \text{Machine}[0] \\ \text{User} &:= \text{coin}!2.\text{coin}!2.\text{coin}!2.\text{coffee?}.\text{change?}.0 \\ &(\text{Machine}[0] \mid \text{User}) \setminus \{\text{coin}, \text{change}, \text{coffee}\} \end{aligned}$$

As you can see, we now have a powerful and flexible mechanism for modelling systems that store, process, and communicate data. We call the resulting dialect CCS_{vp} (CCS with value passing). Before we formalize it in the next section, you can use the exercise below to practice using the new SOS rules on this example. Don't worry, in this exercise, the old, non-value-passing rules will be completed automatically, and you'll get some pre-filled hints for the other steps to reduce the amount of typing.

🔗 **Interactive Exercise**

book.pseuCo.com

In this exercise, you'll explore the semantics of our coin-counting coffee machine.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/ccs-vp-coin-counting-coffee-machine-and-user>

If you want to experiment with this model, you can also open it in pseuCo.com.

1.5.4 Syntax

Definition 10 (CCS with value passing (CCS_{vp}^Z)). Let $\mathbb{C}$ be a set of channels, D a set of data variables, and Var be a set of recursion variables, with all of these sets being disjoint. Let AExp and BExp be sets of arithmetic and Boolean expressions over the data variables, respectively. The set of expressions in CCS_{vp}^Z is defined as

$$\begin{aligned} P ::= & 0 \mid X[u_1, \dots, u_n] \mid P + P \mid \alpha.P \mid P \mid P \mid P \setminus H \mid \text{when}(b) P \\ \alpha ::= & \tau \mid a! \mid a? \mid a!e \mid a?e \mid a?x \end{aligned}$$

provided that $X \in Var$, $a \in \mathbb{C} \cup D$, $e \in \text{AExp}$, $b \in \text{BExp}$, $x \in D$, $u_i \in \text{AExp} \cup \mathbb{C}$, and $H \subseteq \text{Com}$ with $a \in H \Leftrightarrow \bar{a} \in H$.

We use the same precedence rules as for CCS, shown below Definition 6.

You may have noticed that the syntactic rules $a?e$ and $a?x$ for constructing actions are ambiguous. While a term like $\text{get?}(n + 42).0$ clearly contains an expression, $\text{get?}n.0$ is ambiguous: This looks like a prefix term where we receive any value into a variable n , but technically, n also is an expression, so this could have been constructed through the $a?e$ rule. This is more realistic than it may seem – think of someone trying to model a system that can receive any value, than must receive the same value again (with other options simply being impossible). How would you represent that? $\text{get?}n.\text{get?}n.0$ looks like a good model, but this could also describe a process that can receive two arbitrary values.

To resolve this ambiguity, we require that *all expressions within actions that could be misconstrued for variables must be parenthesized*. Thus, $a!x$, $a?(1 + 1)$, $a?42$, and $a?(x)$ are all valid examples of actions containing an expressions, while $a?x$ is a (non-ambiguous) example of an action containing a variable.

Another curiosity is that we allow using data variables as the channel of an action, and channels as process parameters. This is a convenience feature that allows parametrizing the channels of process definitions. For example,

$$\text{Cell}[\text{get}, \text{put}, v] := \text{get!}v.\text{Cell}[\text{get}, \text{put}, v] + \text{put?}v.\text{Cell}[\text{get}, \text{put}, v]$$

is a model of a memory cell (i.e. a CCS process that stores a value that can be read and written) where the actions to read and write can be customized, so e.g. $\text{Cell}[\text{get_}x, \text{put_}x, 0]$ then is a memory cell initialized with 0 that can be read with $\text{get_}x$ and written with $\text{put_}x$. This makes the cell reusable – different instantiations of cells can use different actions for reading and writing, allowing them to be used in parallel.

Figure 1.22: The SOS rules for $CCS_{vp}^{\mathbb{Z}}$

$$\begin{array}{c}
 \frac{\alpha \in \{a!, a? \mid a \in \mathbb{C}\} \cup \{\tau\}}{\alpha.P \xrightarrow{\alpha} P} \text{prefix} \qquad \frac{e \Downarrow n}{a!e.P \xrightarrow{a!n} P} \text{output} \\
 \\
 \frac{e \Downarrow n}{a?e.P \xrightarrow{a?n} P} \text{value} \qquad \frac{n \in \mathbb{Z}}{a?x.P \xrightarrow{a?n} P \{n/x\}} \text{input} \\
 \\
 \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \text{choice.l} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \text{choice.r} \\
 \\
 \frac{\Gamma(X[u_1, \dots, u_n]) = P \quad P \{e_1/u_1, \dots, e_n/u_n\} \xrightarrow{\alpha} P'}{X[e_1, \dots, e_n] \xrightarrow{\alpha} P'} \text{rec} \\
 \\
 \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{par.l} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \text{par.r} \\
 \\
 \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{sync} \qquad \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin H}{P \setminus H \xrightarrow{\alpha} P' \setminus H} \text{res} \\
 \\
 \frac{P \xrightarrow{\alpha} P' \quad b \Downarrow \text{true}}{\text{when}(b) P \xrightarrow{\alpha} P'} \text{cond}
 \end{array}$$

1.5.5 Semantics

Definition 11 (Semantics of $CCS_{vp}^{\mathbb{Z}}$). Let $\llbracket \cdot \rrbracket : (Var \rightarrow CCS) \rightarrow CCS \rightarrow LTS$ be defined as $\llbracket P \rrbracket_{\Gamma} := (CCS_{vp}^{\mathbb{Z}}, \rightarrow_{\Gamma}, P)$, where $\rightarrow_{\Gamma}$ is the smallest relation $\rightarrow_{\Gamma} \subseteq CCS_{vp}^{\mathbb{Z}} \times Act \times CCS_{vp}^{\mathbb{Z}}$ that adheres to all inference rules shown in Figure 1.22.

We call $\llbracket P \rrbracket_{\Gamma}$ the semantics of P under binding Γ .

You have seen all these operators and rules before, so nothing in this definition should come as a surprise.

The replacement function used in these rules is defined as follows:

$$\begin{aligned}
0 \{v/x\} &= 0 \\
(P + Q) \{v/x\} &= P \{v/x\} + Q \{v/x\} \\
(P \mid Q) \{v/x\} &= P \{v/x\} \mid Q \{v/x\} \\
(P \setminus H) \{v/x\} &= P \{v/x\} \setminus \{\alpha \{v/x\} \mid \alpha \in H\} \\
(\alpha.P) \{v/x\} &= \alpha \{v/x\}.P \{v/x\} \\
a! \{v/x\} &= \begin{cases} a! & \text{if } a \neq x \\ v! & \text{if } a = x \end{cases} \\
a? \{v/x\} &= \begin{cases} a? & \text{if } a \neq x \\ v? & \text{if } a = x \end{cases} \\
\tau \{v/x\} &= \tau \\
(a?y.P) \{v/x\} &= \begin{cases} a? \{v/x\}y.P \{v/x\} & \text{if } x \neq y \\ a? \{v/x\}y.P & \text{if } x = y \end{cases} \\
(a!y.P) \{v/x\} &= \begin{cases} a! \{v/x\}y.P \{v/x\} & \text{if } x \neq y \\ a! \{v/x\}v.P \{v/x\} & \text{if } x = y \end{cases} \\
X[y_1, \dots] \{v/x\} &= X \left[\begin{cases} y_1 & \text{if } x \neq y_1 \\ v & \text{otherwise} \end{cases}, \dots \right]
\end{aligned}$$

For simplicity, we omit the handling of multiple replacements (e.g. $P \{v_1/x, v_2/y\}$) and replacements within expressions (i.e. handling of the processes $a!e.P$ and $a?e.P$).

In the next section, look at some more details of $CCS_{vp}^{\mathbb{Z}}$.

1.5.6 Ranges for CCS in pseuCo.com

As you have seen before, processes like $get?x.P$ are very useful, especially in a context where they must synchronize with another process sending a value. When used in such a context, deriving transitions is more complicated than using a naïve implementation of CCS semantics, but with a good strategy, it is certainly possible. You can try to strategically guess, try to always derive the sending transition first, or leave the received value blank and only fill it in later when trying to assemble a *sync* rule.

PseuCo.com can handle these cases fine. It internally employs the last strategy, and thus can easily handle synchronization despite the set of possible values that could be received being infinite.

When a receiving action is used in a context where no synchronization is required (or even possible), we get an infinitely branching state in the LTS. These are harder to handle programmatically, and cannot be rendered properly in an LTS. Consequently, pseuCo.com does not support exploring such states, and will yell at you if you try.

But of course, it would sometimes be possible to analyze such processes in pseuCo.com. While this is not possible in general, pseuCo.com has added a feature to input actions that is not there in the formal definition: Range restrictions.

In pseuCo.com, you can write e.g. `a?x:0..9 . 0`, and this process can offer exactly 10 transitions, with actions ranging from `a?0` through `a?9`. Ranges can also be defined (`range` $\vdash$ `Digit := 0..9`) and used (`a?x:Digit . 0`).

1.6 Workout

Here is a quiz that checks your knowledge throughout this chapter:

🔖 CCS – Mixed Bag Questions

book.pseuCo.com

Which operators and other syntactic elements are part of CCS_0 ?

- (a) Stop (0)
- (b) Recursion Variables (X)
- (c) Prefix ($\cdot$)
- (d) Choice ($+$)
- (e) Parallel ($\parallel$)
- (f) Restriction ($\backslash$)

Which operators and other syntactic elements are part of CCS_0^ω ?

- (a) Stop (0)
- (b) Recursion Variables (X)
- (c) Prefix ($\cdot$)
- (d) Choice ($+$)
- (e) Parallel ($\parallel$)
- (f) Restriction ($\backslash$)

Which of the following are valid CCS terms?

- (a) $a?.b!.0$
- (b) $a?.(b!.0)$
- (c) $(a?.b!).0$
- (d) $(X + Y).a!.0$
- (e) $a!.(X + Y)$
- (f) $(a!.X \mid a?.0) \setminus \{X\}$
- (g) $(a!.X \mid a?.0) \setminus \{a?, a!\}$

Which CCS semantic rules have more than one premise?

- (a) prefix
- (b) choice_l, choice_r
- (c) par_l, par_r
- (d) sync
- (e) res
- (f) rec

How many states does $\llbracket (a!.0 \mid a?.0) \setminus \{a!, a?\} \rrbracket$ have?

- (a) 1
- (b) 2
- (c) 4
- (d) ∞

What behavior does the process X have in the empty binding $\Omega = \emptyset$?

- (a) $\llbracket X \rrbracket_\Omega$ is undefined
- (b) X has no outgoing transitions
- (c) X has a τ self-loop

Which of the following statements about *guarded CCS* are true?

- (a) If a CCS process P is guarded, the naïve implementation of CCS semantics terminates when computing $\llbracket P \rrbracket_\Gamma$ for any finite binding Γ .
- (b) If a finite binding Γ only maps to guarded CCS processes, the naïve implementation of CCS semantics terminates when computing $\llbracket P \rrbracket_\Gamma$ for any CCS process P .
- (c) A CCS process is guarded if recursion variables only occur within prefix expressions.
- (d) A CCS process is guarded if recursion variables do not occur below static operators.

Which of the following statements about *regular CCS* (i.e. regular CCS processed and bindings that only map to regular CCS processes) are true, provided the binding in question is finite?

- (a) Regular CCS guarantees that the naïve implementation of CCS semantics terminates.
- (b) Regular CCS guarantees that there are only finitely many reachable states.
- (c) Regular CCS does allow recursion, but restricts where it may be used.
- (d) Regular CCS does not allow static operators to be used within the binding.

Consider the binding $\Gamma = \{(X, X + a!.0)\}$. Which statements about the reachable fragment of $\llbracket X \rrbracket_\Gamma$ are true?

- (a) The naïve implementation of CCS semantics diverges while computing transitions of the initial state.
- (b) It has infinitely many states.
- (c) It is infinitely branching.

Consider the binding $\Gamma = \{(X, a!.(X \mid a!.0))\}$. Which statements about the reachable fragment of $\llbracket X \rrbracket_\Gamma$ are true?

- (a) The naïve implementation of CCS semantics diverges while computing transitions of the initial state.
- (b) It has infinitely many states.
- (c) It is infinitely branching.

Consider the binding $\Gamma = \{(X, X \mid a!.0)\}$. Which statements about the reachable fragment of $\llbracket X \rrbracket_\Gamma$ are true?

- (a) The naïve implementation of CCS semantics diverges while computing transitions of the initial state.
- (b) It has infinitely many states.

(c) It is infinitely branching.

Solve this quiz and check your solutions online:
<https://book.pseuco.com/#/quiz/ccs-mixed-bag>

Chapter 2

Equality

2.1 Introduction

In the previous chapters, we’ve learned how to describe and model the behavior of concurrent processes. This opens the door to analysing concurrent systems.

One of the simpler forms of analysis is to ask the question whether two processes, i.e. two CCS terms or their corresponding LTS, are *equal*. Being able to determine equality allows us to answer important questions:

- “Does this system follow its specification?”
- “Can I exchange this component with that one without affecting the whole system?”

Unfortunately, it is not obvious when exactly we should consider two processes to be equal.

This chapter is concerned with identifying properties a good notion of equality should have, considering multiple candidates and their relationship.

Throughout the chapter, we will consider arbitrary LTS states wherever possible, falling back to the special case of CCS processes as necessary.

2.2 Trivial Notions of Equality

2.2.1 Identity

An obvious first candidate for a notion of equality is *mathematical identity*. Under identity, we consider two states of an LTS $TS = (S, \rightarrow, s_0)$ to be equal only if they are truly identical:

$$Id(S) = \{(P, P) \mid P \in S\}$$

For example, $Id(CCS)$ is the relation that only relates identical CCS processes.

This clearly is a *valid* way to define equality, by which we mean that it is an *equivalence relation*: It is reflexive, symmetric and transitive. But is it a useful one?

Let’s consider the CCS terms $a!.b!.0$ and $a!.b!.(0 + 0)$, shown in Figure 2.1.

Inspecting (the reachable part of) the LTS shows us that both LTS are clearly not identical: Their initial state is different. However, they can only be distinguished by their state identities: Their graph representations are otherwise identical.

Figure 2.1: LTS for the CCS terms $a!.b!.0$ and $a!.b!.(0 + 0)$.

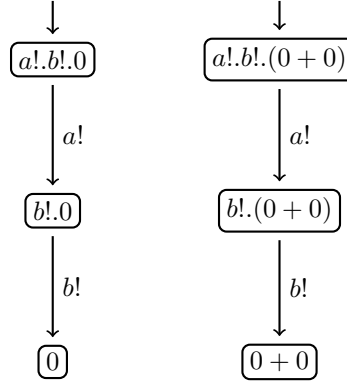
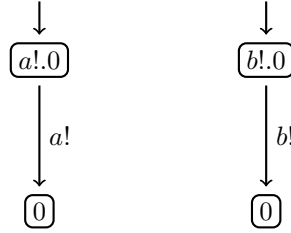


Figure 2.2: LTS for the CCS terms $a!.0$ and $b!.0$.



We've said that our model of processes is focused on (atomic) actions. We considered states as the place where time is spent, indistinguishable except through the interactions they offer. With this understanding, our two LTS model the same concurrent process. Therefore, while identity is technically a valid notion of equality, it is too strict, too finely grained to be useful in practice.

So, we need to look for notions of equality that impose more lenient requirements for when to consider processes to be equal.

2.2.2 Universal Equality

For completeness, we also briefly consider the other end of the spectrum: *universal equality*.

$$Univ(S) = \{(P, Q) \mid P, Q \in S\}$$

Under universal equality, all states are considered equal, no questions asked.

Once again, this is an equivalence relation, making it a valid way of thinking about equality. However, in this case, it is obvious that it is not useful: Clearly, there are different concurrent processes we want to be able to model, for example the ones shown in Figure 2.2, so considering them all to be equal is not useful.

2.3 Isomorphism

These considerations have clearly shown us that we need a notion of equality that is more carefully adapted to our needs: At the very least, it needs the ability to ignore the identities of states while still comparing the behaviors of the processes in some way.

One such notion of equality is called *isomorphism*. Conceptually, we will call two LTS *isomorphic* if they are identical up to renaming of states. Formally, the definition uses a bijection that identifies equal states. Note that $\text{Reach}(TS)$ denotes the set of states that are reachable from the initial state of TS .

Definition 12 (Isomorphism ($\sim_{\text{iso}}$)). We call two LTS $TS = (S, \rightarrow, s_0)$ and $TS' = (S', \rightarrow', s'_0)$ *isomorphic* ($TS \sim_{\text{iso}} TS'$) if there is a bijection $f : \text{Reach}(TS) \rightarrow \text{Reach}(TS')$ with $f(s_0) = s'_0$ and such that

$$\forall s_1, s_2 \in \text{Reach}(TS) : \forall \alpha \in \text{Act} : s_1 \xrightarrow{\alpha} s_2 \iff f(s_1) \xrightarrow{\alpha'} f(s_2).$$

Put differently, two LTS are isomorphic if there is a one-to-one mapping between their reachable states that preserves transitions and the initial state.

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we'll learn to apply the definition of isomorphism to simple examples.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/isomorphism-roulette>

Now, we need to investigate whether isomorphism is formally sound and whether it is a good equivalence in practice.

2.3.1 Equivalence Relation

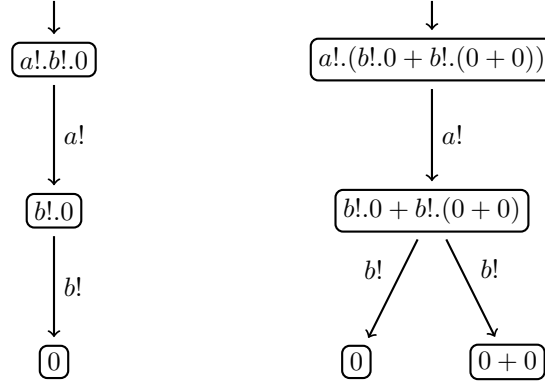
We can easily verify that isomorphism is an equivalence relation:

Proof. We consider the three properties.

- $\sim_{\text{iso}}$ is reflexive: For any LTS $TS = (S, \rightarrow, s_0)$, $TS \sim_{\text{iso}} TS$ obviously holds with the bijection $f : S \rightarrow S$ with $f(s) = s$.
- $\sim_{\text{iso}}$ is symmetric: Let $TS = (S, \rightarrow, s_0)$ and $TS' = (S', \rightarrow', s'_0)$ be LTS with $TS \sim_{\text{iso}} TS'$. This must be justified by a bijection $f : \text{Reach}(TS) \rightarrow \text{Reach}(TS')$ with $f(s_0) = s'_0$ and $\forall s_1, s_2 \in \text{Reach}(TS) : \forall \alpha \in \text{Act} : s_1 \xrightarrow{\alpha} s_2 \iff f(s_1) \xrightarrow{\alpha'} f(s_2)$. Then, the inverse relation $f^{-1} : \text{Reach}(TS') \rightarrow \text{Reach}(TS)$ is a bijection with $f^{-1}(s'_0) = s_0$ and $\forall s'_1, s'_2 \in \text{Reach}(TS') : \forall \alpha \in \text{Act} : s'_1 \xrightarrow{\alpha'} s'_2 \iff f^{-1}(s'_1) \xrightarrow{\alpha} f^{-1}(s'_2)$, so $TS' \sim_{\text{iso}} TS$.
- $\sim_{\text{iso}}$ is transitive: Let $TS = (S, \rightarrow, s_0)$, $TS' = (S', \rightarrow', s'_0)$ and $TS'' = (S'', \rightarrow'', s''_0)$ with $TS \sim_{\text{iso}} TS'$ and $TS' \sim_{\text{iso}} TS''$. This must be justified by bijections $f : \text{Reach}(TS) \rightarrow \text{Reach}(TS')$ and $f' : \text{Reach}(TS') \rightarrow \text{Reach}(TS'')$. It is easy to verify that $f \circ f' : \text{Reach}(TS) \rightarrow \text{Reach}(TS'')$ is a bijection justifying $TS \sim_{\text{iso}} TS''$. ■

2.3.2 Isomorphism in Practice

To determine whether isomorphism is a good notion of equality in practice, there are two questions we need to answer:

Figure 2.3: LTS for the CCS terms $a!.b!.0$ and $a!.(b!.0 + b!.(0 + 0))$


1. Are we confident that whenever $P \sim_{\text{iso}} Q$, the processes P and Q are truly indistinguishable in practice?
2. Are we confident that whenever two processes P and Q are indistinguishable in practice, $P \sim_{\text{iso}} Q$ holds?

The former question is easy to answer. Isomorphism only allows processes to deviate in state identities and in unreachable parts of the process, preserving its transition structure. Since we only consider actions to be observable, and unreachable fragments cannot influence the behavior of a process, we are certain that isomorphic processes are equal in any way that is relevant to us.

Unfortunately, the latter question hints at a problem. To investigate this, consider the LTS shown in Figure 2.3.

These processes clearly cannot be isomorphic because the number of reachable states is different.

In practice, however, no difference between these processes is observable. After performing action $b!$, in the right process, we can end up in different states – but these states cannot be distinguished in any way as their behaviour is identical (or, more specifically, empty) and we do not consider state identities to be observable.

This shows us that isomorphism is too strict for our use.

2.3.3 An Unexpected Problem

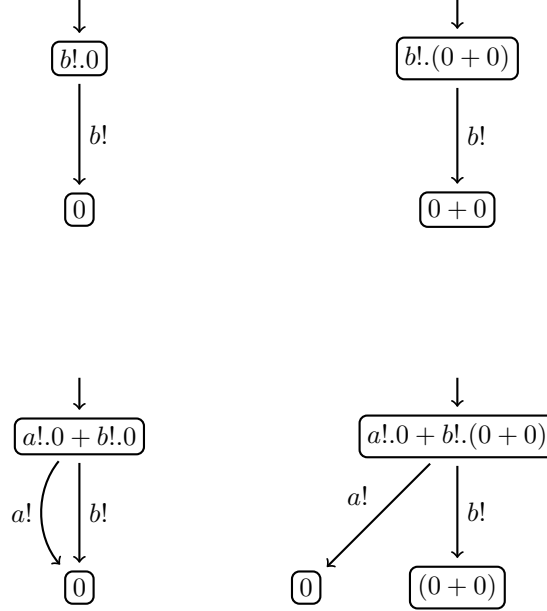
Unfortunately, isomorphism also has a more formal problem. While it is an equivalence relation, there is another formal requirement for an equivalence to be a good notion of equality.

Consider the processes $b!.0$ and $b!.(0 + 0)$, shown in Figure 2.4. They are obviously isomorphic because they only differ in the identity of the states.

Now, let's look at the process $a!.0 + b!.0$, also shown in Figure 2.4. It is a simple process with two reachable states and transitions each. Its CCS term contains $b!.0$ as a sub-term. What happens if we replace it with $b!.(0 + 0)$? It would be reasonable to expect that nothing of relevance changes, as we just replaced a sub-term with one we consider equal.

However, if we draw the process $a!.0 + b!.(0 + 0)$, as shown in Figure 2.4, we see that it now contains an additional state. This means that after the replacement, our process is not isomorphic to the original process!

Figure 2.4: Isomorphism is not a congruence relation



This is a situation we are not used to in mathematics. Not only is $1 + 1 = 2$, but it's also correct that if you find the subterm $1 + 1$ anywhere in a term, you can safely replace it with 2 without changing the value of the term.

If we use isomorphism as our notion of equality for processes, the same does not hold for CCS terms: Even if two CCS terms are equal, they cannot be safely replaced within larger CCS terms without risking to change the process!

To formally grasp the process of replacing sub-terms, we use *contexts*. A context is a CCS term with one hole. For example, $C[\cdot] = a!.0 + [\cdot]$ is the context we used in the example above, with the instantiations $C[b!.0] = a!.0 + b!.0$ and $C[b!.(0 + 0)] = a!.0 + b!.(0 + 0)$.

We call a relation a *congruence relation* if it does allow replacement of sub-terms:

Definition 13 (Congruence Relation). An equivalence relation $\approx$ on CCS is called a *congruence relation* for CCS if for all CCS terms P and Q and all CCS contexts $C[\cdot]$:

$$P \approx Q \implies C[P] \approx C[Q].$$

As we have seen, isomorphism is not a congruence relation.

2.3.4 Hierarchy

Of course, isomorphism is more strict than universal equality and less strict than identity, as shown in Figure 2.5.

This graph will become more interesting as we add more equalities to it.

2.3.5 Summary

Figure 2.5: The inclusion hierarchy for $Id(CCS)$, $Univ(CCS)$ and $\sim_{iso}$

$$Id \subsetneq \sim_{iso} \subsetneq Univ$$

🔖 Properties of Isomorphism

book.pseuCo.com

Which of the following statements about $\sim_{iso}$ are true?

- (a) Isomorphism is an equivalence relation.
- (b) Isomorphism is a congruence relation.
- (c) $\sim_{iso} \subsetneq Id(CCS)$
- (d) $\sim_{iso} \subsetneq Univ(CCS)$
- (e) If two processes are isomorphic, they have the same observable behavior.

Solve this quiz and check your solutions online:

<https://book.pseuco.com/#/quiz/isomorphism-properties>

2.4 Trace Equivalence

So far, we have seen that we need a notion of equality that is a congruence relation and is more lenient than isomorphism.

Given that we want to focus on the behavior of the processes, an obvious candidate for equality is to call processes equal if the set of behaviors they can show is equal.

Since we only consider actions to be observable, a reasonable representation for the behaviour of a process is a *trace*, a sequence of actions a process performs. A trace is a subset of Act^* ¹ that is justified by transitions starting in the initial state of the underlying transition system:

Definition 14 (Traces). For an LTS $TS := (S, \rightarrow, s_0)$, its set of traces is defined as

$$Traces(TS) = \left\{ \alpha_1 \alpha_2 \dots \alpha_n \in Act^* \mid n \in \mathbb{N} \wedge \exists s_1, \dots, s_n \in S : s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n \right\}.$$

Now, we can consider processes to be equal if they have the same traces:

Definition 15 (Trace Equivalence ($\sim_{tr}$)). We call two LTS $TS = (S, \rightarrow, s_0)$ and $TS' = (S', \rightarrow', s'_0)$ *trace equivalent* ($TS \sim_{tr} TS'$) if $Traces(TS) = Traces(TS')$.

This relation clearly is an equivalence relation. It is also a congruence for CCS, which can be proven by structural induction.

However, unfortunately, processes having identical traces is not enough for them to be indistinguishable in practice, as demonstrated by the following example.

🔖 Interactive Exercise

book.pseuCo.com

In this example, we'll consider two vending machines that are trace equivalent.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/trace-equivalent-vending-machines>

¹The asterisk is the Kleene star that turns sets into sequences of their elements. For example, $\{a, b\}$ contains ε (the empty word), a , b , aa , ab , ba , bb , aab , and so on.

The take-away from this example is that while we don't want the identities of states to be considered when comparing processes, states are not irrelevant either: If states differ in what behaviors they enable, this difference can be observed externally, which can cause processes with the same set of traces to be distinguishable.

2.4.1 Termination Sensitivity

The problem we've just seen is enough to convince us that trace equality is not the equality we are looking for. However, just having an intuitive understanding of the problem will not help us later on to show that other notion of equality have solved the problem. Therefore, it is important that we try to get a formal grasp on the problem we have discovered.

We can describe the vending machines we have seen before with these CCS terms (shortening the names for readability):

$$\begin{aligned} HH &:= \text{coin?}. \text{Paid}HH \\ \text{Paid}HH &:= sC?.c!.HH + sW?.w!.HH + sL?.l!.HH \\ FF &:= \text{coin?}.sC?.c!.FF + \text{coin?}.sW?.w!.FF + \text{coin?}.sL?.l!.FF \end{aligned}$$

The core observation made before was that while *Felix's Refreshment Store* can serve every possible type of drink, it does not offer everything all the time. This becomes relevant when an opinionated customer wants a specific drink, such as delicious lemonade:

$$\text{Customer} := \text{coin!}.sL!.l?.\text{smile!}.0$$

Now, let's consider the context $C[\cdot] := ([\cdot] \mid \text{Customer}) \setminus H$ with $H := \text{Com} \setminus \{\text{smile!}, \text{smile?}\}$ for HH and FF , i.e. the customer synchronizing with one of the machines. The situation is shown in Figure 2.6.

Note that this does *not* contradict trace equivalence being a congruence relation: Our two vending machines have the same traces, and within the context with a customer, the LTS look quite different, but the set of traces is actually identical. Most notably, both contain the trace τ – traces do not care that one of the two LTS can be terminated after that trace.

We can try to formally capture this difference by considering the set of traces that end in a terminal state:

Definition 16 (Terminating Traces). For an LTS $TS = (S, \rightarrow, s_0)$, its set of terminating traces is defined as

$$TTraces(TS) := \left\{ \alpha_1 \alpha_2 \dots \alpha_n \in Act^* \mid n \in \mathbb{N} \wedge \exists s_1, \dots, s_n \in S : s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n \wedge s_n \nrightarrow \right\}.$$

This does show the difference:

$$\begin{aligned} TTraces((HH \mid \text{Customer}) \setminus H) &= \{\tau.\tau.\tau.\text{smile!}\} \\ TTraces((FF \mid \text{Customer}) \setminus H) &= \{\tau.\tau.\tau.\text{smile!}, \tau\} \end{aligned}$$

This shows us that trace equivalence does not ensure that the set of terminating traces is identical. We say that it is not *termination sensitive*:

Definition 17 (Termination Sensitivity). Let $\mathcal{R} \in S^2$ be a relation. We say that $\mathcal{R}$ is *termination sensitive* if $\forall s, t \in S : s \mathcal{R} t \implies TTraces(s) = TTraces(t)$.

2.4.2 Hierarchy

Trace equivalence takes a place between isomorphism and universal equality, as shown in Figure 2.7.

Figure 2.6: Two different vending machines with a customer preferring lemonade.

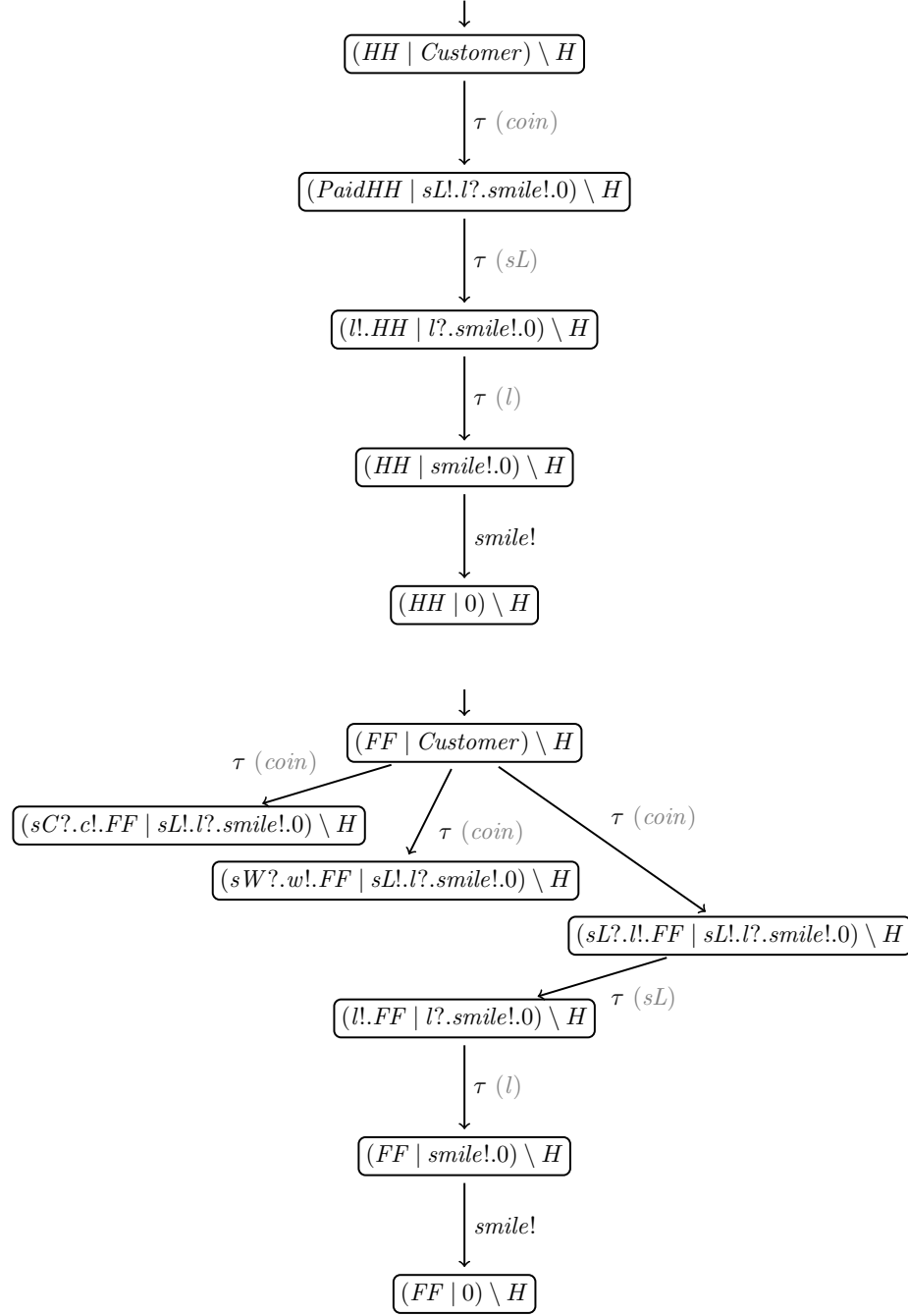
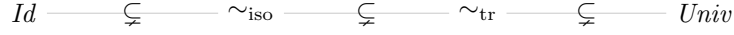


Figure 2.7: The inclusion hierarchy for $Id(CCS)$, $Univ(CCS)$, $\sim_{iso}$ and $\sim_{tr}$ 

2.5 Bisimilarity

As we have seen before, our previous definitions of *equality* do not fully describe what it means for two systems to be interchangeable. In this section, we'll design another concept of equality that fulfills the lessons we have learned before:

- The equality must abstract away the identity of states, as we don't consider them to be part of the behavior of a process.
- But the equality must internally consider state identities, as purely looking at traces does not respect the interaction behavior of the processes.
- The relation between states must be more flexible than requiring a bijection, as isomorphism was clearly too strict when comparing processes.

Since the equality has to consider states, for simplicity, we will now focus on comparing states within a single LTS. Different LTS can be compared by combining them to a single, disconnected LTS, and comparing the former initial states.

2.5.1 First Attempt

Conceptually, we want to consider two states to be *equal* when they can perform the same actions and end up in *equal* states afterwards. We can write that down rather easily:

Definition 18 (Bisimilarity (first attempt)). Let $LTS = (S, \rightarrow, s_0)$ be an LTS. We call two states s and t *bisimilar*, denoted $s \sim t$, if for all $\alpha \in Act$:

- whenever $s \xrightarrow{\alpha} s'$ for a $s' \in S$, then there also is an $t' \in S$ such that $t \xrightarrow{\alpha} t'$ and $s' \sim t'$, and
- whenever $t \xrightarrow{\alpha} t'$ for a $t' \in S$, then there also is an $s' \in S$ such that $s \xrightarrow{\alpha} s'$ and $s' \sim t'$.

While this is quite close to our final definition, it does have a fatal issue.

🔗 Problems with naïve Bisimilarity

book.pseuCo.com

What is the main problem with the naïve definition of Bisimilarity?

- No states can fulfill this definition.
- This is not actually a definition, multiple relations $\sim$ can fulfill the requirements above. The smallest interpretation is that *no* states are bisimilar.
- This is not actually a definition, multiple relations $\sim$ can fulfill the requirements above. The largest interpretation is that *all* states are bisimilar.
- Every pair of states fulfills this definition.

Solve this quiz and check your solutions online:

<https://book.pseuco.com/#/quiz/bisimilarity-first-attempt-problem>

Despite this not being a proper definition, it is close: The *largest* relation satisfying the requirements is what we actually want. However, we will not fix this by simply adding the word *largest* to the definition. The reason for this is that while the result would be correct, such a definition not only makes it hard to prove states to be bisimilar, but it isn't even clear

whether such a definition is valid (i.e. clearly describes exactly one relation). Instead, we will follow a more constructive approach.

2.5.2 Second Attempt

Instead of directly defining that two states are considered equal if they can perform the same actions and end up in equal states, we will now consider two states as equal if *there is a way of relating states* (including the pair in question) such that each pair of states in the relation behaves as discussed previously: both states can perform the same actions and end up in states that are, in turn, related.

We will call a relation that relates equal states a *bisimulation*.

Definition 19 (Bisimulation). A binary relation $\mathcal{R}$ over the states S of an LTS $TS = (S, \rightarrow, s_0)$ is called a *bisimulation* for TS if $\forall (s, t) \in \mathcal{R}$ and $\forall \alpha \in Act$:

- for all $s' \in S$, if $s \xrightarrow{\alpha} s'$ then there is a $t' \in S$ such that $t \xrightarrow{\alpha} t'$ and $(s', t') \in \mathcal{R}$, and
- for all $t' \in S$, if $t \xrightarrow{\alpha} t'$ then there is a $s' \in S$ such that $s \xrightarrow{\alpha} s'$ and $(s', t') \in \mathcal{R}$.

Definition 20 (Bisimilarity ($\sim$)). We call two states s and t of an LTS $TS = (S, \rightarrow, s_0)$ *bisimilar* if there is a bisimulation $\mathcal{R}$ with $(s, t) \in \mathcal{R}$.

The relation $\sim := \{(s, t) \mid s \text{ and } t \text{ are bisimilar}\}$ is called *bisimilarity*.

We call two LTS $TS_1 = (S_1, \rightarrow_1, s_{01})$ and $TS_2 = (S_2, \rightarrow_2, s_{02})$ bisimilar if the states s_{01} and s_{02} are bisimilar in the LTS $(S_1 \sqcup S_2, \rightarrow_1 \sqcup \rightarrow_2, s_{01})$, where $\sqcup$ denotes the disjoint union. (If the union is not disjoint, states have to be consistently renamed.)

Because we now typically only consider LTS that are the semantics of a CCS process and a fixed binding Γ , we do not mention the LTS and simply talk about two states or two CCS processes being bisimilar.

2.5.3 Proving Bisimilarity

To show that two states (or two LTS) are bisimilar, we have to...

1. find a relation $\mathcal{R}$ that captures our understanding of which states are equal,
2. check that the two states (or the initial states) are in fact in $\mathcal{R}$, and
3. show that the relation is a bisimulation: For every pair $(s, t) \in \mathcal{R}$, ...
 - (a) check that for every possible move $s \xrightarrow{\alpha} s'$, there is an appropriate move in the other state (and the resulting pair is in $\mathcal{R}$),
 - (b) and vice versa.

Let's look at an example.

Interactive Exercise

book.pseuCo.com

In this exercise, we'll show that two simple LTS are bisimilar by constructing a bisimulation and showing its correctness step-by-step.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-bisimilarity-simple>

In the previous example, we could build the Bisimulation “greedily”, without looking ahead. This is not possible in general, and choosing what pairs to add to the relation may require careful inspection of the future behaviour of the states.

Let's try another example that is slightly more involved:

 Interactive Exercise

book.pseuCo.com

In this exercise, we'll again show that two simple LTS are bisimilar, but on a slightly harder example.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-bisimilarity-harder>

2.5.4 Disproving Bisimilarity

Not all states are created equal, so we also need to learn how to prove that states are *not* bisimilar.

Let's first look at an example for how states can be different despite having the same traces.

 Interactive Exercise

book.pseuCo.com

Here, we'll try to prove two states to be bisimilar despite them not being bisimilar to see what happens.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-bisimilarity-impossible>

As seen in the previous exercise, attempting to show that states are bisimilar when they are not leads to a conflict in the proof, where no relation $\mathcal{R}$ that contains the pair of states in question can satisfy all requirements of the definition of bisimulation.

In principle, proving states to be not bisimilar can be done directly along the definition, by showing that no bisimulation can contain the pair of states. However, this is often somewhat difficult in practice.

To disprove bisimilarity, it is typically easier to think of the proof as a game. The rules of the game are derived from the definition of bisimilarity, and are (apart from an omission discussed later) as follows:

- The game is played by two players, an attacker and defender.
- Both players start in the two states in question: The attacker picks one of them, and the defender is placed in the other.
- In each turn, the attacker picks a transition from the state it is in and moves along it. The defender is then required to also move along a transition with the same action.
- If the defender cannot answer an attack, the attacker wins.

Before we consider what exactly this game means in relation to bisimilarity, let's first try to win a game as the attacker:

 Interactive Exercise

book.pseuCo.com

Here, we play a first round of the bisimilarity game.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-non-branching>

If you try playing this game on two *bisimilar* states, you'll notice that it is easy for the attacker to win if the defender deliberately or unintentionally makes mistakes. This is meaningless, just

CHAPTER 2. EQUALITY

like a contradiction in a bisimilarity proof could simply be caused by a bad choice of $\mathcal{R}$. We have to consider a perfect defender!

If there is an attack strategy that *guarantees* the attacker victory, the initial states of the game are not bisimilar.

Therefore, for such a strategy to be considered a proof of non-bisimilarity, it has to consider every possible move of the defender.

Let's try the previous example again, this time finding a path to victory for each strategy the defender could deploy:

👉 Interactive Exercise

book.pseuCo.com

In this exercise, we play the bisimilarity game again, this time building a complete victory strategy for the attacker, to show two simple LTS are not bisimilar.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-simple>

However, the ruleset above is not complete yet. There is a class of states that can only be proven different under bisimilarity by the inclusion of the final rule:

- Before every turn, the attacker is allowed to swap the attacker and defender position.

Consider the following example. Note that it is not possible to find a strategy for the attacker that does not contain a swap.

👉 Interactive Exercise

book.pseuCo.com

In this exercise, we'll play the bisimilarity game in a case where switching sides is required.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-harder>

It is helpful to consider how the rules of the bisimilarity game correspond to the definition of bisimulation.

👉 Rules of the Bisimilarity Game

book.pseuCo.com

What part of the formal definition corresponds to the attacker being allowed to pick the transition to attack with?

- The symmetry of both conditions within the definition of bisimulation.
- The requirement that the pair of states in question has to be in $\mathcal{R}$.
- The requirement that in the definition of bisimulation, $s \xrightarrow{\alpha} s'$ and $t \xrightarrow{\alpha} t'$ share the same action α .
- The $\forall$ -quantifications over α and s' or t' .

In the game, what corresponds to the bisimulation $\mathcal{R}$?

- The pairs of possible positions the players can be in before the defender moves.
- The pairs of possible positions the players can be in before the attacker moves.

- (c) The strategy of the attacker.
- (d) The strategy of the defender.

What part of the formal definition corresponds to the rule allowing the attacker to swap places with the defender?

- (a) The symmetry of both conditions within the definition of bisimulation.
- (b) The requirement that the pair of states in question has to be in $\mathcal{R}$.
- (c) The requirement that in the definition of bisimulation, $s \xrightarrow{\alpha} s'$ and $t \xrightarrow{\alpha} t'$ share the same action α .
- (d) The $\forall$ -quantifications over α and s' or t' .

Solve this quiz and check your solutions online:
<https://book.pseuco.com/#/quiz/bisimilarity-game-rules-explained>

2.5.5 Properties of Bisimilarity

Now that we have gained an understanding of how bisimilarity works, it is time to take a look at its properties.

Equivalence Relation

Bisimilarity is an equivalence relation.

Proof. Let $TS = (S, \rightarrow, s_0)$ be an LTS. We show all three properties separately:

- $\sim$ is reflexive: Let $s \in S$. $\mathcal{R} := \{(s', s') \mid s' \in S\}$ obviously is a bisimulation with $(s, s) \in \mathcal{R}$.
- $\sim$ is symmetric: Let $s_1, s_2 \in S$ with $s_1 \sim s_2$. This must be justified by a bisimulation $\mathcal{R}$ with $(s_1, s_2) \in \mathcal{R}$. But then, $\mathcal{R}^{-1}$ is a bisimulation (as the requirements of bisimulation are symmetric) with $(s_2, s_1) \in \mathcal{R}^{-1}$, so $s_2 \sim s_1$.
- $\sim$ is transitive: Let $s_1, s_2, s_3 \in S$ with $s_1 \sim s_2$ and $s_2 \sim s_3$. This must be justified by relations $\mathcal{R}_1$ and $\mathcal{R}_2$ with $(s_1, s_2) \in \mathcal{R}_1$ and $(s_2, s_3) \in \mathcal{R}_2$. The relation $\mathcal{R} := \mathcal{R}_1 \circ \mathcal{R}_2$ obviously contains the pair (s_1, s_3) . To show $s_1 \sim s_3$, it remains to show that $\mathcal{R}$ is a bisimulation.

To show this, consider any pair $(A, C) \in \mathcal{R}$. This pair must be justified by pairs $(A, B) \in \mathcal{R}_1$ and $(B, C) \in \mathcal{R}_2$. Now, consider both directions separately.

- Let $A \xrightarrow{\alpha} A'$. Because $\mathcal{R}_1$ is a bisimulation, there must be a B' with $B \xrightarrow{\alpha} B'$ and $(A', B') \in \mathcal{R}_1$. Because $\mathcal{R}_2$ is a bisimulation, there must be a C' with $C \xrightarrow{\alpha} C'$ and $(B', C') \in \mathcal{R}_2$. But then, we also have $(A', C') \in \mathcal{R}$.
- The other direction follows analogously (swap A with C except within tuples).

■

Congruence

Next, we need to check whether bisimilarity is a congruence for CCS.

Proving this requires a lengthy case distinction about every possible CCS term. We'll only show parts of the proof here.

Proof. We need to show that for all CCS terms P and Q and for all CCS contexts $C[\cdot]$ with $P \sim Q$, we have $C[P] \sim C[Q]$. So let P , Q and $C[\cdot]$ be fixed as specified.

Proof by structural induction over $C[\cdot]$.

As $C[\cdot]$ is a CCS term with a hole, it is either the trivial context consisting just of a hole (e.g. $C[\cdot] = [\cdot]$) or consists of an operator where one operand is a context (has a hole somewhere) and the other (if the operator has two) is a regular CCS term. For the trivial context, the statement is trivially true. For all other contexts, given that the terms we insert into the holes are bisimilar, the induction hypothesis tells us that the whole operand with the hole is bisimilar. What remains to show is that for every CCS operator, for each side, if we replace the operand with a bisimilar one, the whole term stays bisimilar:

- For any $\alpha \in Act$: $\alpha.P \sim \alpha.Q$.
- For any $R \in CCS$: $P + R \sim Q + R$ and $R + P \sim R + Q$.
- For any $R \in CCS$: $P \mid R \sim Q \mid R$ and $R \mid P \sim R \mid Q$.
- For any $H \subseteq Com$: $P \setminus H \sim Q \setminus H$.

The proofs for these statements are different, but structurally similar. Here, we will only consider one half of one of these statements and show that for any $R \in CCS$: $P + R \sim Q + R$. So, let $R \in CCS$ be fixed. We know that $P \sim Q$, so there must be a bisimulation $\mathcal{R}$ with $(P, Q) \in \mathcal{R}$. We claim that $\mathcal{R}' := \mathcal{R} \cup Id(CCS) \cup \{(P + R, Q + R)\}$ is a bisimulation that contains $(P + R, Q + R)$. The latter is trivial, so we only need to show it is a bisimulation. So, assume we have a pair from $\mathcal{R}'$. Case distinction over what subset the pair is from:

- If the pair is from $\mathcal{R}$, we are trivially done, as $\mathcal{R}$ is already known to be a bisimulation.
- If the pair is from $Id(CCS)$, the properties are also trivial.
- Consider the pair $(P + R, Q + R)$. We consider both sides separately.
 - Consider an attack $P + R \xrightarrow{\alpha} X$. Case distinction over the semantic rule that justifies this transition.
 - * $P + R \xrightarrow{\alpha} P'$ with *choice_l*. Then, we also have $P \xrightarrow{\alpha} P'$. Since (P, Q) is in the bisimulation $\mathcal{R}$, there is a Q' such that $Q \xrightarrow{\alpha} Q'$ with $(P', Q') \in \mathcal{R}$. But then, with *choice_l*, we also know $Q + R \xrightarrow{\alpha} Q'$. Moreover, we know $(P', Q') \in \mathcal{R}'$ by its definition.
 - * $P + R \xrightarrow{\alpha} R'$ with *choice_r*. Then, we also have $R \xrightarrow{\alpha} R'$. But then, with *choice_r*, we also know $Q + R \xrightarrow{\alpha} R'$, and $(R', R') \in Id(CCS) \subseteq \mathcal{R}$.
 - The other direction is analogous (swap P and Q exception within tuples).

The principle shown above can be used in most other parts of the proof: Assume a bisimulation for the bisimilarity we know, and use it to construct a new one by adding or modifying pairs. ■

2.5.6 Termination Sensitivity

Lemma 4. Bisimilarity $\sim$ is termination sensitive.

Conceptually, proving this is easy: Let s and t be bisimilar. Then, if $\alpha_1 \cdots \alpha_n$ is in $TTraces(s)$, the situation is as sketched in Figure 2.8: The trace is justified by a sequence of states s_i ending in a terminal state s_n . Bisimilarity asserts that t also has such a trace, and bisimilarity is propagated throughout the trace. It remains to show that t_n is terminal. That can be done by contradiction: If it had an outgoing transition, by bisimilarity, s_n would also have to have a transition, which contradicts our previous knowledge.

Writing down this argument formally requires solving a few technicalities and is left as a recommended exercise.

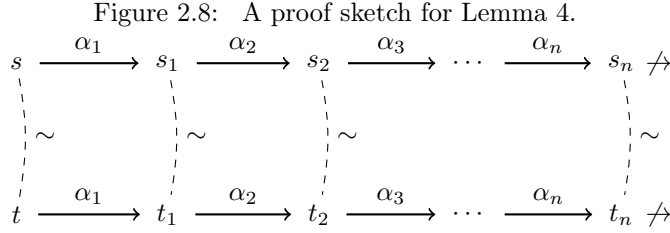


Figure 2.9: The inclusion hierarchy for $Id(CCS)$, $Univ(CCS)$, $\sim_{iso}$, $\sim_{tr}$ and $\sim$.

$$Id \subsetneq \sim_{iso} \subsetneq \sim \subsetneq \sim_{tr} \subsetneq Univ$$

2.5.7 Hierarchy

Bisimilarity takes a place between isomorphism and trace equivalence, as shown in Figure 2.9.

2.5.8 Minimization

As we have seen before, bisimilarity allows LTS of different sizes to be considered equal. Of course, in many use cases, smaller LTS are advantageous. This raises an interesting type of question: For any given LTS, what is the smallest representation of it, according to bisimilarity?

This question is not always easy to answer. Moreover, it isn't even clear whether it has a unique answer: What does *smallest* mean? Minimal number of transitions? Of states? Or both?

Conveniently, it turns out there is a unique answer, optimizing for transition count and state count at once:

Lemma 5. Let $TS = (S, \rightarrow, s_0)$ be an arbitrary LTS. Then, there is a *minimal representative up to bisimilarity* $TS' = (S', \rightarrow', s'_0)$ such that

- $TS \sim TS'$,
- there is no $TS'' = (S'', \rightarrow'', s''_0)$ such that $TS \sim TS''$ and $|S''| < |S'|$, and
- there is no $TS'' = (S'', \rightarrow'', s''_0)$ such that $TS \sim TS''$ and $|\rightarrow''| < |\rightarrow'|$.

While this lemma shows us that there always is a unique minimal representative, it does not help us to actually identify it. However, there is a constructive approach to finding the minimal representative:

1. Start with an arbitrary LTS $TS = (S, \rightarrow, s_0)$.
2. Identify the equivalence classes $S/\sim$ as follows:
 - (a) Initialize the set of equivalence classes $S/\sim$ with only a single class: $S/\sim := \{S\}$
 - (b) While there is an action α and a “splitter” equivalence class $s \in S/\sim$ such that the set of predecessors $Pre(s, \alpha)$ is not a union of equivalence classes, i.e. there is at least one class $t \in S/\sim$ such that there are states $t_1, t_2 \in t$ such that $t_1 \in Pre(s, \alpha)$ but $t_2 \notin Pre(s, \alpha)$, split the equivalence classes as follows:
 - For each equivalence class $r \in S/\sim$ (including s), remove r from $S/\sim$, and add the new classes $r \cap Pre(s, \alpha)$ and $r \setminus Pre(s, \alpha)$ to $S/\sim$ if they are non-empty.
3. Construct the minimal representative $(S', \rightarrow', s'_0)$ as follows:
 - (a) Each equivalence class becomes a state:

$$S' := S/\sim$$

(b) New states are connected by all transitions their member states had:

$$\rightarrow' := \left\{ (s', \alpha, t') \mid \exists s \in s' : \exists t \in t' : s \xrightarrow{\alpha} t \right\}$$

(c) The new initial state is the equivalence class of the old initial state:

$$s'_0 := [s_0]$$

This algorithm is best understood by applying it to an example.

 Interactive Exercise

book.pseuco.com

Here, we run the minimization algorithm step-by-step to minimize our first LTS.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/strong-bisimilarity-minimization>

Here's a slightly more involved example:

 Interactive Exercise

book.pseuco.com

Here, we apply minimization to a larger LTS.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/strong-bisimilarity-minimization-harder>

2.5.9 The perfect equality?

So far, bisimilarity is holding up well. We've learned that it...

- is an equivalence relation ✓
- is a congruence relation, making it robust against component replacement up to equality ✓
- preserves traces ✓
- preserves the behavior of processes ✓
- is termination sensitive ✓
- abstracts away states where possible ✓
- allows finding a minimal representation ✓

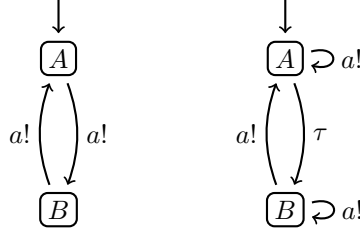
Unfortunately, there is one case we still have not considered properly. For example, consider Figure 2.10. It shows two different LTS. They clearly are not trace equivalent (the right one has traces with a τ , the left one does not) and, consequently, also not bisimilar.

But are they really *different*? The left LTS describes a system that continuously performs $a!$ actions and nothing else. The right LTS differs only in one regard: In some cases it can perform a τ action. Nonetheless, it can always perform $a!$, and apart from τ nothing else. Given that we originally introduced τ as an *internal* action that is not observable to the outside, it seems inconsistent to distinguish LTS because of it.

2.6 Weak Bisimilarity

Our next goal is to find a variant of bisimilarity that incorporates our understanding that τ transitions can be silently taken without being observable.

Figure 2.10: These LTS are not bisimilar, but are they equal?



Before we do this, we have to answer an important question: Are τ transitions relevant at all? It's clear they cannot always be simply omitted (imagine an LTS that starts with a τ transition before anything else happens – deleting the τ transition would remove all its behavior). But it is reasonable to think that they can be “patched out” by merging the source and target states, since they can be switched between silently anyways. Does that work every time without changing the observable behavior?

To answer this question, consider the following example:

🔗 Interactive Exercise

book.pseuCo.com

In this example, we'll consider two vending machines where a τ plays a crucial role.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/relevant-tau-vending-machines>

Now that we know that τ transitions are sometimes relevant, we can look into how to incorporate that into bisimilarity. Conceptually, we will be doing this by replacing our notion of taking a transition, the elements $s \xrightarrow{\alpha} s'$ of the transition relation, with a variant that allows silently taking τ transitions before or after the “main” transition, and conversely allows omitting τ transitions.

Definition 21 (Trace Reachability ($\sim$)). Let $\sigma = \alpha_1\alpha_2 \dots \alpha_n \in Act^*$ be a trace. Then, by $s \xrightarrow{\sigma} s'$, we denote that either $\sigma = \epsilon$ and $s = s'$ or there are states $s_0, \dots, s_n$ such that $s_0 = s$, $s_n = s'$ and $\forall i \in \{1, \dots, n\} : s_{i-1} \xrightarrow{\alpha_i} s_i$.

Definition 22 (Weak Transitions ($\Rightarrow$)). Let $TS = (S, \rightarrow, s_0)$ be an LTS, and let $s, s' \in S$ be states.

We say that s has a weak τ -transition to s' , written $s \xRightarrow{\tau} s'$, if and only if $\exists n \in \mathbb{N} : s \xrightarrow{\tau^n} s'$.

For $a \in Com$, we say that s has a weak a -transition to s' , written $s \xRightarrow{a} s'$, if and only if $\exists n, m \in \mathbb{N} : s \xrightarrow{\tau^n a \tau^m} s'$.

Note that by this definition, $s \xrightarrow{\alpha} s'$ implies $s \xRightarrow{\alpha} s'$. Also, for every state s , we have $s \xRightarrow{\tau} s$.

Using this notion of transitions, we can re-define bisimulation and bisimilarity to get a *weak* versions:

Definition 23 (Weak Bisimulation). A binary relation $\mathcal{R}$ over the states S of an LTS $TS = (S, \rightarrow, s_0)$ is called a *weak bisimulation* for TS if $\forall (s, t) \in \mathcal{R}$ and $\forall \alpha \in Act$:

- for all $s' \in S$, if $s \xRightarrow{\alpha} s'$ then there is a $t' \in S$ such that $t \xRightarrow{\alpha} t'$ and $(s', t') \in \mathcal{R}$, and

- for all $t' \in S$, if $t \xrightarrow{\alpha} t'$ then there is a $s' \in S$ such that $s \xrightarrow{\alpha} s'$ and $(s', t') \in \mathcal{R}$.

Definition 24 (Weak Bisimilarity ($\approx$)). We call two states s and t of an LTS $TS = (S, \rightarrow, s_0)$ *weakly bisimilar* if there is a weak bisimulation $\mathcal{R}$ with $(s, t) \in \mathcal{R}$.

The relation $\approx := \{(s, t) \mid s \text{ and } t \text{ are weakly bisimilar}\}$ is called *weak bisimilarity*.

Weak bisimilarity of LTS (as opposed to states within an LTS) is defined analogously to (strong) bisimilarity.

Before we attempt to work with this definition, we need to add an important lemma. The definition of weak bisimilarity has introduced weak transitions on both sides, the universally quantified side of transitions we need to consider (the attacker side) and the existentially quantified side of transitions we may use to simulate them (the defender side). The latter change is crucial: It is what allows us to silently take τ transitions while simulating. Also making the former replacement is consequential, but it dramatically increases the work needed to prove a relation to be a bisimulation, as we now need to find a defense for each of the (possibly numerous) weak transitions.

Thankfully, this is not required:

Lemma 6. Let $\mathcal{R}$ be a binary relation over the states S of an LTS $TS = (S, \rightarrow, s_0)$ and let TS be a LTS such that $\forall (s, t) \in \mathcal{R}$ and $\forall \alpha \in Act$:

- for all $s' \in S$, if $s \xrightarrow{\alpha} s'$ then there is a $t' \in S$ such that $t \xrightarrow{\alpha} t'$ and $(s', t') \in \mathcal{R}$, and
- for all $t' \in S$, if $t \xrightarrow{\alpha} t'$ then there is a $s' \in S$ such that $s \xrightarrow{\alpha} s'$ and $(s', t') \in \mathcal{R}$.

Then $\mathcal{R}$ is a weak bisimulation.

Proving states to be weakly bisimilar works just as before, except that we now can use weak transitions when choosing how to simulate a transition from the other state:



Interactive Exercise

book.pseuCo.com

In this exercise, we prove weak bisimilarity for the first time.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-bisimilarity-weak>

Disproving weak bisimilarity also works similarly: The defender can now use weak transitions as well, resulting in more cases to consider in your proof.



Interactive Exercise

book.pseuCo.com

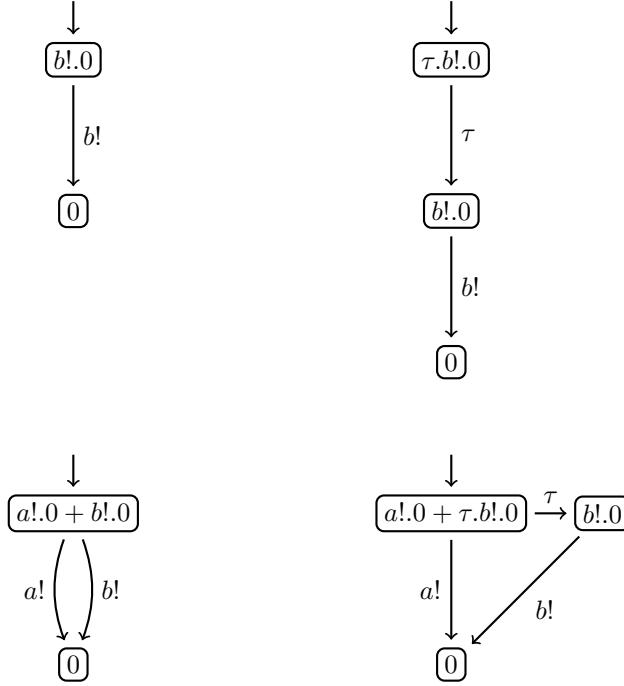
In this exercise, we disprove weak bisimilarity for the first time.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-weak-simple>

In this example, we only allowed the attacker to use regular transitions. This is always sufficient, as we have shown with Lemma 6, but if our goal is to disprove bisimilarity, allowing the attacker to use weak transitions can significantly shorten the proof. As a rule of thumb, when using the bisimilarity game, prefer attacker moves that leave the defender fewer options to chose from.

Figure 2.11: LTS showing the context $C[\cdot] = a!.0 + [\cdot]$ and the CCS terms $b!.0$ and $\tau.b!.0$.



In the next exercise, we will see an example of this: Using a weak transition, the attacker can win in a single move. Using regular transitions, the proof needs to distinguish many possible intermediate moves by the defender.

🔗 Interactive Exercise

book.pseuCo.com

In this exercise, we play the weak bisimilarity game in a case where the attacker can win in a single move, but only by using a weak transition.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-weak-attacker-steps>

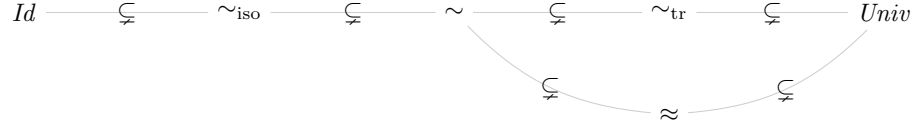
2.6.1 A Problem Returns

As we have seen before, strong bisimilarity is a congruence relation for CCS. Therefore, it may seem reasonable to assume that weak bisimilarity has the same property.

Unfortunately, this is not the case. There is one problem preventing this, and it is related to the choice operator. Consider the context $C[\cdot] = a!.0 + [\cdot]$ and the CCS terms $b!.0$ and $\tau.b!.0$.

The situation is shown in Figure 2.11. First, let us consider $b!.0$ and $\tau.b!.0$. They are clearly weakly bisimilar, as a quick proof shows:

Figure 2.12: The inclusion hierarchy for $Id(CCS)$, $Univ(CCS)$, $\sim_{iso}$, $\sim_{tr}$, $\sim$ and $\approx$.



🔗 **Interactive Exercise**

book.pseuco.com

Here, we show that $b!.0$ and $\tau.b!.0$ are weakly bisimilar.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-bisimilarity-weak-is-no-congruence-example>

However, within our context, the τ transitions becomes relevant, as we can see when we play the weak bisimilarity game:

🔗 **Interactive Exercise**

book.pseuco.com

Here, we show that within our context, $b!.0$ and $\tau.b!.0$ have an observable difference.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-weak-is-no-congruence-example>

This means that weak bisimilarity is not a congruence. In the next section, we will investigate how to change the definition of weak bisimilarity to resolve this problem.

2.6.2 Hierarchy

Adding weak bisimilarity to the inclusion hierarchy opens up an alternative path, as shown in Figure 2.9.

The inclusion $\sim \subsetneq \approx$ is easy to show given that $s \xrightarrow{\alpha} t$ implies $s \xRightarrow{\alpha} t$, and strictness follows by comparing the processes $a!.0$ and $a!.\tau.0$.

The inclusion $\approx \subsetneq Univ(CCS)$ is trivial.

Trace equality is incomparable to weak bisimilarity, as one abstracts away some τ transitions (compare $a!.0$ and $a!.\tau.0$) and the other abstracts away branching behavior (compare $a!.(b!.0 + c!.0)$ and $a!.b!.0 + a!.c!.0$).

2.7 Observation Congruence

As we have seen before, weak bisimilarity is not a congruence. The problem we have found is related to τ steps within the Choice operator: While τ steps may sometimes be irrelevant under weak bisimilarity, when they occur on one side of a $+$, they cause that side to be *selected*, removing the option of taking transitions from the other side.

This problem is only relevant for τ transitions starting in the initial state of a process. Therefore, to turn weak bisimilarity into a congruence, we attempt to make it slightly more strict by changing the rules as follows: τ transitions are treated under the rules for weak bisimilarity,

except in the initial states, where we treat them as observable transitions that must be matched with another real τ -transition:

Definition 25 (Observation Congruence ($\cong$)). Let $TS = (S, \rightarrow, s_0)$ be an LTS, and let $s, t \in S$ be states.

We say that s and t are *observation congruent*, written $s \cong t$, if for all $\alpha \in Act$:

- Whenever $s \xrightarrow{\alpha} s'$, then $\exists t' \in S : \exists n, m \in \mathbb{N} : t \xrightarrow{\tau^n \alpha \tau^m} t' \wedge s' \approx t'$, and
- whenever $t \xrightarrow{\alpha} t'$, then $\exists s' \in S : \exists n, m \in \mathbb{N} : s \xrightarrow{\tau^n \alpha \tau^m} s' \wedge s' \approx t'$.

Observation congruence of LTS (as opposed to states within an LTS) is defined analogously to (strong) bisimilarity.

This definitions deserves a closer look:

- It is direct, instead of being based on a relation.
- It is not recursive, but falls back to $\approx$ after one step.
- It is named *observation congruence* – we will investigate later on whether that name is appropriate.

2.7.1 Proving Observation Congruence

In principle, observation congruence can be shown by simply following the definition, considering all transitions from the states in question and then proving the pairs of states that can be reached to be weakly bisimilar as usual. Doing so naïvely leads to multiple weak bisimilarity proofs, which is *no bueno*.

Instead, to prove two states to be observation congruent, we start by showing them to be weakly bisimilar, making sure to simulate initial τ -steps with a real τ -step. Once this proof is completed, we can argue that since the states are bisimilar, and we happened to simulate all initial τ -steps according to the rules of observation congruence, they must also be observation congruent.

Let's apply this to a simple example:

Interactive Exercise

book.pseuCo.com

In this exercise, we'll prove two LTS to be observation congruent by first proving them to be weakly bisimilar.

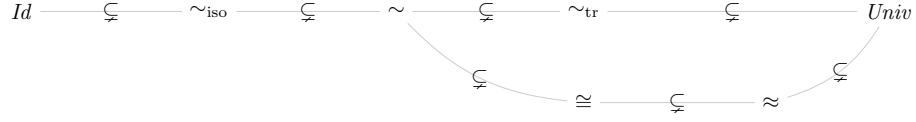
Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-observation-congruence-simple>

2.7.2 Disproving Observation Congruence

For disproving observation congruence, we use a variant of the weak bisimilarity game. It differs only in one rule: If the attacker uses a real τ -transition in their first attack, the defender may not answer by standing still.

Figure 2.13: The inclusion hierarchy for $Id(CCS)$, $Univ(CCS)$, $\sim_{iso}$, $\sim_{tr}$, $\sim$, $\approx$ and $\cong$.



Interactive Exercise

book.pseuCo.com

Here, we'll play a simple instance of the observation congruence game where the attacker has to exploit the new rule.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-observation-congruence-simple>

2.7.3 Hierarchy

Before we analyse our new notion of equality, let's quickly add it to our inclusion hierarchy. It is slightly stricter than weak bisimilarity, as shown in Figure 2.9.

The incomparability of trace equivalence and observation congruence follows from the same examples as we used to show incomparability of trace equivalence and weak bisimilarity.

The inclusion $\sim \subsetneq \cong$ and $\cong \subsetneq \approx$ both can be shown rather easily from their definitions (for inclusion) and the example pair $a!. \tau.0$ versus $a!.0$ and $\tau.a!.0$ versus $a!.0$ (for strictness).

2.7.4 Properties of Observation Congruence

Before we can see whether observation congruence is our ideal notion of equality for now, we must answer two questions:

1. Did we succeed in creating a congruence relation for CCS?
2. We made our relation more strict. If it is a congruence, is it the coarsest one? (Or did we go too far? Are there notions of equality between weak bisimilarity and observation congruence that are a congruence?)

We can answer both questions positively, but the proof is involved. It is based on a trick: First, we define another notion of equality that does fulfill these properties. Then, we show it agrees with observation congruence for all $P, Q \in CCS$, allowing us to transfer our results. Since we are only defining this equality to show these properties, we restrict ourselves to CCS processes (instead of considering arbitrary LTS).

We need to choose this intermediate notion of equality such that it is as easy as possible to show our properties. Therefore, we are looking for a direct way of making weak bisimilarity just strict enough to become a congruence. So, let us consider two processes equal if they are weakly bisimilar *in any choice-operator context*:

Definition 26 (Choice Congruence ($\approx^+$)). Let $P, Q \in CCS$ be CCS terms. We say they are choice congruent, written $P \approx^+ Q$, if for every $R \in CCS$ we have $P + R \approx Q + R$.

First, we establish that choice congruence is a subset of weak bisimilarity:

Lemma 7. $\approx^+ \subsetneq \approx^2$

Proof. Consider a pair $(P, Q) \in \approx^+$. By definition, we know that $\forall R \in CCS : P + R \approx Q + R$, therefore $P + 0 \approx Q + 0$. Because $\forall P \in CCS : P + 0 \sim P$ and strong bisimilarity implies weak bisimilarity, we can follow $P \approx Q$, concluding the proof. ■

The strictness follows from $a!.0 \not\approx^+ \tau.a!.0$. ■

Three proofs lie ahead of us. They are involved and need case distinctions over all CCS operators, so we will only consider a few select cases.

Lemma 8. Choice congruence is a congruence relation for CCS .

Proof. Let $P, Q \in CCS$ be arbitrary CCS terms with $P \approx^+ Q$. As always, we need to make a case distinction over all operators of CCS . We only consider some cases here:

- First, we show that for all $R \in CCS$, we have $P + R \approx^+ Q + R$. This is rather easy, as it is the case choice congruence is defined around.

Let $R \in CCS$ be fixed. Since $P \approx^+ Q$, we know that $\forall S \in CCS : P + S \approx Q + S$. Therefore $\forall T \in CCS : P + (R + T) \approx Q + (R + T)$. Since we know that $+$ is associative for isomorphism, and therefore also for weak bisimilarity, it follows that $\forall T \in CCS : (P + R) + T \approx (Q + R) + T$, so $P + R \approx^+ Q + R$.

- As an example for a more normal case, we consider the prefix operator: We show that $\forall \alpha \in Act : \alpha.P \approx^+ \alpha.Q$.

By definition, we need to show $\forall R \in CCS : \alpha.P + R \approx \alpha.Q + R$. Let $\mathcal{R} := \{(\alpha.P + R, \alpha.Q + R)\} \cup \approx$. Obviously $(\alpha.P + R, \alpha.Q + R) \in \mathcal{R}$, so it only remains to show $\mathcal{R}$ is a weak bisimulation. This is obviously the case for $\approx$, so we only need to consider the pair $(\alpha.P + R, \alpha.Q + R)$. Consider both directions:

- Consider transitions from $\alpha.P + R$. Case distinction by the inference rule justifying the transition.
 - * Consider *choice_l*. Then the transition has the form $\alpha.P + R \xrightarrow{\alpha} P$ (because $\alpha.P \xrightarrow{\alpha} P$ because of *prefix*). By *prefix* and *choice_l*, we also know $\alpha.Q + R \xrightarrow{\alpha} Q$. Since $P \approx^+ Q$ implies $P \approx Q$ by Lemma 7, we know $(P, Q) \in \mathcal{R}$.
 - * Consider *choice_r*. Then the transition has the form $\alpha.P + R \xrightarrow{\beta} R'$ for some $\beta \in Act$, justified by $R \xrightarrow{\beta} R'$. By *choice_r*, we also have $\alpha.Q + R \xrightarrow{\beta} R'$. $(R', R') \in \approx$ holds trivially.
- The other direction is analogous by swapping P and Q except in the context of pairs in $\mathcal{R}$.

We skip the other cases here. ■

Lemma 9. Choice congruence is the coarsest congruence relation for CCS within weak bisimilarity.

Proof. We've already shown that choice congruence is a subset of weak bisimilarity in Lemma 7. It remains to show that any congruence within weak bisimilarity is a subset of choice congruence.

So, let Λ be a congruence relation with $\Lambda \subseteq \approx$. We need to show $\Lambda \subseteq \approx^+$. So, let $(P, Q) \in \Lambda$ be a pair from Λ . As Λ is a congruence, membership in it is preserved under the $+$ operator, so $\forall R \in CCS : (P + R, Q + R) \in \Lambda$. As we know $\Lambda \subseteq \approx$, we can conclude $\forall R \in CCS : (P + R \approx Q + R)$, which by definition means $P \approx^+ Q$. ■

²Remember that we silently treat CCS terms as states (within the LTS created by CCS semantics), allowing us to compare $\approx^+$ and $\approx$ directly.

CHAPTER 2. EQUALITY

Lemma 10. Choice congruence and observation congruence agree for all $P, Q \in CCS$.

Proof. We show $\cong \subseteq \approx^+$ and $\approx^+ \subseteq \cong$ separately:

- Let $P, Q \in CCS$ be arbitrary processes with $P \cong Q$. We need to show $P \approx^+ Q$, which by definition means $\forall R \in CCS : P + R \approx Q + R$. So, let $R \in CCS$ be an arbitrary CCS term.

We set $\mathcal{R} = \{(P + R, Q + R)\} \cup \approx$. Since $(P + R, Q + R) \in \mathcal{R}$, it only remains to show that $\mathcal{R}$ is a weak bisimulation. Since $\approx$ is a weak bisimulation, we only need to check the pair $(P + R, Q + R)$. Consider both directions:

- Assume $P + R \xrightarrow{\alpha} X$. Case distinction on the underlying inference rule.
 - * Assume $P + R \xrightarrow{\alpha} P'$ because of *choice_l* and $P \xrightarrow{\alpha} P'$. As $P \cong Q$ there are $n, m \in \mathbb{N}$ such that $Q \xrightarrow{\tau^n \alpha \tau^m} Q'$ and $P' \approx Q'$. We can conclude with *choice_l* that $Q + R \xrightarrow{\alpha} Q'^3$. Since $P' \approx Q'$ we can also conclude $(P', Q') \in \mathcal{R}$, concluding this case.
 - * Assume $P + R \xrightarrow{\alpha} R'$ because of *choice_r*. Also by *choice_r*, we know $Q + R \xrightarrow{\alpha} R'$, and $(R', R') \in \approx \subseteq \mathcal{R}$.
- The other direction is analogous by swapping P and R except in pairs from $\mathcal{R}$.
- Let $P, Q \in CCS$ be arbitrary processes. Instead of showing $P \approx^+ Q \implies P \cong Q$ directly, we prove its contraposition: $P \not\approx Q \implies P \not\approx^+ Q$. So assume $P \not\approx Q$. This means there is a transition from P or Q that violates the property. We assume it is from P , the other case is analogous (swap P and Q). So, assume $P \xrightarrow{\alpha} P'$ and that there are no $n, m \in \mathbb{N}$ and $Q' \in CCS$ such that $Q \xrightarrow{\tau^n \alpha \tau^m} Q' \wedge P' \approx Q'$. Case distinction on α :
 - If $\alpha \neq \tau$, we know that $Q \not\approx Q'$. But this implies $P \not\approx Q$, and therefore $P + 0 \not\approx Q + 0$, and therefore, by definition, $P \not\approx^+ Q$.
 - If $\alpha = \tau$, i.e. $P \xrightarrow{\tau} P'$, we know that there is no $n \in \mathbb{N}$ such that $Q \xrightarrow{\tau^n} Q' \wedge P' \approx Q'$. Let $\beta \in Act$ be an action that cannot be executed by P or any of its successors.⁴ To show $P \not\approx^+ Q$, we show $P + \beta.0 \not\approx Q + \beta.0$. Proof by contradiction, assume they are weakly bisimilar. By *choice_l*, we know $P + \beta.0 \xrightarrow{\tau} P'$. Because of weak bisimilarity, this implies $Q + \beta.0 \xrightarrow{\tau} Q''$. By our knowledge about which transitions Q cannot have, the only possibility is $Q = Q''$, e.g. the weak transition corresponds to standing still. Then, by weak bisimilarity, $P' \approx Q + \beta.0$. However, $Q + \beta.0$ can execute β , but we know that no successor of P can do so, contradicting weak bisimilarity.

■

2.7.5 Minimization

In Section 2.5.8, we discussed minimizing LTS according to bisimilarity. Our main results were:

- Minimization yields a unique result that is minimal both in the number of states and transitions.
- We identified an algorithm that constructs this minimal representative.

The uniqueness also holds for observation congruence:

³If this is not obvious, consider a case distinction on n : If $n = 0$, we apply *choice_l* for the α -step, otherwise for the τ -step. In both cases, the remaining steps are then identical.

⁴Such an action must exist because we only allow finite bindings and finite CCS terms, but Act is infinite.

Lemma 11. Let $TS = (S, \rightarrow, s_0)$ be an arbitrary LTS. Then, there is a *minimal representative up to observation congruence* $TS' = (S', \rightarrow', s'_0)$ such that

- $TS \cong TS'$,
- there is no $TS'' = (S'', \rightarrow'', s''_0)$ such that $TS \cong TS''$ and $|S''| < |S'|$, and
- there is no $TS'' = (S'', \rightarrow'', s''_0)$ such that $TS \cong TS''$ and $|\rightarrow''| < |\rightarrow'|$.

Adapting the algorithm starts with an obvious step: During the partitioning phase, instead of considering regular transitions ($\rightarrow$), we need to consider weak transitions ($\Rightarrow$). However, the resulting algorithm still has two problems, both of which can be solved with a postprocessing step:

1. While the modified algorithm already guarantees to create the minimal number of states, the number of transitions is not guaranteed to be minimal. The reason is that when τ transitions survive minimization, the LTS could contain superfluous transitions. The can be easily removed:

For each transition $s \xrightarrow{\alpha} t$, consider a modified LTS where that transition is removed. If in that LTS, $s \xRightarrow{\alpha} t$ still holds (either because $s = t$ and $\alpha = \tau$, or because some τ transitions plus a different α -transition can emulate the removed transition), $s \xrightarrow{\alpha} t$ is superfluous and must be removed.

2. After this step, we are guaranteed to have a minimal representative up to weak bisimilarity. However, we need the result to be observation congruent. If it is not already observation congruent, too, this is because a τ -transition from the original state was removed. This can be easily corrected by reintroducing a τ -transition. To keep the number of states minimal, we do not introduce another state, but simply add a self loop to the initial state.

So, as a last step: If the original LTS had a transition $s_0 \xrightarrow{\tau} t$ from its initial state s_0 to a state t that lies in the same partition, i.e. $t \in [s_0]$, then modify the output LTS by adding the transition $[s_0] \xrightarrow{\tau} [s_0]$.

The resulting algorithm allows us to mechanically minimize an LTS up to observation congruence. Let's apply it to an example:

Interactive Exercise

book.pseuCo.com

Here, we apply the modified minimization algorithm to a first example.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/observation-congruence-minimization>

2.8 Workout

In this chapter, you've learned how to work with the different notions of equality that help us compare concurrent systems. In this section, you will find more examples for the exercise types you've seen in this chapter to help you hone your skills (plus some helpful tips).

Keep in mind that while the exercises in this workout section try to make you do all the hard choices, they still guide you – *it is important that you also practice solving these exercises with pen & paper only*. This is especially important for the proofs.

2.8.1 Bisimilarity

Proving Bisimilarity

When proving bisimilarity in more involved examples, it is easy to get lost. Try to create the full bisimulation before starting with the remainder of the proof to make the process a bit easier to navigate.

Interactive Exercise

book.pseuCo.com

Here, we'll prove bisimilarity in an example that may look like a spaghettified mess in the beginning.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-bisimilarity-workout-1>

Disproving Bisimilarity

When disproving bisimilarity using the bisimilarity game, you'll need to do two things:

1. find a relevant difference between the LTS & a reliable strategy to lead the defender there
2. write down a proof showing your strategy can win for every possible behavior of the defender

In many cases, which strategy you find in the first step decides how much work the second step will be. You should avoid unnecessary detours, but more importantly, make sure you minimize the number of possible defender moves. Where possible, it's more efficient to switch sides and force the defender down a specific path than to show you can win either way, simply because you have to consider fewer cases.

The next exercise is big enough that it can easily get out of hand – make sure you play strategically to keep it manageable. (The exercise will not let you refer to a previously-solved state like you could in a pen-and-paper proof. If you encounter the same state on two different ways, you will have to do double work.)

Interactive Exercise

book.pseuCo.com

This bisimilarity game can look deceptively simple, or turn into a huge monstrosity – it's all up to you.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-bisimilarity-workout-1>

Minimizing up to Bisimilarity

In Section 2.5.8, you learned how to minimize LTS up to bisimilarity by running an algorithm. While there are faster ways to minimize LTS, the algorithm is one of the easier ones. Let's practice it on another example:

Interactive Exercise

book.pseuCo.com

Here's another example for the minimization algorithm for bisimilarity.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/strong-bisimilarity-minimization-workout-1>

2.8.2 Weak Bisimilarity

Proving Weak Bisimilarity

Once you have mastered bisimilarity proofs and understood how weak transitions work, proving weak bisimilarity should quickly become second nature.

Interactive Exercise

book.pseuco.com

In this exercise, you'll prove weak bisimilarity on a larger example than in the main section.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/proving-weak-bisimilarity-workout-1>

Disproving Weak Bisimilarity

Remember that when using the game to disprove weak bisimilarity, using weak transitions can be crucial to keep the proof short. Still, you'll often have to consider multiple moves of the defender. Make sure you always have a plan before you start attacking!

Interactive Exercise

book.pseuco.com

Here's another weak bisimilarity game for you to win (or get lost in).

Solve this exercise online:

<https://book.pseuco.com/#/interactive/disproving-weak-bisimilarity-workout-1>

2.8.3 Observation Congruence

Minimizing up to Observation Congruence

Just as for bisimilarity, minimizing up to weak bisimilarity with the algorithm needs patience:

Interactive Exercise

book.pseuco.com

Here's another example for the minimization algorithm for observation congruence.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/observation-congruence-minimization-workout-1>

Chapter 3

Programming with pseuCo

3.1 A Gentle Introduction to pseuCo

3.1.1 About this Chapter

This chapter introduces you to pseuCo, a small programming language made to teach and learn concurrent programming.

Please note that pseuCo Book can only offer a cursory tour through the main features of pseuCo. A lot of practice outside of pseuCo Book will be needed to make you an expert!

3.1.2 Sequential pseuCo

The pseuCo programming language is a highly simplified language and is based upon a Java-like syntax. If you are familiar with programming in Java, C, JavaScript, or similar languages, you will feel right at home. Here is a first example of a simple pseuCo program:

```
1 int factorize(int input) {
2     int res = 1;
3
4     for (int i = 1; i <= input; i++) {
5         res *= i;
6     }
7
8     return res;
9 }
10
11 mainAgent {
12     int fac10 = factorize(10);
13     println("The factorial of 10 is " + fac10);
14 }
```

One aspect that may stand out is the `mainAgent`. This is where execution starts. It is similar to how `public static void main(String[] args)` works in Java, but has dedicated syntax instead of looking like a normal procedure declaration.

While pseuCo itself is already highly simplified, for now, we'll consider an even more restricted subset. In addition to not allowing some more advanced features we will introduce later, we

will only consider programs adhering to the following restriction:

All variable declarations must be local, i.e. within the `mainAgent` or a procedure.

To get your feet wet (as far as programming in `pseuCo` is concerned), let's start with a simple Hello World program.



Interactive Exercise

book.pseuCo.com

In this exercise, you'll write your first `pseuCo` program.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-sequential-hello-world>

3.1.3 Starting Agents

So far, programming in `pseuCo` did not involve concurrency. It's time to change that.

In principle, using concurrency in `pseuCo` is easy. Whenever you could call a procedure `doSomething(arg1, arg2)`, you can instead write `start(doSomething(arg1, arg2))` to *start* the procedure as an *agent*. This will cause the procedure to run independently from the caller (e.g. the `mainAgent`), allowing the caller to resume working before the started procedure has finished.

Starting procedures introduces some restrictions:

- Because the caller may resume working before the procedure has finished, the return value of the started procedure is not accessible.
- Arrays and structures (which will be covered in more detail in later sections) may not be passed to a procedure that is being started as an agent. (We will learn why, and eventually lift this restriction, later on.)

With this out of the way, let's look at our first example:

```
1 void countGerman() {
2     println("eins");
3     println("zwei");
4     println("drei");
5 }
6
7 mainAgent {
8     start(countGerman());
9     println("one");
10    println("two");
11    println("three");
12 }
```

In this example, two agents are counting in two different languages. Because both can continue without waiting for the other, their output is intermingled. In `pseuCo.com`, you can explore the corresponding LTS to identify which outputs are possible.

🔗 Dual Counting – What happens?

book.pseuCo.com

Which outputs can the example above produce (with newlines represented by spaces)?

- (a) `"eins_zwei_drei_one_two_three"`
- (b) `"one_two_three_eins_zwei_drei"`
- (c) `"oneins_e_two_thrzwei_ee_drei"`
- (d) `"one_zwei_drei_two_three_eins"`
- (e) `"one_eins_zwei_two_drei_three"`

Solve this quiz and check your solutions online:
<https://book.pseuco.com/#/quiz/dual-counting-outputs>

At this point, being able to start agents is, arguably, useless, because agents cannot communicate or interact with each other in any way after they are started. Soon, we'll look into two different approaches for lifting this restriction.

3.1.4 Intermezzo: The Dangers of Uncoordinated Access to Shared Memory

If our biggest problem so far is that agents cannot communicate in any way, this poses a rather obvious question: Why did we limit declarations of variables to local variables only? If we simply remove this restriction, we can easily get agents to cooperate by having them read and write the same variables (i.e. using *shared memory*).

Let's try this out and see what happens!

🔗 Interactive Exercise

book.pseuCo.com

In this dangerous expedition, we'll do the unthinkable and use a global variable to establish communication between agents.

Solve this exercise online:
<https://book.pseuco.com/#/interactive/uncoordinated-shared-memory-hello-world>

If you complete this example (and analyze its behaviour in pseuCo.com), you'll see that it works fine. Sure, repeatedly checking a variable until its value is as expected is inefficient, seems inelegant and raises some questions (Is there any guarantee it will eventually succeed?), but apart from that, is there any issue with this program?

As it turns out, there *are* significant issues, and *we consider the example above to be incorrect*. We'll explore the details now.

Uncoordinated Shared Memory is Fragile

Let's consider another innocuous-looking example: A program that counts down from 6 to 0 using two agents:

Listing 3.1: Concurrent counting with two agents

```
1 int n = 6;
2 bool done = false;
3
```

```

4 void countDown() {
5     for (int i = 0; i < 3; i++) {
6         n--;
7     }
8     done = true;
9 }
10
11 mainAgent {
12     start(countDown());
13
14     for (int i = 0; i < 3; i++) {
15         n--;
16     }
17
18     while (!done) {};
19     println(n);
20 }

```

Which results can this program print? Take a guess, use pseuco.com to confirm it, then answer below:

🔖 **Concurrent Counting: What the Frick?**

book.pseuco.com

Which outputs can our concurrent counting program produce?

- (a) -1
- (b) 0
- (c) 1
- (d) 2
- (e) 3
- (f) 4
- (g) 5
- (h) 7

Solve this quiz and check your solutions online:

<https://book.pseuco.com/#/quiz/concurrent-counting-results>

If you are surprised by the behaviour of the program, you are not alone. What is going on here?

To understand this example, it is important to understand what `n--` does. Of course, this statement decrements `n` by one – but how, exactly, does it do that? In most programming languages, including `pseuco`, `n--` runs (at least) three separate actions:

1. read the variable `n`
2. compute the new value to be written
3. write the new value to `n`

These actions are fully separate – as if `n--` was syntactic sugar for `int tmp = n; tmp = tmp - 1; n = tmp`. This means that if multiple agents are running at the same time, other agents can change a variable while one agent is working on it. For example, here's one way how the result 3 can occur:

1. The `mainAgent` starts `countDown()`.
2. The `mainAgent` begins the first decrement of `n`, so it reads the value 6.
3. Now, `countDown()` starts operating, and runs to completion in one go: It reads 6, decrements it to 5, and so on, until `n` is 3 and the flag `done` has been set.
4. Now, `mainAgent` continues working: Using the already read value 6, it continues working by computing and writing the new value 5.
5. The `mainAgent` decrements `n` two more times, with the final result being 3.

This is just one of many possible executions – there are many, and it’s quite unintuitive which results can occur. Did you guess correctly?

To sum up: *Any nontrivial program using shared memory without special care to coordinate variable accesses is probably wrong.*

Uncoordinated Shared Memory is Completely Broken

So far, we’ve seen that using shared memory is very fragile and can yield to surprising and incorrectly behaving programs. But what happens if we take care to consider every possible execution? Can we craft a program that uses shared global variables correctly?

So far, we’ve only considered the formal semantics of `pseuCo` programs, given by their translation to a CCS process and, ultimately, to an LTS. However, to fully understand the behaviour of concurrent programs using shared memory, it is important to consider the real world, too.

There, a program is executed by real hardware: A *central processing unit*, CPU for short, that contains multiple *cores* that allow running multiple *threads* concurrently. Variables are stored in *RAM*. Because RAM is slow, it’s typically not accessed directly: Between each CPU core and RAM, there’s a very fast *cache* that can store copies of small areas of RAM.¹ And, of course, CPUs *don’t* run `pseuCo` programs directly line-by-line: Programs are translated to *machine code* containing *instructions* that are defined for a specific type of CPU only.

Typically, all this is not important to you as a programmer. The compiler, in coordination with the CPU and cache controllers, keeps these details hidden and manages them for you. However, this changes when you use shared memory: You’ll be exposed to implementation details of the compiler and CPU architecture. For example, you may see...

- write accesses to variables appearing to be lost (because they get stuck in a cache),
- values appearing in variables that should never have been written (e.g. because a larger value is written in multiple steps² and another thread manages to see a halfway-written value), or
- statements appearing to be executed in the wrong order (because the compiler or CPU swapped them in a way not noticeable to a single thread, but noticeable from another thread).

Instead of trying to understand these phenomena, for now, we’ll try to avoid them instead. They can only occur if a program contains a *data race*.

Definition 27 (Data Race). A program contains a *Data Race* if it allows an execution which contains a state from which two different agents can access the same memory cell (e.g. the same variable), with at least one access being a writing access.

We consider a `pseuCo` program containing a data race to be incorrect even if its CCS-based semantics only exhibit correct behavior.

¹In fact, there usually is a hierarchy of caches, some shared between CPU cores.

²for example, a 64-bit number may be written in two 32-bit halves

The same holds for other programming languages: Unless you have read the language specification and know otherwise, you should consider any program in any programming language to be incorrect if it has a data race.

For now, we will fully eschew shared memory by disallowing global variables. Instead, in the next section, we'll introduce a way for `pseuCo` agents to communicate without causing data races. Later on, we will get to know dedicated mechanisms for moderating access to global variables, allowing us to start using them without risk.

3.2 Message Passing

In this section, we will learn a way to allow `pseuCo` agents to safely communicate. Instead of allowing implicit communication by allowing shared access to global variables, we will introduce a way for agents to communicate explicitly by exchanging messages through *channels*.

3.2.1 Introducing Channels

A channel is a “tunnel” allowing multiple agents to send and receive values. They come in two different flavors. For now, we'll focus on *synchronous* channels. They're available for `ints`, `bools` and `strings` and can be declared just like other variables:

```
1 intchan c1;
2 boolchan c2;
3 stringchan c3;
```

There's no need to explicitly initialize them – when you declare a channel, it can immediately be used.

Channels can be passed to procedures (and, therefore, agents). Because channels are intended to be safely usable by multiple agents, they can even be declared globally.³

```
1 intchan gchan; // a global channel
2
3 void myAgent(intchan c) {
4   // can use gchan and lchan (from mainAgent) here
5 }
6
7 mainAgent {
8   intchan lchan; // a local channel
9   start(myAgent(lchan)); // pass it to a new agent
10 }
```

Of course, declaring channels and passing them around is not interesting by itself: We need to actually *use* them. To do so, there are two new syntax elements:

- You can send value `e` on channel `c` with the statement `c <! e` (where `e` is an arbitrary expression).
- You can receive a value from channel `c` with the expression `<? c`.
This expression is often used as part of a declaration or an assignment (like `int v = <? c`), but it is a normal expression and may be used anywhere, e.g. as an argument to a procedure or `println`.

³In return, channel variables may not be reassigned – *channels* are safe to be used concurrently, but the variables storing them are not!

Synchronous channels do not store values – they are a means of instantaneously passing a value from one agent to another. This means that both trying to send a value to a channel and trying to receive a value from a channel will *block* (prevent the agent from proceeding) until a partner is available, i.e. another agent that wants to perform the other action on the same channel.

It's time for you to write your first concurrent message-passing program! We'll start with a variant of the traditional Hello World program:

🔖 Interactive Exercise

book.pseuCo.com

In this exercise, you'll use a channel for the first time.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-message-passing-hello-world>

🔖 Making It Sequential

book.pseuCo.com

What happens if, in your solution to the exercise above, you replace the `start()` with a regular procedure call?

- (a) The program still prints "HelloWorld!" as before.
- (b) The procedure deadlocks because no value can be retrieved from the channel.
- (c) The statement that sends "World" is never reached.

Solve this quiz and check your solutions online:

<https://book.pseuco.com/#/quiz/pseuco-message-passing-hello-world-sequential>

In addition to synchronous channels, pseuCo also supports *asynchronous* channels. Asynchronous channels have fixed *capacity* which is specified as part of its type when declaring a channel. So, for example, `intchan5 c;` declares a channel for integers that can store up to 5 values.

An asynchronous channel is never used by two agents at the same time – it only serves as a message box where a fixed number of values can be deposited and retrieved by other agents (or even the the same agent). Just like synchronous channels, they are *blocking*: If an action on an asynchronous channel cannot be performed (because no value is available to be received or no space is available for a value to be sent), the agent has to wait until the action can be performed. Asynchronous channels operate under the first-in-first-out principle.

Syntactically, they are used just like synchronous channels.

🔖 Interactive Exercise

book.pseuCo.com

In this exercise, you'll use an asynchronous channel for the first time.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-message-passing-hello-world-async>

On a side note, when a procedure is taking a channel as an argument, you do not need to specify whether the channel is synchronous or asynchronous. For example, if a procedure is declared as `void factorial(intchan c){ ... }`, it can be called with an `intchan` or an `intchan3` (but not with a `boolchan`).

3.2.2 Select-Case

So far, we've seen the expressions `c <! v` and `<? c` to send and receive values. As discussed previously, they are blocking, causing agents to wait until the channel operation has been completed. This behavior is typically intended, but can be restrictive in some cases.

Let's consider this example:

```

1 void factorial(intchan c) {
2     int z, j, n;
3     while (true) {
4         z = <? c; // receive input
5
6         n = 1;
7         for (j = z; j > 0 ; j--) {
8             n = n*j;
9         }
10
11        c <! n; // send result
12    };
13 }
14
15 mainAgent {
16     intchan cc;
17     agent a = start(factorial(cc));
18     cc <! 3;
19     int mid = <? cc;
20     println("3! evaluates to " + mid + ".");
21     cc <! mid;
22     println("(3!)! evaluates to " + (<? cc) + ".");
23 }
```

This `pseuco` program contains an agent that can compute the factorial of a number. It waits for a value to arrive on a channel, locally computes the factorial, then sends the result back on the same channel. The main agent uses this to compute $(3!)!$.

The `factorial` agent works in a `while(true)`-loop so it can be reused. Let's assume we wanted to add a way for other agents to terminate the `factorial` agent. To do so, of course, we have to somehow send a message to the agent, asking it to terminate. While we could define a specific numeric value (like `-1`) to be used as a signal, let's assume we didn't want to do that, ensuring that the `factorial` agent responds to *any* number sent on the channel `c` with its factorial. Can we simply introduce another `boolchan` for the termination signal?

We can – but if we try to use it, we run into a problem. As `factorial` does not know whether it should compute a factorial or terminate, it has to check whether someone is sending a number to be factorized on `c`. The only way to do so is to evaluate `<? c` – but this blocks until a value is available, making it impossible for the agent to react to a termination signal. What's missing is a way for an agent to wait for multiple channel operations at the same time (until one can be executed).

This is enabled by the `select-case`-statement. It allows to specify multiple cases, each one consisting of one channel operation and some code to be run if the operation succeeds. Only one case will be taken. Optionally, one or multiple `default` cases can be specified, which may always be selected⁴.

The syntax is simple:

```

1 select {
```

⁴In particular, a `default` case can be selected even if a non-`default` case is available, too.

```
2     case i = <? c1: {
3         println(i);
4     }
5     case c2 <! j: {}
6     default: {
7         println("tick");
8     }
9 };
```

This statement can either receive a value from `c1` and print it, or send `j` on `c2`, or simply print `"tick"`.

Now, let's use this ability to extend the example above to include a mechanism to terminate the `factorial` agent:

Interactive Exercise

book.pseuco.com

In this exercise, we'll extend a message passing program to include termination using `select-case`.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-message-passing-select-case-termination>

3.2.3 Working with Message Passing

In theory, you now know everything there is to know about message passing. You've seen all the features and know how they work. In practice, of course, structuring programs using message passing is not always easy. To get better at doing so, it's not only important to practice – there are also some recurring patterns of using message passing that apply in a wide range of situations. In this section, we'll get to know some of these patterns.

Producer & Consumer

Let's consider the following example: We want to write a program that computes the factorial of all prime numbers (up to a certain limit) – so the program should output `2!`, `3!`, `5!`, `7!`, `11!`, and so on. Here is a sequential implementation of that in `pseuco`:

```
1 int factorize(int n) {
2     int f = 1;
3     for (; n > 0; n--) {
4         f *= n;
5     }
6     return f;
7 }
8
9 bool isPrime(int n) {
10    for (int i = 2; i < n; i++) {
11        if (n % i == 0) return false;
12    }
13    return true;
14 }
15
16 void primer(int max) {
17     int n = 2;
18     while(n <= max) {
19         if (isPrime(n)) {
```

```

20         println(factorize(n));
21     }
22     n++;
23 }
24 }
25
26 mainAgent {
27     primer(15);
28 }

```

Of course, our next step is to parallelize this program. There are many possible approaches to do so. For example, we could try to find a way to divide the prime numbers up between multiple agents. But for now, a different approach is easier: We will split up the tasks of finding primes and computing (and printing) factorials between two agents.

This approach lends itself well to message passing: The prime-finding agent somehow has to pass the primes it identifies to the factorial-computing agent, a perfect use case for a channel. If we use an asynchronous channel with enough capacity, this allows proper concurrency: This way, the prime-finding agent does not have to wait for the factorial computation to finish and can pre-compute and buffer a certain number of primes, which ensures the factorial-computing agent does not have to wait unnecessarily⁵ when the prime-finding agent encounters a longer gap between primes.



Interactive Exercise

book.pseuCo.com

It's time to implement the changes and make our prime-factorial-computing program concurrent!

Solve this exercise online:

<https://book.pseuco.com/#/interactive/message-passing-prime-producer-factorial-consumer>

The way both agents cooperate in this example is called the *Producer-Consumer-Pattern*. In this pattern, there are two kinds of agents: *Producers* create values that need further processing and send them onwards, and *consumers* receive these items and finish the work.

To visualize how the agents cooperate, we can draw a program's *channel structure diagram*. For our prime-factorial-example, depending on how exactly you implemented it, the channel structure could look like this:

The channel diagram contains a lot of information:

- It contains all agents started by the program.
- Every channel is indicated by a line. Arrows on the line indicate in which directions messages are sent on the channel. Asynchronous channels are denoted by a “buffer” symbol on the line.
- The diagram also shows the names of the channels. For channels that are passed around as arguments, the channel's local variable name is shown inside the box of every agent it is stored in. For channels stored in a global variable, the variable's name is printed outside of any agent.

In our example, we had a single producer and consumer each. It is also possible to have multiple producers and consumers, all sharing a single channel to pass on the intermediate items.

⁵Of course, most likely, one of the agents will have to wait regularly, as one of the jobs is likely faster than the other. What we want to prevent, however, is that *both* agents often have to wait for each other, as this means there's additional slowdown due to an inefficient communication pattern.

Figure 3.1: Channel structure diagram of a producer-consumer-pattern with a single producer and consumer each

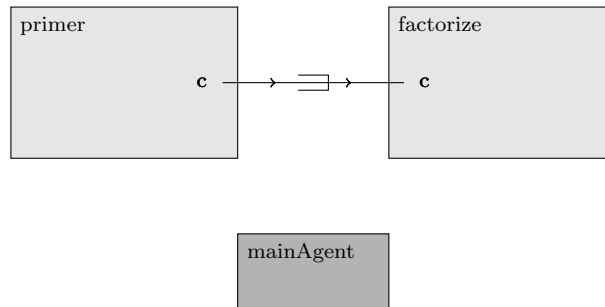


Figure 3.2: Channel structure diagram of a producer-consumer-pattern with a multiple producers and consumers

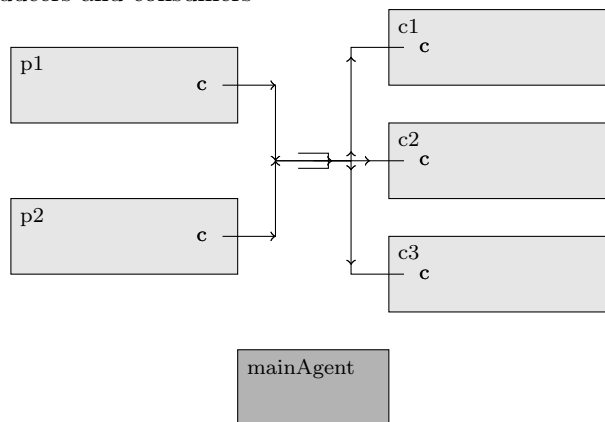
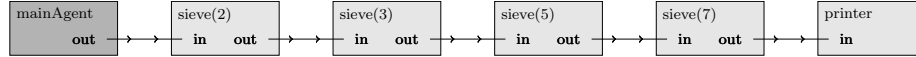


Figure 3.3: Channel structure diagram of a pipeline-based prime sieve



In this case, it is important to understand how this affects the order in which items are processed:

- With a single producer and consumer, everything is processed in-order: As channels operate under the first-in-first-out paradigm, items cannot overtake each other, and items are guaranteed to be consumed in the order they have been produced.
- If there are multiple consumers, they will retrieve the values from the channel in the order they have been put in, but any further processing done by a consumer is independent of the other consumers. In our example, if we used two consumers and one consumer happens to finish computing a factorial before the other consumer has finished computations for the previous prime, the results could be printed out-of-order. (Whether such reorderings are allowed behaviour depends on the problem statement.)
- If there are multiple producers, they produce the intermediate items in an uncoordinated way, so they will arrive at the consumers unordered.

Pipelining

In the previous section, we had considered an example where we wanted to compute something for every prime number, and we parallelized it by splitting prime number generation and the computation between two kinds of agents. Now, let's take a deeper look into just the former part: generating prime numbers.

Prime-number generation does not appear to fit into the producer-consumer pattern. How could one parallelize it instead? An obvious approach would be to somehow split the candidate numbers between multiple agent, have each agents check its candidates, and then collect the primes. While this works, it has a few disadvantages, most notably that this means the primes are generated out-of-order.

Another approach is based around the observation that checking whether a number $n \leq \max$ is prime requires running a few distinct checks for every candidate, namely whether it is dividable by 2, 3, 5, $\dots$, and so on for every prime $p \leq \sqrt{\max}$. For simplicity, let's assume that we already know this smaller list of prime numbers (and are willing to hard-code them into our program). What happens if we divide these checks between agents?

The resulting structure is somewhat reminiscent of a *pipeline*: Numbers enter a long series of steps, connected by one channel each, during which numbers are processed (and, therefore, sometimes filtered out).

With the channel structure diagram at hand, writing the program is not too challenging:

Interactive Exercise

book.pseuco.com

Here, we implement a pipelining-based prime sieve.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/message-passing-pipelining-prime-sieve>

3.3 Shared Memory

As you have already seen in Section 3.1.4, working with shared memory is perilous – *Data Races* are lurking everywhere, causing programs to be incorrect. In this section, instead of eschewing shared memory and using channels instead, we will learn how to use shared memory safely.

At its core, making shared memory *safe* means to place limits on the parallel execution of agents – limits that leave as much parallelism as possible, but prevent any execution that causes a data race or any incorrect computation to happen.

In the following section, we'll learn about the simplest way of limiting parallelism.

3.3.1 Introducing Locks

In this section, we'll start working with *locks*. Syntactically, working with locks is easy. They support three operations:

```
1 lock l;      // declare a lock
2 lock(l);    // lock a lock
3 unlock(l);  // unlock a lock
```

Locks are simple data structures that can be locked and unlocked. (When an agent has locked a lock, we often say this agent *holds* or *owns* the lock.) They can only be locked once⁶ – if an agent is trying to lock a lock that another agent owns, the attempts to lock the lock will *block* until the lock is free (just as attempting to read from a channel where no value is available blocks until the situation changes).

Their main feature, however, is that *locks never cause data races*⁷. Locks are designed to be operated concurrently, and they will use whatever means is necessary and available (depending on what platform the code is being compiled for) to ensure using them concurrently works as expected.

Now, we can start using locks to control code execution. It's important to understand that *by themselves, locks do not prevent data races* – but they can be used to prevent them. To do so, we ensure that for every shared variable, every access to it is guarded by a lock, i.e. this lock is locked before any access, and is unlocked only after the variable has been left in a state that is safe for other agents to operate on.

It's important to consider the fine print of this approach:

- Every access to the shared variable (that could be executed while other agents are running) needs to be guarded with the lock – locking a lock does not automatically prevent other agents from accessing the associated variable, only from obtaining the same lock.
- For every shared variable, every access must be guarded by *the same lock*. If multiple locks are used for the same variable (in the sense that different agents use different locks), multiple agents can hold the lock at the same time and access the variable concurrently. (Protecting a shared variable with multiple locks in the sense that every agent must hold *all* relevant locks to access the variable is safe, but generally does not provide any advantage over using a single lock.)
- It's important to not just blindly add `lock` and `unlock` statements around every variable access, but to ensure each lock is held until the associated variable can be safely accessed by others again. For example, if a global variable is read, a new value is computed,

⁶We will discuss an exception to that rule later on.

⁷This only holds for the locks themselves – if you attempt to *reassign* lock variables while others may read them, you are in trouble!

and then written to the global variable, in most cases, the lock should not be returned between the read and write access. Failing to do so properly results in programs that have no data races, but still produce incorrect result because variables are changed at unexpected times.

Before we'll start working with locks, here is another small shared-memory feature that makes working with multiple agents easier:

```
1 agent a = start(test()); // agents can be stored when they are
   ↳ started...
2 join(a); // ...which allows waiting for their termination later on.
```

Now, we'll reconsider the scenario from Section 3.1.4, and introduce a lock to make it work correctly.

Interactive Exercise

book.pseuco.com

In this exercise, we protect an example program where two agents operate on a shared counter by introducing a lock.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-shared-memory-locked-counting>

3.3.2 Locking Locked Locks

So far, we've said that locks can only be locked once. Upon first inspection, this makes a lot of sense – it is the point of using locks, after all.

However, when using locks in more involved cases, this leads to a problem. As an example, let's consider an example where we have a counter that is protected by a lock:

```
1 int n;
2 lock l;
3
4 int get() {
5     lock(l);
6     int res = n;
7     unlock(l);
8     return res;
9 }
10
11 void step() {
12     lock(l);
13     n++;
14     unlock(l);
15 }
```

Think of the counter as a stand-in for a more complicated data structure. We'll consider the two methods `get()` and `step()` to be the only methods allowed to touch `n` directly so the implementation of our data structure is abstracted away from the rest of our code.

Now, assume we want to write a method that can increment `n` multiple times *atomically*, without giving anyone else the chance to observe the intermediate value. With what we know so far, this is actually impossible:

- If we call `step()` without taking the lock `l` before, someone else could `get()` the value before we can finish all `step()`s.

- We can take the lock `l` to prevent this – *but then `step()` cannot get the lock anymore.*

There are ways to solve this dilemma, for example, we could offer an “unsafe” version of `step()` that does not take the lock, to be used in contexts where we know the lock is already held. However, all of these ways require code duplication or make it easier to accidentally introduce a data race.

Instead, we can solve this problem using a feature of `pseuCo` locks:

```
1 void stepMulti(int steps) {
2     lock(l);
3
4     for (int i = 0; i < steps; i++) {
5         step();
6     }
7
8     unlock(l);
9 }
```

If you inspect the `stepMulti()` procedure above, you will not find anything special – *it just works*. The reason is that locks in `pseuCo` are *reentrant*. This means that their behavior in the case an agent is trying to lock an already locked lock again depends on the agent’s identity:

- If the lock is currently held by *a different* agent than the one trying to lock it, the attempt blocks until the lock is free.
- If the lock is currently already being held by *the same* agent that is trying to lock it, the attempt succeeds. The result is that the agent has two “copies” of the lock. This can be repeated multiple times, with the lock keeping track of the number of copies owned by the agent.

If an agent has obtained multiple copies of a lock, calling `unlock()` only returns a single copy of the lock. A lock is only fully returned (and therefore free for other agents) if all copies have been returned.

This ensures that you are free to call procedures that will take locks you already own and enables recursive use of procedures holding locks.

3.3.3 Arrays, Structures, and Sneaky Data Races

Like many programming languages, `pseuCo` features *arrays* and *structures*. Both are relatively straightforward by themselves, but come with a catch we’ll discuss later on.

Introducing Arrays

`PseuCo` allows declaring arrays of arbitrary types using Java-like syntax:

```
1 mainAgent {
2     int[5] a1;           // an array of integers
3     a1[2] = 42;         // writing to arrays
4     println(a1[2]);     // reading from arrays
5
6     int[3] a2 = { 1, 2, 3 }; // explicit initialization
7     int[100] a3 = { 42, 1337, }; // incomplete initialization
8
9     lock[5] a4; // lots of locks
10
11     // agent arrays must be initialized
12     agent[2] a5 = { start(doIt()), start(doIt()) };
```

13 }

Introducing Structures

Similarly, pseuCo allows declaring and using structures:

```

1 struct Pair {
2     int a, b = 0;
3
4     void setA(int aa) {
5         a = aa;
6     }
7
8     void setB(int bb) {
9         b = bb;
10    }
11
12    int getSum() {
13        return a + b;
14    }
15 }
16
17 mainAgent {
18     Pair p;
19     p.setA(42);
20     p.setB(1337);
21 }
```

PseuCo does not allow customizing the visibility of fields and methods:

- Fields are only accessible from within the struct, i.e. they act like “private” fields.
- Methods can be called by anyone having a reference to the struct and from within the struct, i.e. they act like “public” methods.

A Heap of Problems

Just as in many other languages like Java, in pseuCo, arrays and structures are stored on the *heap*, a central storage place for data that may be used in multiple places. The actual variable that has an array- or struct-type does not store the data itself, only a *reference* to the location where the data is stored on the heap.

In many cases, this is an implementation detail the programmer does not need to know about – arrays and structs can be used as if they were stored directly in the corresponding variables. This starts to change, however, as soon as an array or structure reference is duplicated, e.g. by passing it to a procedure. In this case, changes to the array or struct made using one reference become visible through the other reference as well. (In a sense, this is analogous to channels in pseuCo-MP, where „duplicating“ a channel results in multiple variables granting access to the same underlying channel.) For example, this means that a procedure can modify an array directly:

```

1 void swap(int[2] a) {
2     int tmp = a[0];
3     a[0] = a[1];
4     a[1] = tmp;
5 }
6
7 mainAgent {
```

```
8     int[2] a = { 42, 1337 };
9     swap(a);
10    println(a[1]);
11 }
```

For sequential programs, this behavior is often useful (albeit sometimes confusing). However, for concurrent programs, this exposes significant risk: *Arrays and structures can cause data races on the heap*. Accessing the same field of the same array or structure instance concurrently (with at least one access being a writing access) is a data race even if the access happens through a different variable.

Interactive Exercise

book.pseuCo.com

It's time to try this out and create a top-secret data race.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-shared-memory-hidden-data-race>

Conceptually, these types of data races are nothing new, and they can be prevented using the same techniques as data races on global variables – but they are easier to miss, so extra care is required when working with arrays and structures. In the next section, we will discuss a systematic way to prevent these types of data races.

3.3.4 Protecting Structured Data with Monitors

With our new-found ability to structure data also comes the task of protecting that data properly. For example, consider this data structure that can store a single message (consisting of an integer) or be empty (indicated by the value `-1`) and is emptied when read:

```
1 struct MessageBox {
2     int message = -1;
3
4     void putMessage(int msg) {
5         message = msg;
6     }
7
8     int getMessage() {
9         int tmp = message;
10        message = -1;
11        return tmp;
12    }
13 }
```

So far, this structure does not take concurrency into consideration. Of course, this is not required for concurrent use – locking can always be done outside of the structure by its user by declaring a lock and obtaining it before interacting with the `MessageBox`. However, it would be more convenient to have the `MessageBox` take care of this.

This is rather easy to do, and is a recurrent pattern: We simply declare another field holding a lock *inside* the structure and ensure every method of the structure acquires this lock at the start and returns it after the last access to a field, just before returning a value. (If the `return` expression accesses a field, it has to be extracted to a new temporary variable.)

Let's try this!

 Interactive Exercise

book.pseuCo.com

In this exercise, we make our structure safe by adding an internal lock.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-shared-memory-locked-message-box>

A data structure that correctly follows this pattern, ensuring all its internal data is safe from data races by full protecting every method that can access it with a single lock, is called a *monitor*. When using shared memory concurrency, monitors are the safest and most comfortable data structures to work with, as they can safely be used concurrently from multiple agents.

You should keep in mind the following pitfalls, though:

- While monitors cannot have data races internally, the rest of your program can still have them. An especially nasty corner case are variables holding *references* to monitors – if you reassign them, these accesses are *not* protected by the monitors, and ensuring their safety is up to you.
- Monitors not having internal data races does not guarantee everything you do with them is correct: Whenever a method of the monitor returns, other agents could modify it – if you want to be able to run multiple operations on a monitor without others being able to interfere, you must either introduce additional locking or modify the monitor to include a method that runs all these operations at once.

pseuCo Monitors

Protecting every method of a struct with a lock is effective, structurally simple, annoying to have to do, and easy to forget – so an automated solution seems in order. Indeed, pseuCo supports a way to automate this, simply by replacing the `struct` keyword with `monitor`:

Listing 3.2: A first monitor implementing a message box for integers

```
1 monitor MessageBox {
2     int message = -1;
3
4     void putMessage(int msg) {
5         message = msg;
6     }
7
8     int getMessage() {
9         int tmp = message;
10        message = -1;
11        return tmp;
12    }
13 }
```

No further changes are necessary. Here's how the `monitor` keyword works:

- Any structure can be turned into a monitor.
- A `monitor` uses a managed (“*implicit*”) lock that is automatically locked and unlocked.
- `return` statements are handled automatically, releasing the lock only after the return value has been computed.
- The implicit lock cannot be accessed or manually locked or unlocked.
- There is no way for a method to opt out – if a structure is declared as a `monitor`, all code inside it is executed while holding the lock.

- A `monitor` may also use advanced features that we will introduce later on.

Note that *monitor* refers to a generic concept, implemented in `pseuCo` with the `monitor` keyword. Every `pseuCo monitor` is a monitor, but a `pseuCo struct` can be a monitor, too.

Waiting for Conditions

When writing `pseuCo` programs using shared memory, one of the more difficult type of situations you may encounter is cases where an agent has to wait for some condition to be fulfilled.

For example, let's consider our `MessageBox` from Listing 3.2 again. In a sense, it implements an asynchronous integer channel of capacity 1 – but with one very notable difference: While a true message-passing channel blocks when trying to read a value that isn't there, our `MessageBox` instead returns a placeholder value.

Our goal will be to change `MessageBox` such that attempting to read a nonexistent value blocks. (For now, we'll focus on that case – writing into a `MessageBox` that already contains a value will simply overwrite what's there). Here's a first attempt:

```
1 monitor MessageBox {
2     int message = -1;
3     bool hasMessage = false;
4
5     void putMessage(int msg) {
6         message = msg;
7         hasMessage = true;
8     }
9
10    int getMessage() {
11        while (!hasMessage) {
12            // wait
13        }
14
15        hasMessage = false;
16        return message;
17    }
18 }
```

Unfortunately, this approach has a rather serious flaw. Take a moment to consider this, then answer this question:

Problems of Waiting in a Tight Loop

book.pseuco.com

Which statements about the preceding implementation of `MessageBox` are true?

- (a) When an agent running `getMessage()` is waiting for a message, the lock is not returned.
- (b) It is impossible to store a message in the box while an agent is actively waiting for it.
- (c) When an agent calls `getMessage()` while the box is empty, it will deadlock (or rather: livelock) forever.

Solve this quiz and check your solutions online:
<https://book.pseuco.com/#/quiz/tight-busy-wait-problems>

CHAPTER 3. PROGRAMMING WITH PSEUCO

So this is no good! We'll have to find a way for the lock to be released while waiting. As a `pseuCo` `monitor` does not allow us to manually release the lock, we have two options to do so:

- We could turn our `monitor` into a `struct` and manually manage the lock.
- We could go back to the approach of returning a special value if no message is available, and ask the user of `MessageBox` to run `getMessage()` in a loop until a message is returned (as we did in a previous exercise).

Both approaches introduce significant complexity – but they also have a more severe problem: When no message is available, the agent isn't actually *waiting*, it just attempts to check for a message again as soon as it can. This is problematic in many ways:

- This causes our program to waste resources for the repeated checks.
- In addition, the checking agent is making it harder for other agents to obtain the lock, causing `putMessage()` calls to be slowed down unnecessarily. (For a more extreme scenario, think of hundreds of agents running `getMessage()`, fighting for the lock, while a single agent tries to store hundreds of messages sequentially.)
- Whether a situation where an agent waits in `getMessage()` while another one tries to call `putMessage()` is guaranteed to *ever* succeed depends on the implementation of locks and the scheduler – in theory, one could implement a valid, deterministic scheduler that happens to only let the `putMessage()` agent run when the `getMessage()` agent is holding the lock!

While the first problems could be somewhat alleviated by introducing a sleep command, this would still cause unnecessary resource usage and delays.

We call such a situation a *busy wait* more specifically:

Definition 28 (Busy Wait). An agent is in a *busy wait* if

- it is waiting for some condition to be fulfilled,
- this condition can only be fulfilled by another agent,
- the waiting agent continues executing code ad infinitum while waiting, and
- the waiting agent performs no meaningful work while waiting.

We generally consider programs that may exhibit a busy wait to be incorrect.

A proper solution would require a way for us to temporarily return the lock and wait for some kind of notification indicating that we should check for a message again.

This is possible with *conditions*, a feature of `pseuCo` `monitors`. Using it correctly requires the use of three new statements:

- `condition c with b` declares a condition, with `c` being the name of the condition and `b` being an arbitrary boolean expression, indicating what this condition is waiting for. This can only be used inside a `monitor`.

You should declare one condition for every situation agents can wait for.

- The statement `waitForCondition(c)` waits for the condition `c` to be fulfilled.
- The statement `signalAll(c)` notifies all agents waiting on condition `c` to check the condition again.

If you know that any agent that is being notified would immediately falsify the condition again (e.g. because the condition is for a message to be available, and the notified agent would read and remove that message), you may use `signal(c)` instead. This statement notifies only a single agent, picked nondeterministically out of the pool of agents waiting for that condition.

In `pseuCo`, conditions have been designed to be mostly foolproof. Notably, (and in contrast to many other programming languages like Java,) the `waitForCondition` command handles all required steps in a single command:

- It returns the lock,
- signs up for notifications from `signal` and `signalAll` calls on the same condition and waits for one (in a way that ensures no notification can be missed between returning the lock and this step),
- re-obtains the lock,
- evaluates the expression that was specified when declaring the `condition` to determine whether it is now fulfilled, and
- repeats the whole process until the expression evaluates to `true`.

This ensures that after `waitForCondition(c)` returns (with `c` having been declared as `condition` `↳ c with b`), `b` is guaranteed to be true.

However, one responsibility falls on you as the developer: *Whenever any change you make to a monitor could cause a condition `c` of it to become fulfilled, you must signal this by calling `signalAll(c)` (or `signal(c)` where appropriate).* If you forget this, your programs can deadlock, as agents sitting in a `waitForCondition(c)` call may not notice they can resume working.

Keep in mind that signalling too much is always safe (and causes a small overhead), while missing a required signal causes deadlocks. Make sure you only use `signal(c)` if you are certain waking up a single agent is enough⁸, and double-check you always signal when you have made any change that could satisfy a condition.

It's time to put this into practice and turn our `MessageBox` into a proper replacement for an `intchan1`:

Interactive Exercise

book.pseuco.com

In this exercise, we'll improve our `MessageBox` to be able to wait for a message and for space to be available.

Solve this exercise online:

<https://book.pseuco.com/#/interactive/pseuco-shared-memory-monitor-message-box>

⁸Determining whether this is the case will be more difficult in other programming languages, e.g. in Java.