Symbolic compositional model checking for mCRL2

van der Pol, K.

Award date:
2013

Symbolic compositional model checking for mCRL2

January 15, 2013

Kevin van der Pol, BSc.
Master's Thesis
Computer Science and Engineering

Supervisor:
T. A. C. Willemse, PhD

Eindhoven University of Technology
Department of Computer Science and Mathematics

Abstract

In model checking, we check whether a given model of a system satisfies a given property. Often, the system consists of multiple components working in parallel. We use this information of the system's structure to reduce the model checking problem, in a procedure inspired by H. R. Andersen [1]. There are two alternating steps. First, we remove one parallel component from the model checking equation and change the property accordingly, to arrive at an equivalent problem of equal or greater complexity. Second, we reduce the property again. Hopefully, this results in an overall model checking problem of smaller complexity. We apply this technique to models in the mCRL2 language. Using a symbolic approach, we extend the partial model checking technique to models with infinitely large state spaces.

Contents

Introduction

The model checking problem is to determine whether a given system satisfies a given property. Usually, the system is an abstracted software or hardware system. Model checking can be an important step in validating the safety of a variety of systems, such as railroad switches and signals, or critical medical equipment. This becomes increasingly hard as systems become very large, which they typically are. Fortunately, such a large system usually consists of multiple simpler components working in parallel.

The compositional model checking technique, first proposed by H. R. Andersen [1], exploits this compositionality by operating on each of the components separately instead of calculating the more complex behavior of the entire system. The compositional model checking technique consists of two alternating steps: quotienting and reduction. Quotienting is moving part of the behavior from the system to the property to be checked on the remainder. This makes the system smaller, but the property larger. The next step is to reduce the property again. These steps make the model checking problem smaller and thus faster to solve.

We apply the partial model checking technique to process descriptions in mCRL2 (Groote *et al.* [2]). These process descriptions consist of atomic systems, called linear process equations, working in parallel, with several additional operators on their combined behavior. One of these operators is the communication operator, which provides explicit synchronization of the otherwise completely parallel systems. We extended the quotienting procedure to symbolically incorporate these operators in the property to be checked.

Andersen's technique requires the systems to have a finite state space, which is not typically the case. In mCRL2, one can concisely describe such systems with infinite state spaces. Therefore, we extend compositional model checking to systems with infinite state spaces.

After some preliminaries, we first introduce the descriptions of systems in Section 2: the linear process equations and the process operators working on them. Then, in Section 3, we introduce the formalism we use to describe the properties we wish to check on these systems: the parameterized modal equation systems, very similar to the well-known μ-calculus. The heart of this thesis is found in Section 4 and Section 5. There, we provide definitions and proofs for a symbolic quotienting procedure on parallel linear process equations and on process operators. Finally, in Section 6, we explain a number of reductions which can be performed on the properties we obtain.

1 Preliminaries

1.1 Notation

We use assignments of functions to change environments and data environments.

Definition 1.1.1 (Assignment)**.** An *assignment* of value $b \in B$ to $a \in A$ in function $f : A \to B$, denoted $f[a := b]$, is again a function with signature $A \to B$. It is defined as follows, for any $a' \in A$:

$$f[a := b](a') = \begin{cases} b & \text{if } a = a' \\ f(a') & \text{otherwise} \end{cases}$$

1.2 Knaster-Tarski's theorem

Knaster-Tarski's theorem[1] deals with monotone functions over a complete lattice and states that the fixpoints of f again form a complete lattice. We first introduce *monotonicity*, *complete lattices* and *fixpoints*:

Definition 1.2.1 (Monotonicity)**.** Let A and B be ordered sets. A function $f : A \to B$ is *monotone*, if and only if, for any $a \leq a'$, also $f(a) \leq f(a')$.

This is the first requirement of Knaster-Tarski's theorem. The second is that the monotone function is applied to elements of a complete lattice. The definition of a complete lattice, requires the notion of infimum and supremum:

Definition 1.2.2 (Infimum and supremum)**.** Let A be a set with ordering $\leq$. The *infimum* of some $A' \subseteq A$, denoted $\mathsf{inf}(A')$, is the greatest element of A which is smaller than any element of A':

$$\mathsf{inf}(A') = \mathsf{max}\{a \in A \mid (\forall a' \in A' \,.\, a \leq a')\}$$

The *supremum* of A', denoted $\mathsf{sup}(A')$, is dual.

Note that an infimum or supremum need not exist. For example, let A be the positive real numbers with the usual ordering $\leq$. Does the structure $(A, \leq)$ have an infimum for A? The greatest number that is smaller than all positive real numbers is zero, which is not an element of A. Therefore, $\mathsf{inf}(A)$ does not exist.

A *complete lattice* is a structure for which the infimum and supremum of any subset always exists:

Definition 1.2.3 (Complete lattice)**.** The structure $(A, \leq)$ is a *complete lattice*, if and only if, for every $A' \subseteq A$, there exists an infimum and a supremum in A.

Knaster-Tarski's theorem then concludes something about the fixpoints of f:

Definition 1.2.4 (Fixpoints)**.** An element $a \in A$ is a *fixpoint* of a function $f : A \to A$, if and only if, $a = f(a)$.

[1] Despite its name, Knaster-Tarski's theorem is not actually due to Knaster and Tarski. For clarification on the naming of Knaster-Tarski's theorem and related theorems, refer to Lassez, *et al.* [7].

Of particular interest are the *least fixpoint* and the *greatest fixpoint*:

Definition 1.2.5 (Least and greatest fixpoints)**.** The *least fixpoint* of a function $f : A \to A$, denoted μf, is that fixpoint $a \in A$ that is smaller than any other fixpoint $a' \in A$:

$$\mu f = a \in A, \text{ such that } a = f(a) \text{ and } (\forall a' \in A \,.\, a' = f(a') \Rightarrow a \leq a')\}$$

The greatest fixpoint, denoted νf, is dual.

Now, Knaster-Tarski's theorem is as follows:

Theorem 1.2.6 (Knaster-Tarski's theorem)**.** (Tarski [11])
Let $(A, \leq)$ be a complete lattice, $f : A \to A$ a monotone function and $P \subseteq A$ the set of all fixpoints of f. The set P is not empty and the structure $(P, \leq)$ is a complete lattice.

In particular, the least fixpoint $\mu f = \mathsf{inf}(\{a \in A \mid f(a) \leq a\})$ and the greatest fixpoint $\nu f = \mathsf{sup}(\{a \in A \mid a \leq f(a)\})$.

1.3 Transfinite induction

The least or greatest fixed point can be constructively defined by iteration of the monotone function, starting from the least or greatest element of A, respectively. As the domain may be uncountably large, the number of iterative application may therefore also be uncountable. This means mathematical induction does not suffice and we need transfinite induction.

Transfinite induction is an extension of mathematical induction to well-ordered sets, such as the class of ordinals Ord. The class of ordinals not only contains zero and the successor ordinals, but also limit ordinals:

Definition 1.3.1 (Limit ordinal)**.** An ordinal α is a *limit ordinal* if it differs from zero and is the supremum of all smaller ordinals. The first limit ordinal is commonly denoted ω.

Note that ω, the supremum of all natural numbers, is itself not a natural number.

We use a transfinite induction scheme, consisting of three parts: a base case for ordinal zero, an inductive step for a successor ordinal and a transfinite case for limit ordinals:

Definition 1.3.2 (Transfinite induction on ordinals)**.** Let $P(\alpha)$, where α is an ordinal, be a proposition. From the following three statements, we may conclude P holds for *any* ordinal:

- Base: $P(0)$
- Inductive step: $P(n) \Rightarrow P(n+1)$, for any $n \in Ord$
- Transfinite case: let α be a limit ordinal. $(\forall \beta < \alpha \,.\, P(\beta)) \Rightarrow P(\alpha)$

We apply transfinite induction to finding the least or greatest fixpoint of a function. We define the *σ-approximants* F^α_σ, where α is an ordinal, for finding σf, as follows:

Definition 1.3.3 (σ-approximants)**.** Let $f : A \to A$ be a function and let $\bot$ and $\top$ be the least and greatest elements of A, respectively. The *σ-approximant* F^α_σ, where α is an ordinal, is defined as follows:

- $F^0_\mu = \bot$
- $F^0_\nu = \top$
- $F^{n+1}_\sigma = f(F^{n+1}_\sigma)$
- $F^\alpha_\mu = \mathsf{sup}(\{F^\beta_\mu \mid \beta < \alpha\})$, for limit ordinal α
- $F^\alpha_\nu = \mathsf{inf}(\{F^\beta_\nu \mid \beta < \alpha\})$, for limit ordinal α

From Theorem 1.2.6 (Knaster-Tarski's theorem) and this definition of σ-approximants, we obtain a constructive definition of the least and greatest fixpoints:

Definition 1.3.4 (Constructive least and greatest fixpoints)**.** Let $(A, \leq)$ be a complete lattice and $f : A \to A$ a monotone function.

$$\mu f = \mathsf{sup}(\{F^\alpha_\mu \mid \alpha \in Ord\})$$

$$\nu f = \mathsf{inf}(\{F^\alpha_\nu \mid \alpha \in Ord\})$$

Furthermore, there is some ordinal for which this fixpoint is reached and because it is a fixpoint, all higher σ-approximants are equal. Formally: there exists an ordinal α of cardinality at most the cardinality of A, such that for $\beta \geq \alpha$:

$$\sigma f = F^\beta_\sigma$$

1.4 Previous work

This work has its origin in the work of H. R. Andersen [1], who in 1995 described a quotienting procedure for parallel composition of finite labeled transition systems (cf. Section 2.3 (Labeled transition systems)). This parallel composition had an implicit synchronization mechanism, where for any action a, there is a counterpart $\bar{a}$, which result in a silent action τ when occurring simultaneously. This quotienting procedure was very effective at reducing the complexity of large numbers of simple, finite processes, in parallel.

Other interesting work on the subject is by F. Lang and R. Mateescu [6], who came to similar results with a graph-based procedure. Again, this was restricted to finite processes in parallel composition.

In an effort to take this to the mCRL2 setting, K. van der Pol [12] considered labeled transition systems with multi-actions and the process operators parallel composition, communication and allow. Still, this required the processes to have enumerable state spaces.

The current work seeks to expand this to linear process equations in a syntactic procedure that does not require the state space of a process to be finite.

2 Models

In this section, we introduce formalisms to describe processes. First, we describe the data we allow the user to define in the tool. We abstract from how the user enters this information and only limit ourselves to some necessary requirements on what the user may enter. Secondly, we describe actions. The toolset mCRL2 uses multisets of actions called multi-actions to denote actions occurring simultaneously. Then we look at the well-known labeled transition systems and their syntactic representation in the mCRL2 toolset: the linear process equations. Lastly, linear process equations are combined with process operators, e.g. parallel composition, to form process descriptions. We formalize linear process equations and the process operators working on them.

Readers who are already familiar with mCRL2 process descriptions are encouraged to skip this section. For those readers, we can summarize this section as follows:

- There are no assumptions on user-defined data sorts, except the existence of an equality relation $\approx$. A boolean data sort is assumed.
- Action formulae have no quantifiers and are in disjunctive normal form with boolean expressions, multi-actions and negated multi-actions as literals.
- The most basic description of a process we consider, is linear process equations.
- We consider parallel composition $||$, the communication operator Γ, the allow operator ∇, the rename operator ρ and the abstraction operator τ on linear process equations. Expressions with linear process equations and process operators are called "process descriptions".
- The communication operator is slightly generalized, with *multi*-actions as possible substitution results. For example, $\Gamma_{\{a|b\to c|d\}}(P)$ would substitute a transition label $a(1)|b(1)$ with $c(1)|d(1)$.
- The allow set of the allow operator is a set of multi-actions, e.g. $\{a|b, a\}$.
- Similar to the communication operator, the rename operator permits *multi*-actions as substitution results.
- A rename operator without explicit rename set, renames *all* action names to fresh ones. This is done by adding a prime symbol as a decorator. For example, $\rho(P)$ would substitute a transition label $a(1)|b(1)$ with $a'(1)|b'(1)$.
- Similar to the allow operator, the abstraction set of the abstraction operator is a set of multi-actions, e.g. $\{a|b, a\}$.

2.1 Data

A description of the types of data to be used, is taken as input to the model checking tool.

Definition 2.1.1 (Data sort)**.** A data sort is a non-empty set of expressions with semantics in a corresponding domain. We assume a set $\mathbb{D}$ of data sorts, which are ranged over by $D_1, D_2, \ldots$. The semantic domains of data sorts $D_1, D_2, \ldots$ are denoted $\mathcal{D}_1, \mathcal{D}_2, \ldots$. These semantic domains may not be empty.

For each defined data sort, we assume a set Dvar of data variables and a set func of functions. Constants are functions with zero arguments. Each data variable $x \in \mathsf{Dvar}$ has a type D and each function $f \in \mathsf{func}$ has a profile $D_1 \times \cdots \times D_n \to D$, where $D_1, \ldots, D_n$ are argument types of f and D is its result type. This data sort is then the set of expressions E of the following grammar:

Grammar 2.1.2 (Data expression)**.** The set of *data expressions* is given by the grammar

$$E ::= x \mid f(E_1, \ldots, E_n),$$

where expressions $E_1, \ldots, E_n$ match the data type of the arguments of function f.

Data expressions can only be given a definite semantics, i.e. evaluate to a value in the corresponding semantic domain, when all occurring data variables have a defined value. The value of data variables are given in a data environment:

Definition 2.1.3 (Data environment)**.** A *data environment* $\varepsilon : \mathsf{Dvar} \to \mathcal{D}$ is a partial function that assigns data values to data variables.

Let ε be an arbitrary data environment. A data expression e is called *closed in* ε, if and only if, all variables in e are assigned a value in ε.

Definition 2.1.4 (Data semantics)**.** Let e be a data expression of sort D, closed in data environment ε. The semantics of e in ε, denoted $[\![e]\!]\varepsilon$, is defined as follows:

$$\begin{aligned} [\![x]\!]\varepsilon &= \varepsilon(x) \\ [\![f(e_1, \ldots, e_n)]\!]\varepsilon &= f([\![e_1]\!]\varepsilon, \ldots, [\![e_n]\!]\varepsilon) \end{aligned}$$

Example 2.1.5 (Data expressions)**.** An example of a data sort is Nat, the sort of natural numbers. Its semantic domain is $\mathbb{N}$, the set of natural numbers.

Let $37 : \mathsf{Nat}$ be a constant, n and m be variables of sort Nat and let $+ : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ be a function. Examples of data expressions are n, $+(n, m)$ or $+(n, 37)$. Functions with two arguments are usually written in infix notation, i.e. $+(n, 37)$ is written as $n + 37$.

Furthermore, we assume that the sort $\mathbb{B}$ of booleans is predefined. This includes the usual constants true and false and the boolean connectives $\wedge$,$\vee$,$\neg$ and $\Rightarrow$.

We assume that data equivalence is defined for each data sort:

Definition 2.1.6 (Data equivalence semantics)**.** For each data sort D, a function $\approx$: $D \times D \to \mathbb{B}$ is assumed, with the following semantics:

$$[\![d \approx e]\!]\varepsilon \quad = \quad [\![d]\!]\varepsilon = [\![e]\!]\varepsilon$$

The (syntactic) definition of the $\approx$ operator depends on the data type and we assume this definition is supplied by the user.

2.2 Actions

Transitions in the processes described in mCRL2 are labeled with multi-actions, which is a number of parameterized actions occurring simultaneously. The following definitions are inspired by J. F. Groote [3]. We start with a finite set of *action names*:

Definition 2.2.1 (Action names)**.** We assume a finite set AN of *action names*, which are ranged over by $a, b, c, \ldots$.

We will use Latin lowercase letters to denote action names throughout this thesis. These action names can be parameterized with a data expression to form an *action*:

Definition 2.2.2 (Actions)**.** Let AN be a set of action names and $D_1, \ldots, D_n$ be data sorts. An *action* is an action name with data parameters: for any $a \in \mathsf{AN}$ and any $d_1 : D_1, \ldots, d_n : D_n$, $a(d_1, \ldots, d_n)$ is an action.

We often use the data sort D, without index, to abstract from the number of data parameters. We restrict ourselves to actions with constant signatures, i.e. where the data for each action is always of equal type.

Typically, we encounter multiple actions occurring simultaneously. For this, we introduce *multi-action names* and *multi-actions*, which are multisets of unparameterized and parameterized action names, respectively:

Definition 2.2.3 (Multi-action names)**.** A *multi-action name* is an expression denoting a collection of action names. The empty multi-action name is denoted τ. Let $a \in \mathsf{AN}$ be an action name and let A be a multi-action name, then $a|A$ is also a multi-action name.

For nonempty multi-action names, the trailing τ is left implicit.

Definition 2.2.4 (Multi-actions)**.** A *multi-action* is a collection of actions occurring simultaneously. Multi-actions are ranged over by $\alpha, \beta, \gamma, \ldots$. The set Act of multi-actions is defined as follows: the empty multi-action is denoted τ. Let $a(d)$ be an action and let $\alpha \in \mathsf{Act}$ be a multi-action, then $a(d)|\alpha$ is also a multi-action.

As with multi-action names, the trailing τ is left implicit for nonempty multi-actions. We will use Greek lowercase letters to denote multi-action names or multi-actions throughout this thesis, depending on the context. The expression $\alpha(d)$, where α is a multi-action name $a_1|\ldots|a_n$, d is of sort D and a_1 through a_n all take a parameter of type D, denotes the multi-action $a_1(d)|\ldots|a_n(d)$.

Note that this definition of multi-action is slightly different from the definition from J. F. Groote [3]. A multi-action is a *list* of actions with τ representing the empty list, not a binary tree with actions and τ as leaves, e.g. we exclude $\tau|\tau$. These definitions are essentially equivalent, but we remove all superfluous occurrences of τ.

The constructor $|$ will also be used in this thesis as an operator with a *multi*-action for its left operand. This is a straightforward generalization of the constructor function $|$ we described.

Example 2.2.5 (Actions)**.** Let the sort Nat from Example 2.1.5 be defined and let print be an action name. Let actions with action name print take one parameter of sort Nat. Furthermore, let n be a variable of sort Nat. Then, $\mathsf{print}(37)$, $\mathsf{print}(n)$ and $\mathsf{print}(n + 37)$ are actions. A multi-action name would be τ, print or $\mathsf{print}|\mathsf{print}$. A multi-action would be τ, $\mathsf{print}(37)$ or $\mathsf{print}(37)|\mathsf{print}(n)|\mathsf{print}(n+37)$. The expression $\mathsf{print}(\mathsf{true})$, where true is not an expression in Nat, is disallowed, as is $\mathsf{print}(37, 37)$, and print if it is to be interpreted as an action or multi-action. Note that τ can be both a multi-action name or a multi-action.

Suppose we abbreviate the multi-action name $\mathsf{print}|\mathsf{print}$ to α. The expression $\alpha(n)$ is then an abbreviation of the expression $\mathsf{print}(n)|\mathsf{print}(n)$, a multi-action.

Let stop be an action that takes no parameters. Then stop can be an action name, an action, a multi-action name or a multi-action. Likewise, the expression $\mathsf{stop}|\mathsf{stop}$ can be both a multi-action name and a multi-action.

2.2.1 Operators on actions

In the semantics of multi-actions, we define the equality, inclusion and removal operators by the following set of axioms from J. F. Groote [3]:

Definition 2.2.6 (Multi-action equality, inclusion and removal)**.** The equality relation is the strongest relation $=$ for which the following axioms hold:

MA1	$\alpha\|\beta = \beta\|\alpha$	
MA2	$(\alpha\|\beta)\|\gamma = \alpha\|(\beta\|\gamma)$	
MA3	$\alpha\|\tau = \alpha$	

For the inclusion relation $\sqsubseteq$ and the removal operator $\setminus$, the following axioms hold:

MS1	$\tau \sqsubseteq \alpha = \mathsf{true}$	
MS2	$a(d)\|\alpha \sqsubseteq \tau = \mathsf{false}$	
MS3	$a(d)\|\alpha \sqsubseteq a(e)\|\beta = \alpha \sqsubseteq \beta$	, if $d \approx e$
MS4	$a(d)\|\alpha \sqsubseteq b(e)\|\beta = a(d)\|(\alpha \setminus b(e)) \sqsubseteq \beta$	, if $a \not\equiv b$ or $d \not\approx e$

MD1	$\alpha \setminus \tau = \alpha$	
MD2	$\tau \setminus \alpha = \tau$	
MD3	$\alpha \setminus (\beta\|\gamma) = (\alpha \setminus \beta) \setminus \gamma$	
MD4	$(a(d)\|\alpha) \setminus a(e) = \alpha$	, if $d \approx e$
MD5	$(a(d)\|\alpha) \setminus b(e) = a(d)\|(\alpha \setminus b(e))$	, if $a \not\equiv b$ or $d \not\approx e$

Here, $\equiv$ denotes syntactic equality.

We define a function from multi-actions to multi-action names, which simply removes all data parameters:

Definition 2.2.7 (Data removal from multi-actions)**.** We define $\underline{\alpha}$ as the multi-action name obtained by removing all data from the multi-action α:

$$\underline{\tau} = \tau$$
$$\underline{a(d)|\alpha} = a|\underline{\alpha}$$

The length of a multi-action name or a multi-action α, denoted $|\alpha|$, is the number of action names occurring in it:

Definition 2.2.8 (Length of multi-actions)**.** We define $|\alpha|$ as the number of action names in multi-action α:

$$|\tau| = 0$$
$$|a(d)|\alpha| = 1 + |\alpha|$$

The removal, inclusion, equality and length operators are generalized to multi-action names in the obvious way.

To make multi-action equality, inclusion and removal useful for syntactic descriptions of multi-actions, we define them in the recursion scheme implied by the definition of multi-actions. It can easily be verified that the axioms of Definition 2.2.6 (Multi-action equality, inclusion and removal) are valid for these constructive definitions, i.e. that they are actually the intended operator.

The following three definitions can be skipped if the reader is familiar with multi-action equality, inclusion and removal. The main point is to show that these relations can be expressed as a boolean data expression.

Definition 2.2.9 (Constructive removal)**.** The removal operator $\setminus : \mathsf{MA} \times \mathsf{MA} \to \mathsf{MA}$ removes the second multi-action from the first. It is defined on the multi-action syntax as follows:

$$\begin{aligned} \alpha \setminus \tau &= \alpha \\ \tau \setminus \beta &= \tau \\ a(d)|\alpha \setminus b(e)|\beta &= \begin{cases} \alpha \setminus \beta & \text{if } a \equiv b \wedge d \approx e \\ a(d)|(\alpha \setminus b(e)) \setminus \beta & \text{otherwise} \end{cases} \end{aligned}$$

Using the removal operator, we can define inclusion of one multi-action in another:

Definition 2.2.10 (Constructive inclusion)**.** The inclusion operator $\sqsubseteq: \mathsf{MA} \times \mathsf{MA} \to \mathbb{B}$ checks if the first multi-action is contained in the second. It is defined on the multi-action syntax as follows:

$$\begin{aligned} \alpha \sqsubseteq \tau &= \alpha \equiv \tau \\ \tau \sqsubseteq \beta &= \mathsf{true} \\ a(d)|\alpha \sqsubseteq b(e)|\beta &= \begin{cases} \alpha \sqsubseteq \beta & \text{if } a \equiv b \wedge d \approx e \\ \alpha \sqsubseteq b(e)|(\beta \setminus a(d)) \wedge a(d) \sqsubseteq \beta & \text{otherwise} \end{cases} \end{aligned}$$

We can now define equality as follows:

Definition 2.2.11 (Constructive equality)**.** The equality operator $=: \mathsf{MA} \times \mathsf{MA} \to \mathbb{B}$ checks if the first multi-action equals the second. It is defined on the multi-action syntax as follows:

$$\begin{aligned} \tau = \beta &= \beta \equiv \tau \\ a(d)|\alpha = \beta &= \alpha = \beta \setminus a(d) \wedge a(d) \sqsubseteq \beta \end{aligned}$$

An alternative characterization of equality on multi-actions is that both multi-actions are included in the other.

2.2.2 Action formulae

We denote sets of multi-actions using action formulae.

Grammar 2.2.12 (Action formulae)**.** The *action formulae* are given by the following grammar:

$$\begin{aligned} AF ::= & \; b \\ & \mid \alpha \\ & \mid \neg(AF) \\ & \mid AF \wedge AF \\ & \mid AF \vee AF \end{aligned}$$

Here b is a boolean expression.

The semantics of action formulae are as follows:

Definition 2.2.13 (Action formulae semantics)**.** The semantics of an action formula af in a data environment ε, denoted $[\![af]\!]\varepsilon$, is a set of multi-actions, given by:

$$
\begin{aligned}
[\![b]\!]\varepsilon &= \begin{cases} \mathsf{Act} & \text{if } [\![b]\!]\varepsilon \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\alpha(e)]\!]\varepsilon &= \{\alpha([\![e]\!]\varepsilon)\} \\
[\![\neg(af)]\!]\varepsilon &= \mathsf{Act} \setminus [\![af]\!]\varepsilon \\
[\![af \wedge af']\!]\varepsilon &= [\![af]\!]\varepsilon \cap [\![af']\!]\varepsilon \\
[\![af \vee af']\!]\varepsilon &= [\![af]\!]\varepsilon \cup [\![af']\!]\varepsilon
\end{aligned}
$$

Note that in the rule for b, the occurrence of $[\![b]\!]\varepsilon$ on the left hand side denotes the action formula's semantics, i.e. a set of states, while that on the right hand side denotes the boolean expression's semantics, i.e. true or false. The set Act is the set of all multi-actions (cf. Definition 2.2.4 (Multi-actions)).

Clearly, all action formulae can be rewritten to a form where negations only occur immediately before multi-actions. This can then be rewritten to disjunctive normal form, analogous to disjunctive normal form for boolean formulae.

Definition 2.2.14 (Action formulae disjunctive normal form)**.** The action formulae in *disjunctive normal form* are given by the following grammar:

$$
\begin{aligned}
AF &::= C \mid C \vee C \\
C &::= L \mid L \wedge L \\
L &::= b \mid \alpha \mid \neg\alpha
\end{aligned}
$$

Here, b is a boolean expression and α is a multi-action. The disjuncts C are called *clauses* and the conjuncts L are called *literals*.

2.3 Labeled transition systems

Labeled transition systems are the semantic basis of process descriptions in mCRL2. A labeled transition system has an explicit set of states S, an explicit transition relation and an initial state. The labeled transition systems are restricted to process descriptions of finite state spaces.

Formally, we define the labeled transition systems as follows:

Definition 2.3.1 (Labeled transition systems)**.** Let Act be a set of multi-actions. A *labeled transition system* t is a triple $(S_t, \rightarrow_t, i_t)$, where S_t is a set of states, $\rightarrow_t \subseteq S_t \times \mathsf{Act} \times S_t$ is a transition relation between states and $i_t \in S_t$ is the initial state.

We usually write $s \xrightarrow{a}_t s'$ instead of $(s, a, s') \in \rightarrow_t$. We often also drop the subscript t if it is clear from the context.

Labeled transition systems are visually represent labeled transition systems with dots indicating states and labeled arrows between states indicating transitions between them.

2.4 Linear process equations

Linear process equations are a subset of the process algebra mCRL2 (Groote *et al.* [2]). We use this as our most basic description of a process, because it is sufficiently expressive for real-world systems, while also being easy to understand. Also, it restricts our process descriptions to guarded processes, disallowing such troublesome processes as "the process X equals two processes X in parallel": $X = X \,||\, X$.

Linear process equations are models described as lists of condition-action-effect clauses. The clauses are called *summands*, identified by *summand indices*. The three elements of a condition-action-effect clause are:

- **Condition**: there is a boolean condition, an expression on data, that tells us when the transition is enabled. If multiple transitions are enabled, one is chosen nondeterministically.
- **Action**: this multi-action expression is the transition label.
- **Effect**: the resulting state after taking the transition.

We formally define linear process equations as follows (Groote and Willemse [5]):

Definition 2.4.1 (Linear process equations)**.** Let S be a data sort corresponding to the state space $\mathcal{S}$ of a process. Let I be a set of summand indices, with for each summand index i a data sort E_i. Let for each state s and data value $e : E_i$, the condition $c_i(s, e) : \mathbb{B}$ specify if a transition is possible, $\alpha_i(s, e) \in \mathsf{Act}$ denote the label of that transition and $g_i(s, e) \in S$ denote the resulting state of the process. A *linear process equation* is a description of a system in the following form:

$$P(s : S) = \sum_{i \in I} \sum_{e:E_i} c_i(s, e) \to \alpha_i(s, e) \;.\; P(g_i(s, e))$$

We omit the summation symbol for summations over a single term, i.e. when I or the semantic domain of E_i are singletons or when c_i, α_i and g_i are independent from e. We restrict ourselves to action labels $\alpha_i(s, e) = a_{i,1}(e_{i,1})|\ldots|a_{i,n_i}(e_{i,n_i})$, where $a_{i,1}$ through a_{i,n_i} are action names and $e_{i,1}$ through e_{i,n_i} are expressions of the correct sort for that action name.

The standard process P, given in Definition 2.4.1 (Linear process equations), will be used throughout this thesis.

The semantics of a linear process equation is a labeled transition system:

Definition 2.4.2 (Linear process equation semantics)**.** The semantics of P with initial state $s_i \in \mathcal{S}$, is the labeled transition system $[\![P(s_i)]\!]$, defined as follows:

- $S_{[\![P(s_i)]\!]} = \mathcal{S}$
- $\to_{[\![P(s_i)]\!]} = \{(s, \alpha_i(s, e), g_i(s, e)) \mid i \in I \wedge e : E_i \wedge c_i(s, e)\}$
- $i_{[\![P(s_i)]\!]} = s_i$

We also use a notational variant where the definition of a process occurs within the semantic brackets $[\![\cdot]\!](s_i)$, meaning "$[\![P(s_i)]\!]$, where P is defined as…".

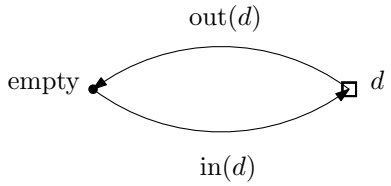


Figure 1: A one-place buffer. The left state represents the buffer being empty. The square symbol to the right represents a possibly infinite number of states where the buffer is full, one for each element in D, denoted by the variable d. Similarly, there are possibly infinitely many transitions: an in and out transition for each element in D.

2.5 Buffer example (1)

As a running example, we consider a buffer process. A buffer has a number of storage locations. Data elements can be put in the buffer with the "in" action, provided it is not already full. If there are data elements in the buffer, the "out" action outputs these elements, in the order in which they entered. In this first example, we look at a one-place buffer.

A one-place buffer has one state for each of the possible data elements it can hold and one extra state "empty" for when it is empty. If it holds a data element s, it can output that data element with a $\mathsf{out}(s)$ action, ending up in the state empty. If it is empty, it can input any data element d with a $\mathsf{in}(d)$ action, ending up in the state where it is holding d. This is illustrated in Figure 1. The one-place buffer cannot be expressed as a finite labeled transition system when the data domain D and thus the state space $D \cup \{\mathsf{empty}\}$ is infinitely large. We express this process as a linear process equation.

Let the linear process equation B_1 represent the one-place buffer. The state parameter is a variable s from $D \cup \{\mathsf{empty}\}$. There are two summands: one for when $s \approx \mathsf{empty}$ and one for when $s \not\approx \mathsf{empty}$. If it is empty, there is a possible in transition for any data element from D, expressed as a summation over $d : D$. The resulting process is $B_1(d)$. If the buffer is not empty, it holds a data element s. The out transition is enabled for s and the resulting process is $B_1(\mathsf{empty})$.

Let the one-place buffer B_1 be defined as follows:

$$B_1(s : D \cup \{\mathsf{empty}\}) = \sum_{d:D} s \approx \mathsf{empty} \to \mathsf{in}(d) \;.\; B_1(d) \\ + s \not\approx \mathsf{empty} \to \mathsf{out}(s) \;.\; B_1(\mathsf{empty})$$

2.6 Process descriptions

In this section, we define operators on linear process equations. These operators are defined on labeled transition systems by J. F. Groote [3] and we lift these definitions to linear process equations. There are five process operators we consider. There is the binary parallel composition operator $||$ and four unary process operators: the communication operator Γ, the allow operator ∇, the rename operator ρ and the abstraction operator τ.

The combination of linear process equations and process operators working on them, form the set of *process descriptions*:

Grammar 2.6.1 (Process descriptions)**.** A *process description* is an expression of the following grammar, where P is a linear process equation and Q is a process description:

$$Q ::= P(s) \mid Q \,||\, Q \mid \Gamma_C(Q) \mid \nabla_V(Q) \mid \rho_C(Q) \mid \tau_H(Q)$$

Here, C is a set of substitutions (cf. Definition 2.8.1 (Substitution function)) and V and H are sets of multi-action names.

The remainder of this section explains these operators in detail. For each operator, we give the following:

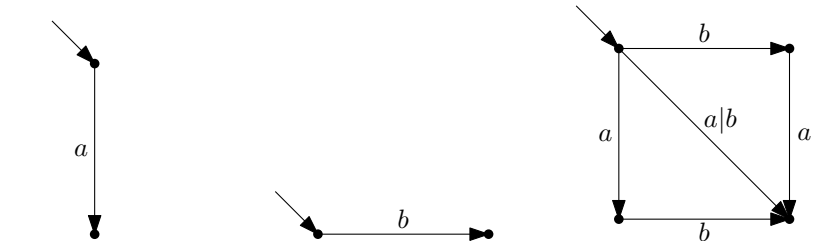
- an explanation of the intuition,
- a formal version of the definition on labeled transition systems, inspired by or taken from J. F. Groote [3],
- an example of the operator on labeled transition systems,
- a derivation of the definition on linear process equations, using the definition on labeled transition systems,
- an example of the operator on linear process equations.

2.7 Parallel composition

The parallel composition between two processes can do a transition from either of the two processes, or a transition in both processes simultaneously. Actions occurring simultaneously will be denoted using multi-actions. There is no implicit communication between those processes. Any communication between processes will be made explicit using the communication operator, which we will see in Section 2.8. The parallel composition on labeled transition systems is defined as follows:

Definition 2.7.1 (Parallel composition on labeled transition systems)**.** The *parallel composition* of two labeled transition systems t_1 and t_2, is the labeled transition system $t_1 \,||\, t_2$ where:

- $S_{t_1 \,||\, t_2} = S_{t_1} \times S_{t_2}$
- $\begin{aligned}\rightarrow_{t_1 \,||\, t_2} = {} & \{((s_1, s_2), \alpha, (s_1', s_2)) \in (S_{t_1} \times S_{t_2}) \times \mathsf{Act} \times (S_{t_1} \times S_{t_2}) \mid s_1 \xrightarrow{\alpha}_{t_1} s_1'\} \cup \\ & \{((s_1, s_2), \beta, (s_1, s_2')) \in (S_{t_1} \times S_{t_2}) \times \mathsf{Act} \times (S_{t_1} \times S_{t_2}) \mid s_2 \xrightarrow{\beta}_{t_2} s_2'\} \cup \\ & \{((s_1, s_2), \alpha|\beta, (s_1', s_2')) \in (S_{t_1} \times S_{t_2}) \times \mathsf{Act} \times (S_{t_1} \times S_{t_2}) \mid s_1 \xrightarrow{\alpha}_{t_1} s_1' \wedge s_2 \xrightarrow{\beta}_{t_2} s_2'\}\end{aligned}$
- $i_{t_1 \,||\, t_2} = (i_{t_1}, i_{t_2})$



(a) System t_1 (b) System t_2 (c) System $t_1 \,||\, t_2$

Figure 2: Example of the parallel composition operator.

Example 2.7.2 (Parallel composition)**.** An example of the parallel composition operator on labeled transition systems, is shown in Figure 2. Given are two systems, t_1 and t_2, their parallel composition $t_1 \,||\, t_2$ is shown on the right.

The following properties on the parallel composition are obvious:

Fact 2.7.3 (Parallel composition properties)**.** Up to state isomorphism and multi-action equality on transition labels, parallel composition on labeled transition systems is associative, commutative, and the system $t_\delta = (\{s\}, \emptyset, s)$ is its unit element.

Next, we derive the parallel composition on linear process equations from the specification that it corresponds to parallel composition on their semantic labeled transition systems:

$$[\![(P_1 \,||\, P_2)(s_{i1}, s_{i2})]\!] = [\![P_1(s_{i1})]\!] \,||\, [\![P_2(s_{i2})]\!]$$

Let P_1 and P_2 be the following linear process equations:

$$P_1(s : S_1) = \sum_{i \in I_1} \sum_{e : E_{1,i}} c_{1,i}(s, e) \to \alpha_{1,i}(s, e) \;.\; P_1(g_{1,i}(s, e))$$

$$P_2(s : S_2) = \sum_{i \in I_2} \sum_{e : E_{2,i}} c_{2,i}(s, e) \to \alpha_{2,i}(s, e) \;.\; P_2(g_{2,i}(s, e))$$

We derive the parallel composition as follows:

$$\begin{aligned}
&[\![P_1(s_{i1})]\!] \,||\, [\![P_2(s_{i2})]\!] \\
&= \{\text{Definition 2.4.2 (Linear process equation semantics)}\} \\
&\quad (\mathcal{S}_1, \{(s, \alpha_{1,i}(s, e), g_{1,i}(s, e)) \mid s \in \mathcal{S}_1 \wedge i \in I_1 \wedge c_{1,i}(s_1, e)\}, s_{i1}) \\
&\quad \,||\, (\mathcal{S}_2, \{(s, \alpha_{2,i}(s, e), g_{2,i}(s, e)) \mid s \in \mathcal{S}_2 \wedge i \in I_2 \wedge c_{2,i}(s, e)\}, s_{i2}) \\
&= \{\text{Definition 2.7.1 (Parallel composition on labeled transition systems)}\} \\
&\quad (\mathcal{S}_1 \times \mathcal{S}_2, \\
&\quad \{((s_1, s_2), \alpha_{1,i}(s_1, e), (g_{1,i}(s_1, e), s_2)) \mid s_1 \in \mathcal{S}_1 \wedge s_2 \in \mathcal{S}_2 \wedge i \in I_1 \wedge e : E_{1,i} \wedge c_{1,i}(s_1, e)\} \\
&\quad \cup \{((s_1, s_2), \alpha_{2,i}(s_2, e), (s_1, g_{2,i}(s_2, e))) \mid s_1 \in \mathcal{S}_1 \wedge s_2 \in \mathcal{S}_2 \wedge i \in I_2 \wedge e : E_{2,i} \wedge c_{2,i}(s_2, e)\} \\
&\quad \cup \{((s_1, s_2), \alpha_{1,i_1}(s_1, e_1) | \alpha_{2,i_2}(s_2, e_2), (g_{1,i_1}(s_1, e_1), g_{2,i}(s_2, e_2))) \mid s_1 \in \mathcal{S}_1 \wedge s_2 \in \mathcal{S}_2 \\
&\qquad \wedge i_1 \in I_1 \wedge i_2 \in I_2 \wedge e_1 : E_{1,i_1} \wedge e_2 : E_{2,i_2} \wedge c_{1,i_1}(s_1, e_1) \wedge c_{2,i_2}(s_2, e_2)\}, \\
&\quad (s_{i1}, s_{i2}))
\end{aligned}$$

$= \{$Definition 2.4.2 (Linear process equation semantics)$\}$

$$\left[\!\!\left[\begin{aligned} P(s_1 : S_1, s_2 : S_2) \;=\; & \sum_{i \in I_1} \sum_{e : E_{1,i}} c_{1,i}(s_1, e) \to \alpha_{1,i}(s_1, e) \;.\; P(g_{1,i}(s_1, e), s_2) \\ & + \sum_{i \in I_2} \sum_{e : E_{2,i}} c_{2,i}(s_2, e) \to \alpha_{2,i}(s_2, e) \;.\; P(s_1, g_{2,i}(s_2, e)) \\ & + \sum_{i_1 \in I_1} \sum_{i_2 \in I_2} \sum_{e_1 : E_{1,i_1}} \sum_{e_2 : E_{2,i_2}} c_{1,i_1}(s_1, e_1) \wedge c_{2,i_2}(s_2, e_2) \to \\ & \qquad \alpha_{1,i_1}(s_1, e_1) | \alpha_{2,i_2}(s_2, e_2) \;.\; P(g_{1,i_1}(s_1, e_1), g_{2,i_2}(s_2, e_2)) \end{aligned} \right]\!\!\right] (s_{i1}, s_{i2})$$

So, we define parallel composition on linear process equations as follows:

Definition 2.7.4 (Parallel composition on linear process equations)**.** Let P_1 and P_2 be the following linear process equations:

$$P_1(s : S_1) = \sum_{i \in I_1} \sum_{e : E_{1,i}} c_{1,i}(s, e) \to \alpha_{1,i}(s, e) \;.\; P_1(g_{1,i}(s, e))$$

$$P_2(s : S_2) = \sum_{i \in I_2} \sum_{e : E_{2,i}} c_{2,i}(s, e) \to \alpha_{2,i}(s, e) \;.\; P_2(g_{2,i}(s, e))$$

The parallel composition of linear process equations P_1 and P_2, denoted $P_1 \,||\, P_2$, is defined as follows:

$$\begin{aligned} & (P_1 \,||\, P_2)(s_1 : S_1, s_2 : S_2) = \\ & \quad \sum_{i \in I_1} \sum_{e : E_{1,i}} c_{1,i}(s_1, e) \to \alpha_{1,i}(s_1, e) \;.\; (P_1 \,||\, P_2)(g_{1,i}(s_1, e), s_2) \\ & \quad + \sum_{i \in I_2} \sum_{e : E_{2,i}} c_{2,i}(s_2, e) \to \alpha_{2,i}(s_2, e) \;.\; (P_1 \,||\, P_2)(s_1, g_{2,i}(s_2, e)) \\ & \quad + \sum_{i_1 \in I_1} \sum_{i_2 \in I_2} \sum_{e_1 : E_{1,i_1}} \sum_{e_2 : E_{2,i_2}} c_{1,i_1}(s_1, e_1) \wedge c_{2,i_2}(s_2, e_2) \to \\ & \qquad \alpha_{1,i_1}(s_1, e_1) | \alpha_{2,i_2}(s_2, e_2) \;.\; (P_1 \,||\, P_2)(g_{1,i_1}(s_1, e_1), g_{2,i_2}(s_2, e_2)) \end{aligned}$$

We assume the variables s_1 and s_2 are distinct. This can always be achieved by renaming variables.

Lemma 2.7.5 (Parallel composition correspondence)**.** The parallel composition on linear process equations corresponds to the parallel composition on labeled transition systems, i.e.

$$[\![(P_1 \,||\, P_2)(s_{i1}, s_{i2})]\!] = [\![P_1(s_{i1})]\!] \,||\, [\![P_2(s_{i2})]\!].$$

Proof. By the derivation of parallel composition and Definition 2.7.4 (Parallel composition on linear process equations). □

As a corollary, the properties of Fact 2.7.3 (Parallel composition properties) also apply to parallel composition on linear process equations:

Corollary 2.7.6 (Parallel composition properties on linear process equations)**.** From Fact 2.7.3 (Parallel composition properties) and Lemma 2.7.5 (Parallel composition correspondence) it follows that parallel composition on linear process equations is associative and commutative with respect to state isomorphism and multi-action equality on transition labels. Let the linear process equation P_δ be defined as follows:

$$P_\delta(s : \{i_{P_\delta}\}) = \mathsf{false} \to \tau \;.\; P_\delta(s)$$

As $[\![P_\delta(i_{P_\delta})]\!] = t_\delta$, it is the unit element of parallel composition on linear process equations, with respect to state isomorphism.

Example 2.7.7 (Example 2.7.2 as linear process equations)**.** The process of Example 2.7.2 (Parallel composition) can be expressed in linear process equations as follows: the left labeled transition system is represented by linear process equation $P_1(\mathsf{true})$, and the right by $P_2(\mathsf{true})$.

$$\begin{aligned}
P_1(c : \mathbb{B}) &= c \to a \;.\; P_1(\neg c)\\
P_2(d : \mathbb{B}) &= d \to b \;.\; P_2(\neg d)\\
(P_1 \,||\, P_2)(c : \mathbb{B}, d : \mathbb{B}) &= c \to a \;.\; (P_1 \,||\, P_2)(\neg c, d)\\
&\quad + \;\; d \to b \;.\; (P_1 \,||\, P_2)(c, \neg d)\\
&\quad + \;\; c \wedge d \to a|b \;.\; (P_1 \,||\, P_2)(\neg c, \neg d)
\end{aligned}$$

2.8 Communication operator

The communication operator connects parallel components together so they can interact. A communication operator performs a number of substitutions of action names, provided their data arguments match.

What actions are to be substituted, is denoted by a set of substitutions. This is an argument of the communication operator, which is then applied to a process. How the action names are substituted, is given by the substitution function:

Definition 2.8.1 (Substitution function)**.** Let $a_1, \ldots, a_n$ and $a'_1, \ldots, a'_m$ be action names. A *substitution* is an expression of the form $a_1|\ldots|a_n \to a'_1|\ldots|a'_m$, commonly abbreviated as $\beta \to \beta'$. The *substitution function* γ substitutes multi-actions for other actions, if their data arguments match. Let α be a multi-action. We use $\beta(d)$, where $\beta = a_1|\ldots|a_n$, to mean $a_1(d)|\ldots|a_n(d)$

$$\begin{aligned}
\gamma_\emptyset(\alpha) &= \alpha\\
\gamma_{C_1 \cup C_2}(\alpha) &= \gamma_{C_1}(\gamma_{C_2}(\alpha))\\
\gamma_{\{\beta \to \beta'\}}(\alpha) &= \begin{cases} \beta'(d)|\gamma_{\{\beta \to \beta'\}}(\alpha \setminus \beta(d)) & \text{if } \beta(d) \sqsubseteq \alpha \text{ for some } d \in \mathcal{D}\\ \alpha & \text{if no such } d \text{ exists} \end{cases}
\end{aligned}$$

The substitution set must yield a *confluent* substitution function: the order in which the substitutions are applied does not influence the result. Formally, we disallow sets of substitutions for which the second rule in Definition 2.8.1 (Substitution function) yields different results for different partitionings of C_1, C_2, i.e. it must hold that $\gamma_{C_1}(\gamma_{C_2}(\alpha)) = \gamma_{C_3}(\gamma_{C_4}(\alpha))$, for any α and for any C_3, C_4 such that $C_1 \cup C_2 = C_3 \cup C_4$.

Example 2.8.2 (Substitution function)**.** Let $C = \{a|b \to c\}$. When $a|b$ is contained in a multi-action α, and their data parameters are equal, it is substituted with c. If it occurs multiple times, it is substituted multiple times: $\gamma_C(a|a|b|b) = c|c$. The result of a substitution is not substituted again: $\gamma_{\{a|a\to a\}}(a|a|a|a) = a|a$, not a. It is therefore not generally the case that $\beta(d) \not\sqsubseteq \gamma_{\{\beta\to\beta'\}}(\alpha)$ for any $d \in \mathcal{D}$.

The data parameter requirement is trivially fulfilled when there are none, i.e. $\gamma_C(a|b) = c$ in any environment ε. When there are data parameters, they must match to be substituted. The data parameter of the result is equal to the data parameters of the original. For example, $\gamma_C(a(1)|b(1)) = c(1)$.

These data parameters can also be variables and then the result of the substitution function depends on the data environment: $\gamma_C(a(x)|b(y) = c(x) = c(y)$ only when $[\![x \approx y]\!]\varepsilon$ equals true. Note that x, y or any other equal data expression can be chosen as the result parameter, as they are all semantically equal.

Data parameters also trivially match when there is only one to match, e.g. for sets of substitutions $a \to \beta$. This is called a single substitution and the communication operator for single substitutions is called the rename operator. It is discussed in more detail in Section 2.10.

We disallow the substitution set $\{a \to b, b \to c\}$, as it matters which substitution occurs first. For example, $\gamma_{\{a\to b\}}(\gamma_{\{b\to c\}}(a)) = \gamma_{\{a\to b\}}(a) = b$ is not equal to $\gamma_{\{b\to c\}}(\gamma_{\{a\to b\}}(a)) = \gamma_{\{b\to c\}}(b) = c$.

The communication operator for substitutions C applied to a labeled transition system t is equal to t with the substitution function γ_C applied to all transition labels:

Definition 2.8.3 (Communication operator)**.** The *communication operator* for substitutions C applied to labeled transition system t, denoted $\Gamma_C(t)$, is defined as follows:

- $S_{\Gamma_C(t)} = S_t$
- $\to_{\Gamma_C(t)} = \{(s, \gamma_C(\alpha), s') \in S_t \times \mathsf{Act} \times S_t \mid s \xrightarrow{\alpha}_t s'\}$
- $i_{\Gamma_C(t)} = i_t$

We disallow substitutions where the actions on the left and right hand side take expressions of a different sort as parameter.

We derive the communication operator on linear process equations from the communication operator on the corresponding labeled transition system:

$$[\![\Gamma_C(P)(s_i)]\!] = \Gamma_C([\![P(s_i)]\!])$$

The derivation is as follows:

$$\begin{aligned}
&\Gamma_C([\![P(s_i)]\!]) \\
&= \{\text{Definition 2.4.2 (Linear process equation semantics)}\} \\
&\quad \Gamma_C((\mathcal{S}, \{(s, \alpha_i(s,e), g_i(s,e)) \mid s \in S \wedge i \in I \wedge e : E_i \wedge c_i(s,e)\}, s_i)) \\
&= \{\text{Definition 2.8.3 (Communication operator)}\} \\
&\quad (\mathcal{S}, \{(s, \gamma_C(\alpha_i(s,e)), g_i(s,e)) \mid s \in S \wedge i \in I \wedge e : E_i \wedge c_i(s,e)\}, s_i)
\end{aligned}$$

We may assume C only has one substitution $\beta \to \beta'$, which we can always establish using the rule $\gamma_{C_1 \cup C_2}(\alpha) = \gamma_{C_1}(\gamma_{C_2}(\alpha))$.

Following the definition of the substitution function, we want to see if $\beta(d) \sqsubseteq \alpha_i(s,e)$ for some $d \in \mathcal{D}$. We split up the transition labels *syntactically* into pairs (α_1, α_2) and check if $\beta(d) = \alpha_1$ for some $d \in \mathcal{D}$. If so, we recursively check if there are some substitutions to perform in α_2. Because $\beta(d) = \alpha_1$ can only hold if β and α_1 are of equal length, we confine our search to submulti-actions of length $|\beta|$. We define the function choose_m on multi-actions α to give a set of all multi-actions contained in α of length m, along with the remainder of α:

$$\mathsf{choose}_m(\tau) = \begin{cases} \{(\tau,\tau)\} & \text{if } m = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{choose}_m(a(e)|\alpha) = \begin{cases} \{(a(e)|\alpha_1, \alpha_2) \mid (\alpha_1,\alpha_2) \in \mathsf{choose}_{m-1}(\alpha)\} & \\ \quad \cup \{(\alpha_1, a(e)|\alpha_2) \mid (\alpha_1,\alpha_2) \in \mathsf{choose}_m(\alpha)\} & \text{if } m > |\alpha| + 1 \\ \{(a(e)|\alpha, \tau)\} & \text{if } m = |\alpha| + 1 \\ \emptyset & \text{otherwise} \end{cases}$$

Let $(\alpha_1, \alpha_2) \in \mathsf{choose}_{|\beta|}(\alpha)$. We need to check if $\beta(d) = \alpha_1$ for some $d \in \mathcal{D}$. Let $a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}) = \alpha_1$. We can immediately determine whether $\beta = a_1|\ldots|a_{|\beta|}$. If not, $\beta(d) \neq \alpha_1$ for any $d \in \mathcal{D}$. Otherwise, we have to see if $e_1 \approx \ldots \approx e_{|\beta|}$. We construct a list of condition-action-effect clauses using the function CAE:

$$\mathsf{CAE}_{P,\beta,\beta',g}(\alpha,\gamma,c) =$$

$$c \wedge \bigwedge_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \neg(e_1 \approx \ldots \approx e_{|\beta|}) \to \alpha|\gamma \;.\; P(g)$$

$$+ \sum_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \mathsf{CAE}_{P,\beta,\beta',g}(\alpha', \gamma|\beta'(e_1), c \wedge e_1 \approx \ldots \approx e_{|\beta|})$$

This function returns a list of condition-action-effect clauses, corresponding to the substitution function being applied to the multi-action α. The multi-action γ accumulates the results of substitutions so that they will not be substituted again. The conditions for when a substitution occurs, are accumulated in the condition c.

The multi-action α is split into $a_1(e_1)|\ldots|a_{|\beta|}$ and α' by the choose function. There is one summand for when there is no substitution, and one summand for each possible split in which there is a substitution. In the first summand, $a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|})$ is not equal to $\beta(d)$ for any possible split and any data element d. No substitution takes place. In the other summands, $a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|})$ is substituted to $\beta'(e_1)$ which is appended to the substitution results γ. Also, the condition for the substitution to be successful, $e_1 \approx \ldots \approx e_{|\beta|}$, is appended to the condition c.

The recursion ends when there are no more submulti-actions in α that could be affected by the substitution function, i.e. the choose function yields no candidates for substitution.

So, we define the communication operator on linear process equations as follows:

Definition 2.8.4 (Communication operator on linear process equations)**.** Let C be a set of substitutions and let process P be the standard process. The communication operator of C applied to process P, denoted $\Gamma_C(P)$, is again a linear process equation, defined as follows:

If C is not a singleton:

$$\Gamma_{C_1 \cup C_2}(P) = \Gamma_{C_1}(\Gamma_{C_2}(P))$$

If C is a singleton $\beta \to \beta'$:

$$\Gamma_{\{\beta \to \beta'\}}(P) = \sum_{i \in I} \sum_{e:E_i} \mathsf{CAE}_{\Gamma_{\{\beta \to \beta'\}}(P),\beta,\beta',g_i(s,e)}(\alpha_i(s,e), \tau, c_i(s,e)),$$

where CAE is defined as:

$$\begin{aligned}
&\mathsf{CAE}_{P,\beta,\beta',g}(\alpha,\gamma,c) = \\
&\quad c \wedge \bigwedge_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \neg(e_1 \approx \ldots \approx e_{|\beta|}) \to \alpha|\gamma \;.\; P(g) \\
&\quad + \sum_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \mathsf{CAE}_{P,\beta,\beta',g}(\alpha', \gamma|\beta'(e_1), c \wedge e_1 \approx \ldots \approx e_{|\beta|}),
\end{aligned}$$

and choose is defined as:

$$\mathsf{choose}_m(\tau) = \begin{cases} \{(\tau,\tau)\} & \text{if } m = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{choose}_m(a(e)|\alpha) = \begin{cases} \{(a(e)|\alpha_1,\alpha_2) \mid (\alpha_1,\alpha_2) \in \mathsf{choose}_{m-1}(\alpha)\} & \\ \quad \cup \{(\alpha_1, a(e)|\alpha_2) \mid (\alpha_1,\alpha_2) \in \mathsf{choose}_m(\alpha)\} & \text{if } m > |\alpha| + 1 \\ \{(a(e)|\alpha,\tau)\} & \text{if } m = |\alpha| + 1 \\ \emptyset & \text{otherwise} \end{cases}$$

Lemma 2.8.5 (Communication operator correspondence)**.** The communication operator on linear process equations corresponds to the communication operator on its semantic labeled transition system, i.e. it holds that $[\![\Gamma_C(P)(s_i)]\!] = \Gamma_C([\![P(s_i)]\!])$.

Proof. By the derivation and definition of the communication operator on linear process equations, Definition 2.8.4 (Communication operator on linear process equations). □

Example 2.8.6 (Communication operator on linear process equations)**.** Consider the following processes P and Q:

$$P = \sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \;.\; P$$

$$Q = \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; Q$$

One repeatedly transmits data and the other repeatedly receives data. Their parallel composition is the following process $P \,||\, Q$:

$$\begin{aligned}
(P \,||\, Q) = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \;.\; (P \,||\, Q) \\
&+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; (P \,||\, Q) \\
&+ \sum_{d:D} \sum_{d':D} \mathsf{true} \to \mathsf{send}(d)|\mathsf{receive}(d') \;.\; (P \,||\, Q)
\end{aligned}$$

A successful data transfer from P to Q is when P sends data which is received by Q. This is the action $\mathsf{transfer}$. We let the actions send and $\mathsf{receive}$ communicate to action $\mathsf{transfer}$, by applying $\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}}$ to $P \,||\, Q$. We abbreviate $\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}}(P \,||\, Q)$ as R:

$$\begin{aligned} R = &\sum_{d:D} \mathsf{CAE}_{R,\mathsf{send}|\mathsf{receive},\mathsf{transfer},\mathsf{id}}(\mathsf{send}(d), \tau, \mathsf{true}) \\ &+ \sum_{d:D} \mathsf{CAE}_{R,\mathsf{send}|\mathsf{receive},\mathsf{transfer},\mathsf{id}}(\mathsf{receive}(d), \tau, \mathsf{true}) \\ &+ \sum_{d:D} \sum_{d':D} \mathsf{CAE}_{R,\mathsf{send}|\mathsf{receive},\mathsf{transfer},\mathsf{id}}(\mathsf{send}(d)|\mathsf{receive}(d'), \tau, \mathsf{true}) \end{aligned}$$

As choose_2 for $\mathsf{send}(d)$ and $\mathsf{receive}(d)$ yields the empty set, the recursion ends immediately for the first two clauses. For the third clause, we calculate that $\mathsf{choose}_2(\mathsf{send}(d)|\mathsf{receive}(d'))$ equals $\{(\mathsf{send}(d)|\mathsf{receive}(d'), \tau)\}$. This yields two clauses: one for the case in which nothing communicates to a $\mathsf{transfer}$ action, and one for each split (there is only one) with a recursive application of CAE when it does communicate.

$$\begin{aligned} R = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \,.\, R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \,.\, R \\ &+ \sum_{d:D} \sum_{d':D} \mathsf{true} \wedge \neg(d \approx d') \to \mathsf{send}(d)|\mathsf{receive}(d') \,.\, R \\ &\quad + \mathsf{CAE}_{R,\mathsf{send}|\mathsf{receive},\mathsf{transfer},\mathsf{id}}(\tau, \mathsf{transfer}(d), \mathsf{true} \wedge d \approx d') \end{aligned}$$

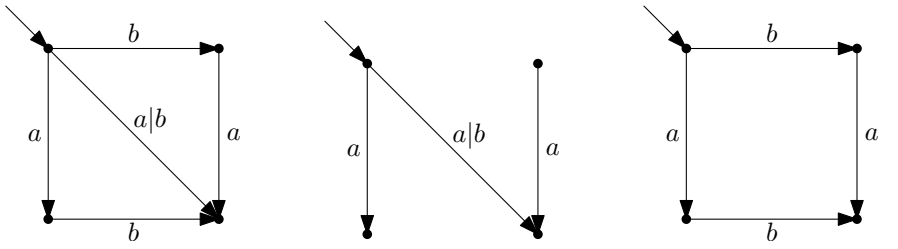
In this second recursion, $\mathsf{choose}_2(\tau)$ yields the empty set, so no candidates for substitution:

$$\begin{aligned} R = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \,.\, R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \,.\, R \\ &+ \sum_{d:D} \sum_{d':D} \mathsf{true} \wedge \neg(d \approx d') \to \mathsf{send}(d)|\mathsf{receive}(d') \,.\, R \\ &\quad + \mathsf{true} \wedge d \approx d' \to \mathsf{transfer}(d) \,.\, R \end{aligned}$$

Note that even though send and $\mathsf{receive}$ can now communicate to $\mathsf{transfer}$, this does not exclude those actions from occurring by themselves or occurring simultaneously with different parameters.

2.9 Allow operator

The allow operator does not change transition labels, but blocks transitions based on their action labels. It uses a set of multi-action names, called the *allow set*, as a white-list. The allow operator prevents multi-actions from occurring unless its multi-action name is in the allow set. This allow set is a parameter to the allow operator. Data parameters are ignored.



(a) System $t_1 \,||\, t_2$ (b) System $\nabla_{\{a,a|b\}}(t_1 \,||\, t_2)$ (c) System $\nabla_{\{a,b\}}(t_1 \,||\, t_2)$

Figure 3: Example of the allow operator. Note that in the system $\nabla_{\{a,b\}}(t_1 \,||\, t_2)$ on the right, the transition with label $a|b$ is removed, even though a and b are in the allow set.

On labeled transition systems, the formal definition of the allow operator is as follows:

Definition 2.9.1 (Allow operator)**.** Let V be a set of multi-action names. The *allow operator* for V applied to labeled transition system t, denoted $\nabla_V(t)$, is defined as follows:

- $S_{\nabla_V(t)} = S_t$
- $\rightarrow_{\nabla_V(t)} = \{(s, \alpha, s') \in S_t \times \mathsf{Act} \times S_t \mid s \xrightarrow{\alpha}_t s' \wedge \underline{\alpha} \in V \cup \{\tau\}\}$
- $i_{\nabla_V(t)} = i_t$

Note that a τ transition cannot be prevented by the allow operator.

Example 2.9.2 (Allow operator)**.** An example of the allow operator on labeled transition systems, is shown in Figure 3. We apply the allow operator to the labeled transition system $t_1 \,||\, t_2$ of example Example 2.7.2.

We derive the definition of the allow operator on linear process equations from the allow operator on the corresponding labeled transition system:

$$[\![\nabla_V(P)(s_i)]\!] = \nabla_V([\![P(s_i)]\!])$$

Let P be the standard process. The derivation is as follows:

$$\begin{aligned}
&\nabla_V([\![P(s_i)]\!]) \\
&= \{\text{Definition 2.4.2 (Linear process equation semantics)}\} \\
&\quad \nabla_V((\mathcal{S}, \{(s, \alpha_i(s,e), g_i(s,e)) \mid s \in S \wedge i \in I \wedge e : E_i \wedge c_i(s,e)\}, s_i)) \\
&= \{\text{Definition 2.9.1 (Allow operator)}\} \\
&\quad (\mathcal{S}, \{(s, \alpha_i(s,e), g_i(s,e)) \mid s \in S \wedge i \in I \wedge e : E_i \wedge c_i(s,e) \wedge \underline{\alpha_i(s,e)} \in V \cup \{\tau\}\}, s_i)
\end{aligned}$$

We can determine whether $\underline{\alpha_i(s,e)}$ is an element of $V \cup \{\tau\}$ for each condition-action-effect clause. The allowed subset I' of I is where the action is in the allow set: $I' = \{i \mid \underline{\alpha_i(s,e)} \in V \cup \{\tau\}\}$.

$$\begin{aligned}
&= \{\text{set theory}\} \\
&\quad (\mathcal{S}, \{(s, \alpha_i(s,e), g_i(s,e)) \mid s \in S \wedge i \in I' \wedge e : E_i \wedge c_i(s,e)\}, s_i) \\
&= \{\text{Definition 2.4.2 (Linear process equation semantics)}\} \\
&\quad [\![P(s:S) = \sum_{i \in I'} \sum_{e:E_i} c_i(s,e) \rightarrow \alpha_i(s,e) \;.\; P(g_i(s,e))]\!](s_i)
\end{aligned}$$

So, we define the allow operator on linear process equations as follows:

Definition 2.9.3 (Allow operator on linear process equations)**.** Let process P be the standard process, and let V be a set of multi-action names. The allow operator for V applied to P, denoted $\nabla_V(P)$, is again a linear process equation, defined as follows:

$$\nabla_V(P)(s:S) = \sum_{i \in I'} \sum_{e:E_i} c_i(s,e) \to \alpha_i(s,e) \;.\; P(g_i(s,e)),$$

where $I' = \{i \mid \underline{\alpha_i(s,e)} \in V \cup \{\tau\}\}$.

Lemma 2.9.4 (Allow operator correspondence)**.** The allow operator on linear process equations corresponds to the allow operator on labeled transition systems, i.e. it holds that

$$[\![\nabla_V(P)(s)]\!] = \nabla_V([\![P(s)]\!]).$$

Proof. By the derivation of allow operator and Definition 2.9.3 (Allow operator on linear process equations). □

We revisit the processes that keep sending and receiving data:

Example 2.9.5 (Allow operator on linear process equations)**.** Consider the process R of Example 2.8.6 (Communication operator on linear process equations):

$$\begin{aligned} R = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; R \\ &+ \sum_{d:D} \sum_{d':D} d \not\approx d' \to \mathsf{send}(d)|\mathsf{receive}(d') \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; R \end{aligned}$$

We noted that while the send and receive actions can communicate to a transfer action, this does not prevent those actions from occurring on their own or occurring simultaneously with different parameters. Using the allow operator, we can do exactly that. The only action we allow, is the successful transfer action transfer. We define the allow set $V = \{\mathsf{transfer}\}$ and calculate $\nabla_V(R)$. There is only one clause for which the actions, stripped of their data parameters, are in the set $\{\mathsf{transfer}, \tau\}$: the last clause. The others are simply removed:

$$\nabla_V(R) = \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; \nabla_V(R)$$

2.10 Rename operator

The rename operator substitutes action names. It is a special case of the communication operator, where the substitutions have single action names on their left hand sides. This is called a single substitution:

Definition 2.10.1 (Single substitutions)**.** A *single substitution* is an expression of the form $a \to a'_1 | \ldots | a'_n$, where a and a'_1 through a'_n are actions.

The rename operator is equal to the communication operator:

Definition 2.10.2 (Rename operator)**.** The *rename operator* for a set of single substitutions C, applied to labeled transition system t, denoted $\rho_C(t)$, is equal to $\Gamma_C(t)$.

We disallow substitutions where the actions on the left and right hand side take expressions of a different sort as parameter.

The reason there is a separate rename operator and not only a communication operator, is that it allows for a simpler definition for linear process equations. Since single substitutions only have single action names on their left hand sides, matching data parameters is trivial. Because of this, we do not need to consider data at all and we can define the renaming on the action right away. We use this observation to derive a concise definition of $\rho_C(P)$ from the equal $\Gamma_C(P)$. Let C be a singleton $a \to \beta$. This can always be established using the rule that $\rho_{C_1 \cup C_2}(P) = \rho_{C_1}(\rho_{C_2}(P))$. We consider the following process:

$$[\![\rho_{\{a \to \beta'\}}(P)(s_i)]\!]$$

We can simply go through the action names in $\alpha_i(s, e)$ and replace the actions one by one. So, we define the rename operator on multi-actions. The rename operator on processes then equals:

$$= \left[\!\!\left[P(s : S) = \sum_{i \in I} \sum_{e : E_i} c_i(s, e) \to \rho_C(\alpha_i(s, e)) \;.\; P(g_i(s, e)) \right]\!\!\right] (s_i)$$

Now, for the definition of rename on multi-actions. This is trivial, following the pattern of Definition 2.8.1 (Substitution function):

$$\begin{aligned} \rho_C(\tau) &= \tau \\ \rho_C(a(e)|\alpha) &= a(e)|\rho_C(\alpha) \\ \rho_C(b(e)|\alpha) &= b(e)|\rho_C(\alpha), \text{ where } b \not\equiv a \end{aligned}$$

We use this as a more convenient definition of the rename operator:

Definition 2.10.3 (Rename operator for linear process equations)**.** Let C be a set of single substitutions and let process P be the standard process. The rename operator of C applied to process P, denoted $\rho_C(P)$, is again a linear process equation, defined as follows:

If C is not a singleton:

$$\rho_{C_1 \cup C_2}(P) = \rho_{C_1}(\rho_{C_2}(P))$$

If C is a singleton $a \to \beta'$:

$$\rho_C(P) = \sum_{i \in I} \sum_{e : E_i} c_i(s, e) \to \rho_C(\alpha_i(s, e)) \;.\; P(g_i(s, e))$$

The rename operator on multi-actions is defined as:

$$\begin{aligned} \rho_C(\tau) &= \tau \\ \rho_C(a(e)|\alpha) &= a(e)|\rho_C(\alpha) \\ \rho_C(b(e)|\alpha) &= b(e)|\rho_C(\alpha), \text{ where } b \not\equiv a \end{aligned}$$

Example 2.10.4 (Rename operator)**.** Consider the process R Example 2.8.6 (Communication operator on linear process equations). The process R is defined as follows:

$$\begin{aligned} R = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; R \\ &+ \sum_{d:D} \sum_{d':D} d \not\approx d' \to \mathsf{send}(d)|\mathsf{receive}(d') \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; R \end{aligned}$$

We rename the send action to an out action. By simply going through every multi-action and replacing the action name send with out, we obtain:

$$\begin{aligned} \rho_{\{\mathsf{send}\to\mathsf{out}\}}(R) = &\sum_{d:D} \mathsf{true} \to \mathsf{out}(d) \;.\; \rho_{\{\mathsf{send}\to\mathsf{out}\}}(R) \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; \rho_{\{\mathsf{send}\to\mathsf{out}\}}(R) \\ &+ \sum_{d:D} \sum_{d':D} d \not\approx d' \to \mathsf{out}(d)|\mathsf{receive}(d') \;.\; \rho_{\{\mathsf{send}\to\mathsf{out}\}}(R) \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; \rho_{\{\mathsf{send}\to\mathsf{out}\}}(R) \end{aligned}$$

We use the rename operator ρ, with no substitution set, to denote replacing every action name a with a fresh action name a':

Definition 2.10.5 (Rename operator with no substitution set)**.** Let process P be the standard process. The rename operator for fresh action names, applied to P, denoted $\rho(P)$, is defined as follows:

$$\rho(P) = \sum_{i\in I} \sum_{e:E_i} c_i(s,e) \to \rho(\alpha_i(s,e)) \;.\; \rho(P(g_i(s,e)))$$

The rename operator for fresh action names on multi-actions is defined as:

$$\begin{aligned} \rho(\tau) &= \tau \\ \rho(a(e)|\alpha) &= a'|\rho(\alpha) \end{aligned}$$

Here, a' is a fresh action name, not occurring in P.

2.11 Abstraction operator

The abstraction operator hides behavior of the system, by renaming specified multi-actions into the internal action τ. It can be thought of as the communication operator with τ on the right hand side of the substitutions and without parameter matching.

First, we define this abstraction on multi-actions:

Definition 2.11.1 (Abstraction on multi-actions)**.** Let H be a set of non-empty multi-action names. The *abstraction operator* for H applied to multi-action α, denoted $\tau_H(\alpha)$, is defined as follows:

$$\tau_{H_1 \cup H_2}(\alpha) = \tau_{H_1}(\tau_{H_2}(\alpha))$$

$$\tau_{\{\beta\}}(\alpha) = \begin{cases} \tau_{\{\beta\}}(\alpha \setminus \beta') & \text{, for some } \beta' \sqsubseteq \alpha \text{ and } \underline{\beta'} = \beta \\ \alpha & \text{, if no such } \beta' \text{ exists} \end{cases}$$

The abstraction set must yield a *confluent* abstraction function: the order in which the abstractions are applied does not influence the result. Formally, we disallow sets of multi-actions for which the first rule in Definition 2.11.1 (Abstraction on multi-actions) yields different results for different partitionings of H_1, H_2, i.e. it must hold that $\tau_{H_1}(\tau_{H_2}(\alpha)) = \tau_{H_3}(\tau_{H_4}(\alpha))$, for any H_3, H_4 such that $H_1 \cup H_2 = H_3 \cup H_4$.

Example 2.11.2 (Abstraction operator on multi-actions)**.** The abstraction operator hides multi-actions occurring in another multi-action, if its multi-action name matches a multi-action name in the abstraction set. For example, $\tau_{\{a|b\}}(a|b|c) = c$. If it occurs multiple times, it is hidden multiple times: $\tau_{\{a|b\}}(a|a|b|b|c) = c$. Any parameters are not taken into account: $\tau_{\{a|b\}}(a(1)|b(2)) = \tau$.

We disallow the abstraction set $\{a|b, a|c\}$, as it is not confluent. It matters which multi-action name is abstracted from first: $\tau_{\{a|b\}}(\tau_{\{a|c\}}(a|b|c)) = \tau_{\{a|b\}}(b) = b$ is not equal to what we obtain when we hide the multi-actions in reverse order: $\tau_{\{a|c\}}(\tau_{\{a|b\}}(a|b|c)) = \tau_{\{a|c\}}(c) = c$.

The abstraction operator for a labeled transition system is the abstraction operator for multi-actions applied to all transition labels:

Definition 2.11.3 (Abstraction operator)**.** Let H be a set of multi-action names. The *abstraction operator* for H applied to labeled transition system t, denoted $\tau_H(t)$, is defined as follows:

- $S_{\tau_H(t)} = S_t$
- $\rightarrow_{\tau_H(t)} = \{(s, \tau_H(\alpha), s') \in S_t \times \mathsf{Act} \times S_t \mid s \xrightarrow{\alpha}_t s'\}$
- $i_{\tau_H(t)} = i_t$

We can derive the abstraction operator on linear process equations from the abstraction operator on the corresponding labeled transition system:

$$[\![\tau_H(P)]\!] = \tau_H([\![P]\!])$$

Assume H is a singleton β, which can always be established using the rule $\tau_{H_1 \cup H_2}(P) = \tau_{H_1}(\tau_{H_2}(P))$.

$$
\begin{aligned}
&\tau_{\{\beta\}}(P)\\
&= \{\text{Definition 2.4.2 (Linear process equation semantics)}\}\\
&\quad \tau_{\{\beta\}}((\mathcal{S}, \{(s, \alpha_i(s,e), g_i(s,e)) \mid s \in S \wedge i \in I \wedge e : E_i \wedge c_i(s,e)\}, s_i))\\
&= \{\text{Definition 2.11.3 (Abstraction operator)}\}\\
&\quad (\mathcal{S}, \{(s, \tau_{\{\beta\}}(\alpha_i(s,e)), g_i(s,e)) \mid s \in S \wedge i \in I \wedge e : E_i \wedge c_i(s,e)\}, s_i))
\end{aligned}
$$

Similar to the derivation of the rename operator, we define the subsets I_m of I as those clauses with exactly m occurrences of β, excluding parameters. We can express this as

$$I_m = \{i \mid \underbrace{\beta|\ldots|\beta}_{m} \sqsubseteq \underline{\alpha_i(s,e)} \wedge \underbrace{\beta|\ldots|\beta}_{m+1} \not\sqsubseteq \underline{\alpha_i(s,e)}\}.$$

There are at most $n = \lfloor \frac{|\alpha_i(s,e)|}{|\beta|} \rfloor$ occurrences of β. Let $\beta = b_1|\ldots|b_k$. We obtain:

$$
\begin{aligned}
P(s:S) = &\sum_{i \in I_0} \sum_{e:E_i} c_i(s,e) \to \alpha_i(s,e) \cdot P(g_i(s,e))\\
&+ \sum_{i \in I_1} \sum_{e:E_i} c_i(s,e) \to \alpha_i(s,e) \setminus b_1(e_{1,1})|\ldots|b_k(e_{1,k}) \cdot P(g_i(s,e))\\
&\vdots\\
&+ \sum_{i \in I_n} \sum_{e:E_i} c_i(s,e) \to\\
&\qquad \alpha_i(s,e) \setminus b_1(e_{1,1})|\ldots|b_k(e_{1,k}) \setminus \cdots \setminus b_1(e_{n,1})|\ldots|b_k(e_{n,k}) \cdot P(g_i(s,e)),
\end{aligned}
$$

where $e_{m,x}$ is the expression of action b_x in the m^{th} occurrence of β. As τ_H is required to be confluent, the exact order does not matter.

So, we define the abstraction operator on linear process equations as follows:

Definition 2.11.4 (Abstraction operator on linear process equations)**.** Let process P be the standard process, and let H be a set of multi-action names. The abstraction operator for H applied to P, denoted $\tau_H(P)$, is again a linear process equation, defined as follows:

$$
\begin{aligned}
\tau_{\{\beta\}}(P)(s:S) = &\sum_{i \in I_0} \sum_{e:E_i} c_i(s,e) \to \alpha_i(s,e) \cdot \tau_{\{\beta\}}(P)(g_i(s,e))\\
&+ \sum_{i \in I_1} \sum_{e:E_i} c_i(s,e) \to \alpha_i(s,e) \setminus b_1(e_{1,1})|\ldots|b_k(e_{1,k}) \cdot \tau_{\{\beta\}}(P)(g_i(s,e))\\
&\vdots\\
&+ \sum_{i \in I_n} \sum_{e:E_i} c_i(s,e) \to\\
&\qquad \alpha_i(s,e) \setminus b_1(e_{1,1})|\ldots|b_k(e_{1,k}) \setminus \cdots \setminus b_1(e_{n,1})|\ldots|b_k(e_{n,k}) \cdot \tau_{\{\beta\}}(P)(g_i(s,e)),
\end{aligned}
$$

where $\beta = b_1|\ldots|b_k$, the sets $I_m = \{i \mid \underbrace{\beta|\ldots|\beta}_{m} \sqsubseteq \underline{\alpha_i(s,e)} \wedge \underbrace{\beta|\ldots|\beta}_{m+1} \not\sqsubseteq \underline{\alpha_i(s,e)}\}$ and expression $e_{m,x}$ is the parameter of action b_x in the m^{th} occurrence of β.

Lemma 2.11.5 (Abstraction operator correspondence)**.** The abstraction operator on linear process equations corresponds to the abstraction operator on its semantic labeled transition system, i.e. it holds that $[\![\tau_H(P)(s_i)]\!] = \tau_H([\![P(s_i)]\!])$.

Proof. By the derivation of the abstraction operator and Definition 2.11.4 (Abstraction operator on linear process equations). □

Example 2.11.6 (Abstraction operator on linear process equations)**.** Consider the process R of Example 2.8.6 (Communication operator on linear process equations):

$$\begin{aligned} R = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; R \\ &+ \sum_{d:D} \sum_{d':D} d \not\approx d' \to \mathsf{send}(d)|\mathsf{receive}(d') \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; R \end{aligned}$$

Suppose that we would like to abstract from the possibility of send and receive occurring separately or simultaneously with different parameters, but not blocking them as in Example 2.9.5. So, we allow the system to perform these steps, but we hide their specifics, as we are only interested in the transfer action. This can be done by the abstraction operator:

$$\begin{aligned} \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) = &\sum_{d:D} \mathsf{true} \to \tau \;.\; \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \\ &+ \sum_{d:D} \mathsf{true} \to \tau \;.\; \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \\ &+ \sum_{d:D} \sum_{d':D} d \not\approx d' \to \tau.\; \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \\ &+ \sum_{d:D} \mathsf{true} \to transfer(d) \;.\; \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \end{aligned}$$

This can be reduced to:

$$\begin{aligned} \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) = &\mathsf{true} \to \tau \;.\; \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; \tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \end{aligned}$$

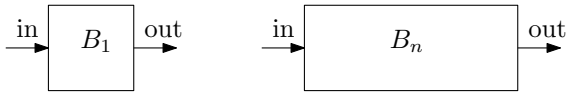
We have now hidden what other steps R can perform besides transfer.

2.12 Buffer example (2)

In Section 2.5, we introduced the process B_1, a one-place buffer. We can string multiple one-place buffers together, to form an n-place buffer. In this section, we use the process operators to construct an n-place buffer from n one-place buffers. We use an inductive construction: given an n-place buffer B_n and the one-place buffer B_1 we have already seen, we construct an $n+1$-place buffer. The illustrations below have no formal meaning and are meant to give some intuition to the process we have modeled.

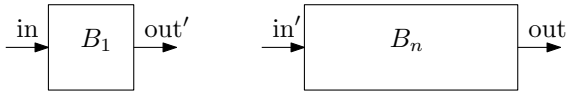
The first step is to put B_1 and B_n in parallel:

$B_1 \,||\, B_n$



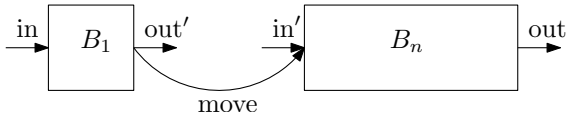
Now, we would like the output of B_1 to serve as input to B_n. The problem is that we cannot distinguish the actions of B_n and B_1. For this, we use the rename operator: we rename the "out" action of B_1 to out$'$, and "in" of B_n to in$'$:

$\rho_{\{\text{out}\to\text{out}'\}}(B_1) \,||\, \rho_{\{\text{in}\to\text{in}'\}}(B_n)$



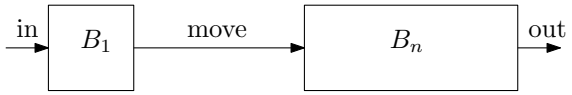
The next logical step is to have these primed actions communicate. This enables us to model that the data element which is given to the n-place buffer is the same element that was outputted by the one-place buffer. We call this action "move". For example, out$'(1)|$in$'(1)$ communicates to move(1).

$B_{n+1} = \Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}(\rho_{\{\text{out}\to\text{out}'\}}(B_1) \,||\, \rho_{\{\text{in}\to\text{in}'\}}(B_n))$



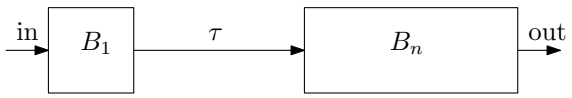
This does not prevent the primed actions from occurring with different parameters, e.g. out$(1)|$in(2), or occurring on their own. To remove this behavior from our model, we use the allow operator. We specify that only the unprimed "in" and "out" actions and the communicating "move" actions are allowed.

$\nabla_{\{\text{in,out,move}\}}(\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}(\rho_{\{\text{out}\to\text{out}'\}}(B_1) \,||\, \rho_{\{\text{in}\to\text{in}'\}}(B_n)))$



The "move" action is really an internal action of the buffer and not an interface to the outside world. To make this explicit, we abstract from the "move" action with the abstraction operator.

$\tau_{\{\text{move}\}}(\nabla_{\{\text{in,out,move}\}}(\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}(\rho_{\{\text{out}\to\text{out}'\}}(B_1) \,||\, \rho_{\{\text{in}\to\text{in}'\}}(B_n))))$



3 Specifications

We present the fixpoint equations family of formalisms in which the specification of a system can be expressed. These are the modal equation systems (Andersen [1]), inspired by the boolean equation systems (Mader [8]), and its parameterized version, the parameterized modal equation systems, which is a new formalism.

Modal equation systems reason over sets of states. The parameterized variants add reasoning with data, allowing for quantification over data values and reasoning over infinitely many variables. Parameterized modal equation systems can also be thought of as an equational variant of the well-known first-order modal μ-calculus. We present the parameterized modal equation systems in detail.

3.1 Parameterized modal equation systems

We use $\mathcal{S}$ to denote the set of states, which encompasses all specific state spaces of processes. This is similar to how the generalized data sort D encompasses all specific user-defined data sorts $D_1, D_2, \ldots$.

Parameterized modal equation systems are a list of equations over a set of recursion variables $\mathcal{X}$. The semantics of a parameterized modal equation system is an assignment of a set of states for each recursion variable. This is called an environment:

Definition 3.1.1 (Environments)**.** Let $\mathcal{X}$ be a set of variables and $\mathcal{S}$ be a set of states. An *environment* $\rho : \mathcal{X} \to (D \to 2^{\mathcal{S}})$ is a function that assigns functions from data to sets of states to variables.

The right hand sides of the equations in a parameterized modal equation system are called assertions:

Definition 3.1.2 (Assertions)**.** The *assertions* are given by the following grammar:

$$
\begin{aligned}
A ::= {} & X(e) \\
\mid {} & b \\
\mid {} & A \vee A \\
\mid {} & A \wedge A \\
\mid {} & \langle af \rangle A \\
\mid {} & [af] A \\
\mid {} & (\exists d : D.A) \\
\mid {} & (\forall d : D.A)
\end{aligned}
$$

Here, b is a boolean expression and af is an action formula.

The semantics of assertions depend on a labeled transition system t, a data environment and an environment for the recursion variables. The assertion's semantics are as follows:

Definition 3.1.3 (Assertion semantics)**.** The semantics of assertion $\mathcal{A}$ in environment ρ, data environment ε and labeled transition system t, denoted $[\![\mathcal{A}]\!]_t\rho\varepsilon$, are defined as follows:

$$
\begin{aligned}
[\![X(e)]\!]_t\rho\varepsilon &= \rho(X)([\![e]\!]\varepsilon)\\
[\![b]\!]_t\rho\varepsilon &= \begin{cases}\mathcal{S} & \text{if } [\![b]\!]\varepsilon\\ \emptyset & \text{otherwise}\end{cases}\\
[\![\mathcal{A}_1 \vee \mathcal{A}_2]\!]_t\rho\varepsilon &= [\![\mathcal{A}_1]\!]_t\rho\varepsilon \cup [\![\mathcal{A}_2]\!]_t\rho\varepsilon\\
[\![\mathcal{A}_1 \wedge \mathcal{A}_2]\!]_t\rho\varepsilon &= [\![\mathcal{A}_1]\!]_t\rho\varepsilon \cap [\![\mathcal{A}_2]\!]_t\rho\varepsilon\\
[\![\langle af\rangle\mathcal{A}]\!]_t\rho\varepsilon &= \{s \in S_t \mid (\exists s' \in S_t \;:\; s \xrightarrow{\alpha}_t s' \wedge \alpha \in [\![af]\!]\varepsilon \wedge s' \in [\![\mathcal{A}]\!]_t\rho\varepsilon)\}\\
[\![[af]\mathcal{A}]\!]_t\rho\varepsilon &= \{s \in S_t \mid (\forall s' \in S_t \;:\; s \xrightarrow{\alpha}_t s' \wedge \alpha \in [\![af]\!]\varepsilon \Rightarrow s' \in [\![\mathcal{A}]\!]_t\rho\varepsilon)\}\\
[\![(\exists d : D.\mathcal{A})]\!]_t\rho\varepsilon &= \bigcup_{v:D} [\![\mathcal{A}]\!]_t\rho\varepsilon[d := v]\\
[\![(\forall d : D.\mathcal{A})]\!]_t\rho\varepsilon &= \bigcap_{v:D} [\![\mathcal{A}]\!]_t\rho\varepsilon[d := v]
\end{aligned}
$$

Here, e is a data expression and $[\![e]\!]\varepsilon$ denotes the expression's semantics in data environment ε, and similarly for the boolean expression b. We often drop the subscript t if it is clear from the context.

We often use a linear process equation P instead of an labeled transition system t with the semantics of assertions and parameterized modal equation systems. As the semantics of linear process equations are labeled transition systems, this is an obvious generalization.

Example 3.1.4 (Assertions)**.** Some simple examples of assertions are as follows:

- "It is possible to perform an a-action": $\langle a\rangle\mathsf{true}$
- "No a-action is possible": $[a]\mathsf{false}$
- "An action other than a is possible": $\langle \neg a\rangle\mathsf{true}$
- "An a or b-action is possible": $\langle a \vee b\rangle\mathsf{true}$

Assertions provide some reasoning over paths, but only paths of limited depth. For example, we cannot express "there is an infinite a-path" as an assertion. This is handled by the fixpoints in a parameterized modal equation system. The parameterized modal equation systems are lists of equations with a fixpoint symbol and a recursion variable on the left hand side and an assertion on the right hand side:

Definition 3.1.5 (Parameterized modal equation system)**.** The *parameterized modal equation systems* are given by the following grammar:

$$PMES ::= \epsilon \mid (\sigma X(d : D) = \mathcal{A})\ PMES$$

Here, $\sigma \in \{\nu, \mu\}$ is the fixpoint symbol, $X \in \mathcal{X}$ is a variable, d is a data variable appropriate for X and $\mathcal{A}$ is an assertion. For nonempty parameterized modal equation systems, the trailing ϵ is left implicit.

We often write a parameterized modal equation system

$$\mathcal{E} = (\sigma_1 X_1(d_1 : D_1) = \mathcal{A}_1)\ \ldots\ (\sigma_n X_n(d_n : D_n) = \mathcal{A}_n)$$

in the following vertical format:

$$\mathcal{E} = \begin{pmatrix} \sigma_1 X_1(d_1 : D_1) = \mathcal{A}_1 \\ \dots \\ \sigma_n X_n(d_n : D_n) = \mathcal{A}_n \end{pmatrix}$$

A variable X is *bound* in $\mathcal{E}$ if and only if there is an equation in $\mathcal{E}$ with X on its left hand side. Similarly, a variable X is *occurring* in $\mathcal{E}$ if and only if there is an equation in $\mathcal{E}$ with X on its right hand side.

Definition 3.1.6 (Bound and occurring variables)**.**

$$\begin{aligned} \mathsf{bnd}(\epsilon) &= \emptyset \\ \mathsf{bnd}((\sigma X(d : D) = \mathcal{A})\ \mathcal{E}) &= \{X\} \cup \mathsf{bnd}(\mathcal{E}) \end{aligned}$$

$$\begin{aligned} \mathsf{occ}(\epsilon) &= \emptyset \\ \mathsf{occ}((\sigma X(d : D) = \mathcal{A})\ \mathcal{E}) &= \mathsf{occ}(\mathcal{A}) \cup \mathsf{occ}(\mathcal{E}) \\ \mathsf{occ}(\mathsf{b}) &= \emptyset \\ \mathsf{occ}(X) &= \{X\} \\ \mathsf{occ}(\mathcal{A}_1 \vee \mathcal{A}_2) &= \mathsf{occ}(\mathcal{A}_1) \cup \mathsf{occ}(\mathcal{A}_2) \\ \mathsf{occ}(\mathcal{A}_1 \wedge \mathcal{A}_2) &= \mathsf{occ}(\mathcal{A}_1) \cup \mathsf{occ}(\mathcal{A}_2) \\ \mathsf{occ}(\langle af \rangle \mathcal{A}) &= \mathsf{occ}(\mathcal{A}) \\ \mathsf{occ}([af]\mathcal{A}) &= \mathsf{occ}(\mathcal{A}) \end{aligned}$$

A parameterized modal equation system $\mathcal{E}$ is called *closed* if and only if all occurring variables are also bound, i.e. $\mathsf{occ}(\mathcal{E}) \subseteq \mathsf{bnd}(\mathcal{E})$. Otherwise, it is called *open*.

The semantics of a parameterized modal equation system is an environment where the recursion variable on the left hand side of each equation matches its definition on the right. Moreover, it is the least or greatest such value for each recursion variable, depending on the fixpoint symbol, where fixpoint symbols higher in the list take priority over those lower in the list:

Definition 3.1.7 (Parameterized modal equation system semantics)**.** The semantics of parameterized modal equation system $\mathcal{E}$ in environment ρ, data environment ε and labeled transition system t, denoted $[\![\mathcal{E}]\!]_t \rho\varepsilon$, are defined as follows:

$$\begin{aligned} [\![\epsilon]\!]_t \rho\varepsilon &= \rho \\ [\![(\sigma X(d : D) = \mathcal{A})\ \mathcal{E}]\!]_t \rho\varepsilon &= [\![\mathcal{E}]\!]_t \rho[X := \sigma\Phi\rho\varepsilon]\varepsilon, \text{ where} \\ &\qquad \Phi\rho\varepsilon := (\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D.[\![\mathcal{A}]\!]_t([\![\mathcal{E}]\!]_t \rho[X := F]\varepsilon[d := v])\varepsilon)) \end{aligned}$$

Note that if a parameterized modal equation system $\mathcal{E}$ is closed, $[\![\mathcal{E}]\!]_t \rho_1\varepsilon = [\![\mathcal{E}]\!]_t \rho_2\varepsilon$ holds for all environments ρ_1, ρ_2, labeled transition system t and data environment ε.

With parameterized modal equation systems, we can express much more about paths of actions:

Example 3.1.8 (Parameterized modal equation systems)**.** Examples of parameterized modal equation systems:

- "an a-action will eventually occur":

  $$\mu X : [\neg a]X \wedge \langle \mathsf{true} \rangle \mathsf{true}$$

- "there is an infinite a-path":

  $$\nu X : \langle a \rangle X$$

- "an a-action is always possible":

  $$\nu X : [\mathsf{true}]X \wedge \langle a \rangle \mathsf{true}$$

We will consider only parameterized modal equation systems where the data variables on the right hand sides are bound on the left hand sides. It has been shown that for such parameterized modal equation systems, the data environment has no influence on the semantics of a parameterized modal equation system:

Lemma 3.1.9 (Parameterized modal equation system semantics data environment independence)**.** Let ρ be an environment and let ε and ε' be data environments. Let $\mathcal{E}$ be a parameterized modal equation system for which all data variables occurring at the right-hand side of an equation are bound in the left-hand side. Then $[\![\mathcal{E}]\!]\rho\varepsilon = [\![\mathcal{E}]\!]\rho\varepsilon'$.

Proof. The situation is analogous to the case for parameterized *boolean* equation systems of Groote and Willemse [4] (lemma 2.4). □

In the remainder of this thesis, we consider only parameterized modal equation systems for which all data variables occurring at the right-hand side of an equations are indeed bound in the left-hand side. As the data environment has no influence on the semantics of such systems, we generally omit it.

We are usually interested in only one variable of the parameterized modal equation system, which is selected by a top assertion:

Definition 3.1.10 (Top assertions)**.** A *top assertion* is an expression $\mathcal{E} \downarrow X(e)$ for a closed parameterized modal equation system $\mathcal{E}$, a variable $X \in \mathsf{bnd}(\mathcal{E})$ and $e : D$.

The semantics of a top assertion are simply the value of the selected variable in the semantics of the parameterized modal equation system:

Definition 3.1.11 (Top assertion semantics)**.** The semantics of a top assertion $\mathcal{E} \downarrow X(e)$ in labeled transition system t, environment ρ and data environment ε, denoted $[\![\mathcal{E} \downarrow X(e)]\!]_t\rho\varepsilon$, are defined as follows:

$$[\![\mathcal{E} \downarrow X(e)]\!]_t = ([\![\mathcal{E}]\!]_t\rho\varepsilon)(X)([\![e]\!]\varepsilon)$$

Finally, we define the notion of satisfaction, i.e. when a system satisfies a top assertion. A system satisfies a top assertion if and only if its initial state is in the top assertion's semantics:

Definition 3.1.12 (Satisfaction)**.** Let t be a labeled transition system and let $\mathcal{E} \downarrow X(e)$ be a top assertion. System t *satisfies* $\mathcal{E} \downarrow X(e)$ in environment ρ and data environment ε, denoted $t \models \mathcal{E} \downarrow X(e)\rho\varepsilon$, if and only if $i_t \in [\![\mathcal{E} \downarrow X(e)]\!]_t\rho\varepsilon$.

3.2 Buffer example (3)

In previous sections, we modeled an n-place buffer. An instance of the model checking problem consists of two parts: a model of a system and a property. In our setting, this property is expressed as a parameterized modal equation system. We define the requirement that every element that goes in our buffer, must eventually come out.

$$\begin{pmatrix} \nu X = [\mathsf{true}]X \wedge (\forall m : D \;.\; [\mathrm{in}(m)]Y(m)) \\ \mu Y(m : D) = [\neg \mathrm{out}(m)]Y(m) \wedge \langle \mathsf{true} \rangle \mathsf{true} \end{pmatrix} \downarrow X$$

The equation for X corresponds to the part where something must always hold: after any step, X must remain valid. The thing that must hold, is that for any m, after an $\mathrm{in}(m)$ transition, $Y(m)$ must be valid. Using a parameter for Y, we express that the element that eventually comes out the buffer, is actually the same element that went in earlier.

Now, the $Y(m)$ equation expresses that an $\mathrm{out}(m)$ action eventually occurs, i.e. within a finite number of transitions. After any transition *other* than $\mathrm{out}(m)$, $Y(m)$ must still hold. Also, some transition must be enabled, otherwise there is no path to the $\mathrm{out}(m)$ transition.

In general, least fixpoints describe finite behavior and greatest fixpoint additionally also infinite behavior (from Mader [8]). This can be seen in our small example. The equation for X is a greatest fixpoint equation, as it must be true of *all* paths, including infinite paths. The equation for $Y(m)$ is a least fixpoint equation, because the $\mathrm{out}(m)$ transition may not be postponed indefinitely: it must be satisfied on some *finite* path.

4 Quotienting

Quotienting takes a parallel component from the model side of the model checking equation, and incorporates its behavior in the property that has to be checked against the remainder. So, let $\mathcal{E}$ be a property to be checked on $P_1 \,||\, P_2$. By *quotienting out* P_2 from $\mathcal{E}$, we mean the construction of a new property $\mathcal{E}/P_2$, such that the original model checking problem, $P_1 \,||\, P_2 \models \mathcal{E}$, has the same solution as $P_1 \models \mathcal{E}/P_2$. In this example we quotient out a parallel process, but we could also quotient out process operators. The important thing to note is that after quotienting, the thing that was quotiented out, is no longer present at the model side of the model checking equation, and incorporated in the property side.

In this section, we look only at quotienting out linear process equations. First, we look at quotienting on finite labeled transition systems, which was already established in previous work. Then, we extend the quotienting procedure to linear process equations, which can have infinitely large state spaces.

One idea we very often use, is to explicitly split a multi-action in pairs of multi-actions, e.g. we split $a|b$ into $(a|b, \tau), (a, b), (b, a)$ and $(\tau, a|b)$. Then, we imagine one part to be done by the process we are quotienting out, and the other part will remain in the formula. We can immediately check whether the process we are quotienting out can perform a step, or we can at least construct a boolean expression that says if it can.

4.1 Quotienting on labeled transition systems

Quotienting on finite labeled transition systems was defined by Andersen [1]. He used a slightly different definition for parallel composition, so we present here the version of Van der Pol [12]. This version is very similar and uses the mCRL2 definition of parallel composition. The formalism used here, is parameterless modal equation systems.

As stated, the most important drawback of the definition on labeled transition systems is that it requires the models to have enumerable state spaces. This model checking procedure also uses single action names in the modal operators and does not deal with quantifiers in assertions.

The formal definition of quotienting on labeled transition systems is as follows:

Definition 4.1.1 (Quotienting on labeled transition systems)**.** Let $t = (\{s_1, \ldots, s_n\}, \rightarrow_t, i_t)$ be a labeled transition system. Quotienting a state $s \in S_t$ from an assertion is defined as follows:

$$
\begin{aligned}
X/s &= X_s \\
\mathsf{true}/s &= \mathsf{true} \\
(\mathcal{A}_1 \wedge \mathcal{A}_2)/s &= (\mathcal{A}_1/s) \wedge (\mathcal{A}_2/s) \\
(\langle\alpha\rangle\mathcal{A})/s &= \bigvee_{\beta|\gamma=\alpha} \bigvee_{s \xrightarrow{\beta} s'} \langle\gamma\rangle(\mathcal{A}/s') \\
&\quad \vee \begin{cases} \bigvee_{s \xrightarrow{\beta} s'} (\mathcal{A}/s') & \text{if } \gamma = \tau \\ \mathsf{false} & \text{otherwise} \end{cases} \\
&\quad \vee \begin{cases} \langle\gamma\rangle(\mathcal{A}/s) & \text{if } \beta = \tau \\ \mathsf{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Quotienting t from a modal equation system is defined as follows:

$$\epsilon/t = \epsilon$$

$$((\sigma X = \mathcal{A})\ \mathcal{E})/t = \begin{pmatrix} \sigma X_{s_1} = \mathcal{A}/s_1 \\ \cdots \\ \sigma X_{s_n} = \mathcal{A}/s_n \\ \mathcal{E}/t \end{pmatrix}$$

Finally, quotienting a labeled transition system t from a top assertion is defined as follows:

$$(\mathcal{E} \downarrow X)/t = (\mathcal{E}/t) \downarrow X_{i_t}$$

There are two interesting things to note: first, we note that this quotienting procedure creates one equation for each original equation and each state, multiplying the number of equations by the size of the state space. This shows why state spaces are required to be finite and enumerable.

Second, we discuss the intuition behind the rule for the modal operators. As explained, we split the multi-action into pairs, where we check if one part, β, can be done from state s, and the other, γ, remains in the formula. This yields an assertion $\langle\gamma\rangle(\mathcal{A}/s')$ for each possible step $s \xrightarrow{\beta} s'$. Now, if either of the two equals τ, rather than doing a τ transition, it is also fine for that process to remain idle: the multi-actions $\beta|\tau$ and β are equal, as are $\tau|\gamma$ and γ. This gives rise to the two case distinctions in this rule.

Soundness of this quotienting procedure was shown in previous work:

Theorem 4.1.2 (Quotienting is sound (Van der Pol [12]))**.** For labeled transition systems with *finite* state spaces, quotienting is sound with respect to satisfaction, i.e. $t_1 \,||\, t_2 \models \mathcal{E} \downarrow X$, if and only if, $t_1 \models (\mathcal{E} \downarrow X)/t_2$ for closed modal equation system $\mathcal{E}$ and some $X \in \mathsf{bnd}(\mathcal{E})$.

4.2 Quotienting on linear process equations

We derive the correct definition for quotienting *assertions* from the following specification:

Let ρ_1, ρ_2 be two environments such that for any variable X, data element e, state $s_1 \in S_1$ from P_1, $s_2 \in S_2$ from P_2, the following holds[2]:

$$(s_1, s_2) \in \rho_1(X)(e) \quad \Leftrightarrow \quad s_2 \in \rho_2(X')(e, s_1)$$

Assuming such ρ_1, ρ_2, for any two processes P_1 and P_2, state $s_1 \in S_1$ from P_1, $s_2 \in S_2$ from P_2, data environment ε and any assertion $\mathcal{A}$, we define quotienting such that:

$$(s_1, s_2) \in [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}\rho_1\varepsilon \quad \Leftrightarrow \quad s_2 \in [\![\mathcal{A}/P_1(s')]\!]_{P_2}\rho_2\varepsilon[s' := s_1]$$

[2]For clarity, we decorate recursion variables with a prime when we quotient out an linear process equation. This distinguishes X before the quotienting, which takes a number of parameters of a certain sort, from X' after the quotienting, which takes parameters of those same sorts and an additional state parameter.

Derivation:
By structural induction on $\mathcal{A}$. The case for variables $X, Y, \ldots$ follows from the assumption on ρ_1, ρ_2, the boolean case is trivial, disjunction and conjunction simply distribute by Induction Hypothesis and thus also existential and universal quantification. The interesting cases are the modal operators $\langle\cdot\rangle$ and $[\cdot]$.

- $X(e)$

$$
\begin{aligned}
& (s_1, s_2) \in [\![X(e)]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& (s_1, s_2) \in \rho_1(X)([\![e]\!]\varepsilon) \\
= \; & \{\text{assumption on } \rho_1 \text{ and } \rho_2\} \\
& s_2 \in \rho_2(X')([\![e]\!]\varepsilon, s_1) \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s_2 \in [\![X'(e, s')]\!]_{P_2} \rho_2 \varepsilon[s' := s_1]
\end{aligned}
$$

- b

$$
\begin{aligned}
& (s_1, s_2) \in [\![b]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& (s_1, s_2) \in \begin{cases} S(P_1 \,||\, P_2) & \text{if } [\![b]\!]\varepsilon \\ \emptyset & \text{otherwise} \end{cases} \\
= \; & \{\text{assumption on } s_1, s_2;\ s' \text{ is free in } b; \text{ set theory}\} \\
& s_2 \in \begin{cases} S_1 & \text{if } [\![b]\!]\varepsilon[s' := s_1] \\ \emptyset & \text{otherwise} \end{cases} \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s_2 \in [\![b]\!]_{P_2} \rho_2 \varepsilon[s' := s_1]
\end{aligned}
$$

Induction Hypothesis: for any data environment ε', it holds that

$$(s_1', s_2') \in [\![\mathcal{A}]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon' \Leftrightarrow s_2' \in [\![\mathcal{A}/P_1(s')]\!]_{P_2} \rho_2 \varepsilon'[s' := s_1'].$$

- $\mathcal{A}_1 \vee \mathcal{A}_2$

$$
\begin{aligned}
& (s_1, s_2) \in [\![\mathcal{A}_1 \vee \mathcal{A}_2]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& (s_1, s_2) \in [\![\mathcal{A}_1]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon \cup [\![\mathcal{A}_2]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon \\
= \; & \{\text{Induction Hypothesis, twice}\} \\
& s_2 \in [\![\mathcal{A}_1/P_1(s')]\!]_{P_2} \rho_2 \varepsilon[s' := s_1] \cup [\![\mathcal{A}_2/P_1(s')]\!]_{P_2} \rho_2 \varepsilon[s' := s_1] \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s_2 \in [\![(\mathcal{A}_1/P_1(s')) \vee (\mathcal{A}_2/P_1(s'))]\!]_{P_2} \rho_2 \varepsilon[s' := s_1]
\end{aligned}
$$

- $(\exists d : D.\mathcal{A})$

$$
\begin{aligned}
&(s_1, s_2) \in [\![(\exists d : D.\mathcal{A})]\!]_{P_1 \,||\, P_2}\rho_1\varepsilon \\
&= \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
&(s_1, s_2) \in \bigcup_{v \in D} [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}\rho_1\varepsilon[d := v] \\
&= \{\text{Induction Hypothesis; set theory}\} \\
&s_2 \in \bigcup_{v \in D} [\![\mathcal{A}/P_1(s')]\!]_{P_2}\rho_2\varepsilon[d := v, s' := s_1] \\
&= \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
&s_2 \in [\![(\exists d : D.\mathcal{A}/P_1(s'))]\!]_{P_2}\rho_2\varepsilon[s' := s_1]
\end{aligned}
$$

Here, it is assumed that d is a variable occurring free in $\mathcal{A}$.

- $\langle af \rangle \mathcal{A}$

This is the interesting case. First, we restrict ourselves to a convenient subset of action formulae. Assume af is in disjunctive normal form. Using the rule that $\langle af_1 \vee af_2 \rangle \mathcal{A} = \langle af_1 \rangle \mathcal{A} \vee \langle af_2 \rangle \mathcal{A}$, this can be further broken down so that the action formula in the modal operator is a conjunction over boolean expressions b, multi-actions α and negated multi-actions $\neg\alpha$. We derive definitions on conjunctive action formulae using structural induction. As true is the unit element for conjunction, we use this as a base case. The inductive steps are $af \wedge b$, $af \wedge \alpha$ and $af \wedge \neg\alpha$.

We want to construct a formula for $\langle af \rangle \mathcal{A}$, where part of the step is taken by the process we are quotienting out, and the remainder still has to be performed by the remaining process. The case for $[\cdot]$ is very similar. To do this, we first decompose af into a pair (af_1, af_2) such that $\alpha_1|\alpha_2 \in [\![af]\!]\varepsilon \Leftrightarrow \alpha_1 \in [\![af_1]\!]\varepsilon \wedge \alpha_2 \in [\![af_2]\!]\varepsilon$ for any ε. This split can be done in many different ways, so we define the set of all possible splits. This is done by the function split:

The specification for split is: for any ε,

$$\alpha|\beta \in [\![af]\!]\varepsilon \Leftrightarrow \exists_{(af_1, af_2) \in \mathsf{split}(af)} \alpha \in [\![af_1]\!]\varepsilon \wedge \beta \in [\![af_2]\!]\varepsilon.$$

Hypothesizing such a function, we can already see what $\langle af \rangle \mathcal{A}/P(s_i)$ should look like: a large disjunction over each possible split (af_1, af_2) with an assertion we call $\mathsf{Quotient}$. Intuitively, the assertion $\mathsf{Quotient}(af_1, af_2, \mathcal{A}, P(s), i)$ expresses that the process we are quotienting out, P, can perform a step satisfying action formula af_1 from state s, and the remainder process still has to perform a step satisfying af_2, after which assertion $\mathcal{A}$ holds. We use the shorthand notation $s \xrightarrow{af}_{[\![P]\!]} s'$ for the assertion $\bigvee_{i \in I}(\exists e : E_i.c_i(s, e) \wedge \mathsf{cond}(\alpha_i(s, e), af) \wedge s' = g_i(s, e))$. If one of the steps can be τ, then that process can also stay idle.

Definition 4.2.1 ($\mathsf{Quotient}$)**.** We define $\mathsf{Quotient}$ as follows:

$$
\begin{aligned}
\mathsf{Quotient}(af_1, af_2, \mathcal{A}, P(s), i) = (&s \xrightarrow{af_1}_{[\![P]\!]} s' \wedge \langle af_2 \rangle \mathcal{A}/P(s')) \\
&\vee (\tau \in [\![af_2]\!]\varepsilon \wedge s \xrightarrow{af_1}_{[\![P]\!]} s' \wedge \mathcal{A}/P(s'))) \\
&\vee (\tau \in [\![af_1]\!]\varepsilon \wedge \langle af_2 \rangle \mathcal{A}/P(s)))
\end{aligned}
$$

The expression $\tau \in [\![af_1]\!]\varepsilon$ is of course not a valid assertion. We define the function cond to construct it. The function cond takes a multi-action and an action formula and yields a boolean expression. Its specification is: for any ε,

$$[\![\mathsf{cond}(\beta, af)]\!]\varepsilon \Leftrightarrow \beta \in [\![af]\!]\varepsilon.$$

The function cond is straightforward from the semantics of action formulae, Definition 2.2.13 (Action formulae semantics):

Definition 4.2.2 (The function cond)**.** Let β be a multi-action and af an action formula. The boolean expression $\mathsf{cond}(\beta, af)$ is defined as:

$$\begin{aligned}
\mathsf{cond}(\beta, b) &= b \\
\mathsf{cond}(\beta, \alpha) &= \alpha = \beta \\
\mathsf{cond}(\beta, \neg\alpha) &= \alpha \neq \beta \\
\mathsf{cond}(\beta, af_1 \wedge af_2) &= \mathsf{cond}(\beta, af_1) \wedge \mathsf{cond}(\beta, af_2) \\
\mathsf{cond}(\beta, af_1 \vee af_2) &= \mathsf{cond}(\beta, af_1) \vee \mathsf{cond}(\beta, af_2)
\end{aligned}$$

Now, we define split:

- true. This case is trivial:

  $\alpha|\beta \in [\![\mathsf{true}]\!]\varepsilon$
  = {Definition 2.2.13 (Action formulae semantics)}
  true
  = {Definition 2.2.13 (Action formulae semantics)}
  $\alpha \in [\![\mathsf{true}]\!]\varepsilon \wedge \beta \in [\![\mathsf{true}]\!]\varepsilon$

  Induction Hypothesis: $\alpha'|\beta' \in [\![af]\!]\varepsilon \Leftrightarrow \exists_{(af_1, af_2) \in \mathsf{split}(af)} \alpha' \in [\![af_1]\!]\varepsilon \wedge \beta' \in [\![af_2]\!]\varepsilon$

- $af \wedge b$. Slightly less trivial:

  $\alpha|\beta \in [\![af \wedge b]\!]\varepsilon$
  = {Definition 2.2.13 (Action formulae semantics), set theory}
  $\alpha|\beta \in [\![af]\!]\varepsilon \wedge \begin{cases} \mathsf{true} & \text{, if } [\![b]\!]\varepsilon \\ \mathsf{false} & \text{, otherwise} \end{cases}$
  = {Induction Hypothesis}
  $(\exists_{(af_1, af_2) \in \mathsf{split}(af)} \alpha \in [\![af_1]\!]\varepsilon \wedge \beta \in [\![af_2]\!]\varepsilon) \wedge \begin{cases} \mathsf{true} & \text{, if } [\![b]\!]\varepsilon \\ \mathsf{false} & \text{, otherwise} \end{cases}$
  = {set theory, Definition 2.2.13 (Action formulae semantics)}
  $\left(\exists_{(af_1, af_2) \in \mathsf{split}(af)} \alpha \in [\![af_1 \wedge b]\!]\varepsilon \wedge \beta \in [\![af_2 \wedge b]\!]\varepsilon\right)$

- $af \wedge a_1(e_1)|\ldots|a_n(e_n)$. This is a more interesting case. We observe that for $\alpha|\beta$ to equal $a_1(e_1)|\ldots|a_n(e_n)$, some subset of actions $\alpha' \sqsubseteq \alpha$ must equal α, and that the remainder, $a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha'$, must equal β.

$$
\begin{aligned}
&\alpha|\beta \in [\![af \wedge a_1(e_1)|\ldots a_n(e_n)]\!]\varepsilon \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\alpha|\beta \in [\![af]\!]\varepsilon \wedge \alpha|\beta = a_1([\![e_1]\!]\varepsilon)|\ldots|a_n([\![e_n]\!]\varepsilon) \\
=\ &\{\text{Definition 2.2.6 (Multi-action equality, inclusion and removal)}\} \\
&\alpha|\beta \in [\![af]\!]\varepsilon \wedge \bigvee_{\alpha' \sqsubseteq a_1(e_1)|\ldots|a_n(e_n)} \alpha \in [\![\alpha']\!]\varepsilon \wedge \beta \in [\![a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha']\!]\varepsilon \\
=\ &\{\text{Induction Hypothesis, Definition 2.2.13 (Action formulae semantics)}\} \\
&\exists_{(af_1,af_2)\in \mathsf{split}(af)} \bigvee_{\alpha' \sqsubseteq a_1(e_1)|\ldots|a_n(e_n)} \alpha \in [\![af_1 \wedge \alpha']\!]\varepsilon \wedge \beta \in [\![af_2 \wedge a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha']\!]\varepsilon
\end{aligned}
$$

- $af \wedge \neg a_1(e_1)|\ldots|a_n(e_n)$. This is the most interesting case.

  We observe that for $\alpha|\beta$ *not* to equal $a_1(e_1)|\ldots|a_n(e_n)$, some subset of actions α' *may* equal α, as long as β does *not* equal the remainder, $a_1(e_1)|\ldots a_n(e_n) \setminus \alpha'$.

  It can also be the case that α equals some $\alpha' \not\sqsubseteq a_1(e_1)|\ldots|a_n(e_n)$, in which case $\alpha|\beta$ will *never* equal $a_1(e_1)|\ldots|a_n(e_n)$, regardless of β.

$$
\begin{aligned}
&\alpha|\beta \in [\![af \wedge \neg a_1(e_1)|\ldots a_n(e_n)]\!]\varepsilon \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\alpha|\beta \in [\![af]\!]\varepsilon \wedge \alpha|\beta \neq a_1([\![e_1]\!]\varepsilon)|\ldots|a_n([\![e_n]\!]\varepsilon) \\
=\ &\{\text{Definition 2.2.6 (Multi-action equality, inclusion and removal)}\} \\
&\alpha|\beta \in [\![af]\!]\varepsilon \wedge (\\
&\qquad \bigvee_{\alpha' \sqsubseteq a_1(e_1)|\ldots|a_n(e_n)} (\alpha \in [\![\alpha']\!]\varepsilon \wedge \beta \in [\![\neg a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha']\!]\varepsilon \\
&\qquad\qquad \vee \alpha \in [\![\neg a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha']\!]\varepsilon \wedge \beta \in [\![\alpha']\!]\varepsilon) \\
&\qquad \vee \exists_{\alpha' \not\sqsubseteq a_1(e_1)|\ldots|a_n(e_n)} (\alpha \in [\![\alpha']\!]\varepsilon \wedge \beta \in [\![\mathsf{true}]\!]\varepsilon \\
&\qquad\qquad \vee \alpha \in [\![\mathsf{true}]\!]\varepsilon \wedge \beta \in [\![\alpha']\!]\varepsilon)) \\
=\ &\{\text{Induction Hypothesis, Definition 2.2.13 (Action formulae semantics)}\} \\
&\exists_{(af_1,af_2)\in \mathsf{split}(af)} \wedge \alpha \in [\![af_1]\!]\varepsilon \wedge \beta \in [\![af_2]\!]\varepsilon \wedge (\\
&\qquad \bigvee_{\alpha' \sqsubseteq a_1(e_1)|\ldots|a_n(e_n)} ((\alpha \in [\![\alpha']\!]\varepsilon \wedge \beta \in [\![\neg a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha']\!]\varepsilon) \\
&\qquad\qquad \vee (\alpha \in [\![\neg a_1(e_1)|\ldots|a_n(e_n) \setminus \alpha']\!]\varepsilon \wedge \beta \in [\![\alpha']\!]\varepsilon)) \\
&\qquad \vee \exists_{\alpha' \not\sqsubseteq a_1(e_1)|\ldots|a_n(e_n)} ((\alpha \in [\![\alpha']\!]\varepsilon \wedge \beta \in [\![\mathsf{true}]\!]\varepsilon) \\
&\qquad\qquad \vee (\alpha \in [\![\mathsf{true}]\!]\varepsilon \wedge \beta \in [\![\alpha']\!]\varepsilon)))
\end{aligned}
$$

So, let split be defined as follows:

Definition 4.2.3 (The function split)**.**

$$\begin{aligned}
\mathsf{split}(\mathsf{true}) &= \{(\mathsf{true},\mathsf{true})\} \\
\mathsf{split}(af \wedge b) &= \{(af_1 \wedge b, af_2 \wedge b) && \mid (af_1, af_2) \in \mathsf{split}(af)\} \\
\mathsf{split}(af \wedge \alpha) &= \{(af_1 \wedge \alpha', af_2 \wedge \alpha \setminus \alpha') && \mid \alpha' \sqsubseteq \alpha \wedge (af_1, af_2) \in \mathsf{split}(af)\} \\
\mathsf{split}(af \wedge \neg\alpha) &= \{(af_1 \wedge \alpha', af_2 \wedge \neg(\alpha \setminus \alpha')), \\
&\qquad (af_1 \wedge \neg\alpha', af_2 \wedge (\alpha \setminus \alpha')) && \mid \alpha' \sqsubseteq \alpha \wedge (af_1, af_2) \in \mathsf{split}(af)\} \\
&\quad \cup \{(af_1 \wedge \alpha', af_2), (af_1, af_2 \wedge \alpha') && \mid \underline{\alpha'} \not\sqsubseteq \underline{\alpha} \wedge (af_1, af_2) \in \mathsf{split}(af)\}
\end{aligned}$$

Example 4.2.4 (Splitting action formulae)**.** Consider splitting the action formula $a|b$:

$$\begin{aligned}
&\mathsf{split}(\mathsf{true} \wedge a|b) \\
&= \{(af_1 \wedge \alpha', af_2 \wedge a|b \setminus \alpha') \mid \alpha' \sqsubseteq a|b \wedge (af_1, af_2) \in \mathsf{split}(\mathsf{true})\} \\
&= \{(\mathsf{true} \wedge \alpha', \mathsf{true} \wedge a|b \setminus \alpha') \mid \alpha' \sqsubseteq a|b\} \\
&= \{(\tau, a|b), (a, b), (b, a), (a|b, \tau)\}
\end{aligned}$$

So, we check for each of these pairs (af_1, af_2) whether the process we are quotienting out, P, can do an action $\alpha \in [\![af_1]\!]\varepsilon$. What is left to check is whether the remainder process can perform an action $\beta \in [\![af_2]\!]\varepsilon$.

Now consider splitting the negated action formula $\neg a|b$:

$$\begin{aligned}
&\mathsf{split}(\mathsf{true} \wedge \neg a|b) \\
&= \{(af_1 \wedge \alpha', af_2 \wedge \neg(a|b \setminus \alpha'))| \ \alpha' \sqsubseteq a|b \wedge (af_1, af_2) \in \mathsf{split}(\mathsf{true})\} \\
&\quad \cup \{(af_1 \wedge \alpha', af_2) \mid \underline{\alpha'} \not\sqsubseteq \underline{a|b} \wedge (af_1, af_2) \in \mathsf{split}(\mathsf{true})\} \\
&= \{(\alpha', \neg(a|b \setminus \alpha')) \mid \alpha' \sqsubseteq a|b\} \\
&\quad \cup \{(\alpha', \mathsf{true}), (\mathsf{true}, \alpha') \mid \underline{\alpha'} \not\sqsubseteq \underline{a|b}\} \\
&= \{(\tau, \neg a|b), (a, \neg b), (b, \neg a), (a|b, \neg\tau), \\
&\quad \cup \{(a|a|b, \mathsf{true}), (a|a|a|b, \mathsf{true}), (a|a|a|a|b, \mathsf{true}), \dots\}
\end{aligned}$$

That last set is of course infinitely large. This means that we cannot enumerate all possible ways in which $\alpha|\beta \in [\![af]\!]\varepsilon$ can be decomposed into $\alpha \in [\![af_1]\!]\varepsilon$ and $\beta \in [\![af_2]\!]\varepsilon$.

As can be seen in the example, there are infinitely many ways in which an action formula containing a negative multi-action $\neg\alpha$ can be split. This means these cannot be enumerated and used in a large disjunction. Fortunately, we do not need to know every af_1 and af_2 explicitly. Inspecting Definition 4.2.1 ($\mathsf{Quotient}$), we observe that we only need to know the following:

- $\alpha_i \in [\![af_1]\!]\varepsilon$ and the corresponding af_2
- $\alpha_i \in [\![af_1]\!]\varepsilon$ and $\tau \in [\![af_2]\!]\varepsilon$ for the corresponding af_2
- $\tau \in [\![af_1]\!]\varepsilon$ and the corresponding af_2

The possibly infinitely large set of action formulae (af_1, af_2) is never explicitly required. We move toward new functions $\mathsf{split2}$ and $\mathsf{Quotient2}$ which do not need to explicitly enumerate these decompositions into af_1 and af_2. This $\mathsf{split2}$ needs an additional argument β, which equals the α_i of a process P. We construct these boolean expressions straight away. The specification of $\mathsf{split2}$ is as follows:

$$\mathsf{split2}(\beta, af) = \{(\mathsf{cond}(\beta, af_1), \mathsf{cond}(\tau, af_2), \mathsf{cond}(\tau, af_1), af_2) \mid (af_1, af_2) \in \mathsf{split}(af)\}$$

What is interesting, is that this *is* a finite set. The many ways in which the case $af \wedge \neg\alpha$ leads to an infinitely large set of af_1's where $\alpha' \not\sqsubseteq \alpha$ and we check whether $\beta \in [\![af_1]\!]\varepsilon$, collapse into only one condition to check: $\beta \not\sqsubseteq \alpha$. This is simply a boolean expression.

The definition of the function $\mathsf{split2} : \mathsf{Act} \times AF \to 2^{\mathsf{BoolExpr} \times \mathsf{BoolExpr} \times \mathsf{BoolExpr} \times AF}$ follows directly from this specification. We define $\mathsf{split2}$ as follows:

Definition 4.2.5 ($\mathsf{split2}$)**.** Let β be a multi-action and af be an action formula in disjunctive normal form. We define the function $\mathsf{split2}$ as follows:

$$
\begin{aligned}
\mathsf{split2}(\beta, \mathsf{true}) &= \{(\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true})\} \\
\mathsf{split2}(\beta, af \wedge b) &= \{(b_1 \wedge b, b_2 \wedge b, b_3 \wedge b, af_2 \wedge b) \mid \\
&\qquad (b_1, b_2, b_3, af_2) \in \mathsf{split2}(\beta, af)\} \\
\mathsf{split2}(\beta, af \wedge \alpha) &= \{(b_1 \wedge \alpha' = \beta, b_2 \wedge \alpha' = \alpha, b_3 \wedge \alpha' = \tau, af_2 \wedge \alpha \setminus \alpha') \mid \\
&\qquad \alpha' \sqsubseteq \alpha \wedge (b_1, b_2, b_3, af_2) \in \mathsf{split2}(\beta, af)\} \\
\mathsf{split2}(\beta, af \wedge \neg\alpha) &= \{(b_1 \wedge \beta \not\sqsubseteq \alpha, b_2, \mathsf{false}, af_2) \mid \\
&\qquad (b_1, b_2, b_3, af_2) \in \mathsf{split2}(\beta, af)\} \\
&\quad \cup \{(b_1 \wedge \alpha' = \beta, b_2 \wedge \alpha' \neq \alpha, b_3 \wedge \alpha' = \tau, af_2 \wedge \neg(\alpha \setminus \alpha')) \mid \\
&\qquad \alpha' \sqsubseteq \alpha \wedge (b_1, b_2, b_3, af_2) \in \mathsf{split2}(\beta, af)\}
\end{aligned}
$$

Finally, we arrive at the assertion $\mathsf{Quotient2}$, which is similar to Definition 4.2.1 ($\mathsf{Quotient}$), with the conditions b_1, b_2, b_3 of $\mathsf{split2}$ directly plugged in for "$\alpha_i(s,e) \in [\![af_1]\!]\varepsilon$", "$\tau \in [\![af_2]\!]\varepsilon$", "$\tau \in [\![af_1]\!]\varepsilon$" in the definition:

Definition 4.2.6 ($\mathsf{Quotient2}$)**.** Let b_1, b_2, b_3 be boolean data expressions, af_2 an action formula, $\mathcal{A}$ an assertion, $P(s)$ a linear process equation with initial state and i a summand of P. We define the assertion $\mathsf{Quotient2}$ as follows:

$$
\begin{aligned}
\mathsf{Quotient2}(b_1, b_2, b_3, af_2, \mathcal{A}, P(s), i) = {} & (\exists e : E_i.c_i(s,e) \wedge b_1 \wedge \langle af_2 \rangle (\mathcal{A}/P(g_i(s,e)))) \\
& \vee (\exists e : E_i.c_i(s,e) \wedge b_1 \wedge b_2 \wedge (\mathcal{A}/P(g_i(s,e)))) \\
& \vee (b_3 \wedge \langle af_2 \rangle (\mathcal{A}/P(s)))
\end{aligned}
$$

So, we define quotienting on assertions as follows:

Definition 4.2.7 (Quotienting on assertions)**.**

$$
\begin{aligned}
X(d)/P(s) &= X'(d,s) \\
b/P(s) &= b \\
(\mathcal{A}_1 \vee \mathcal{A}_2)/P(s) &= \mathcal{A}_1/P(s) \vee \mathcal{A}_2/P(s) \\
(\langle af \rangle \mathcal{A})/P(s) &= \bigvee_{i \in I} \bigvee_{(b_1,b_2,b_3,af_2) \in \mathsf{split2}(\alpha_i, af)} \mathsf{Quotient2}(b_1, b_2, b_3, af_2, \mathcal{A}, P(s), i) \\
(\exists d : D.\mathcal{A})/P(s) &= (\exists d : D.\mathcal{A}/P(s))
\end{aligned}
$$

Using Definition 4.2.7 (Quotienting on assertions), we define quotienting on parameterized modal equation systems:

Definition 4.2.8 (Quotienting on linear process equations)**.** For arbitrary parameterized modal equation system $\mathcal{E}$, we define quotienting the process $P(s_i)$ from $\mathcal{E}$, denoted $\mathcal{E}/P(s_i)$, as follows:

$$
\begin{aligned}
\epsilon/P(s_i) &= \epsilon \\
((\sigma X(d : D) = \mathcal{A})\ \mathcal{E})/P(s_i) &= (\sigma X'(d : D, s' : S) = \mathcal{A}/P(s'))\ (\mathcal{E}/P(s_i))
\end{aligned}
$$

Finally, we define quotienting on top assertions:

Definition 4.2.9 (Quotienting on top assertions)**.** Let $\mathcal{E} \downarrow X(d)$ be a top assertion. We define quotienting the process $P(s_i)$ from $\mathcal{E} \downarrow X(d)$, denoted $(\mathcal{E} \downarrow X(d))/P(s_i)$, as follows:

$$(\mathcal{E} \downarrow X(d))/P(s_i) = (\mathcal{E}/P(s_i)) \downarrow X'(d, s_i)$$

4.3 Buffer example (4)

We return to the running example of the buffer. The quotienting procedure on linear process equations is illustrated by using quotienting to solving the model checking problem

$$B_1(\text{empty}) \models \mathcal{E} \downarrow X,$$

where the one-place buffer B_1 is defined as

$$\begin{aligned} B_1(s : D \cup \{\text{empty}\}) = &\sum_{d:D} s \approx \text{empty} \to \text{in}(d) \;.\; B_1(d) \\ &+ s \not\approx \text{empty} \to \text{out}(s) \;.\; B_1(\text{empty}), \end{aligned}$$

and the top assertion $\mathcal{E} \downarrow X$ expresses that for every data element that is fed to the buffer, that same element will eventually come out again (cf. 3.2 (Buffer example (3))):

$$\begin{pmatrix} \nu X = [\mathsf{true}]X \wedge (\forall m : D \;.\; [\text{in}(m)]Y(m)) \\ \mu Y(m : D) = [\neg\text{out}(m)]Y(m) \wedge \langle \mathsf{true} \rangle \mathsf{true} \end{pmatrix} \downarrow X.$$

As stated, we use quotienting to solve this model checking problem. Using the fact that the deadlock process is the unit element for parallel composition, we can rewrite this problem to

$$B_1(\text{empty}) \,||\, P_\delta \models \mathcal{E} \downarrow X,$$

from which we quotient out $B_1(\text{empty})$, to obtain the equivalent model checking problem

$$\rho(P_\delta) \models (\mathcal{E} \downarrow X)/B_1(\text{empty}).$$

This model checking problem still remains to be solved, which we will deal with later. First, we calculate $(\mathcal{E} \downarrow X)/B_1(\text{empty})$. The state parameter of B_1 becomes a parameter in each of the equations in $\mathcal{E}$, and quotienting distributes over boolean connectives and quantifiers.

$$\begin{aligned} &(\mathcal{E} \downarrow X)/B_1(\text{empty}) \\ &= \begin{pmatrix} \nu X(s : D \cup \{\text{empty}\}) = ([\mathsf{true}]X)/B_1(s) \\ \qquad\qquad\qquad\qquad \wedge (\forall m : D \;.\; ([\text{in}(m)]Y(m))/B_1(s)) \\ \mu Y(m : D, s : D \cup \{\text{empty}\}) = ([\neg\text{out}(m)]Y(m))/B_1(s) \\ \qquad\qquad\qquad\qquad \wedge (\langle \mathsf{true} \rangle \mathsf{true})/B_1(s) \end{pmatrix} \downarrow X(\text{empty}) \end{aligned}$$

From here, the expressions grow rapidly. The entire calculation can be found in Appendix A.

There are four modal operators in $\mathcal{E}$. There are two summands in B_1: one where the buffer is full and one where it is empty. We look at the eight combinations of modal operator and summand, and check if the transition label in that summand matches part of the action formula in the modal operator.

For each of these combinations, we use the function split2 to calculate all possible ways we can split the action formula in two parts; one to match the transition label of the summand, one match a transition in the remainder process, P_δ in this case. This gives 14 occurrences of Quotient2.

Finally, these occurrences of Quotient2 are each unfolded into three cases:

- the summand performs part of an action and the remainder process performs the rest,
- the summand performs the entire action and the remainder process stays idle, or
- B_1 stays idle and the remainder process performs the entire action.

This gives a total of 42 conjuncts, most of which will turn out to equal true and some duplicate conjuncts. In this example, we reduced these by hand. In Section 6, we briefly discuss how to automate reduction of parameterized modal equation systems.

In Appendix A, we calculate that $(\mathcal{E} \downarrow X)/B_1(\text{empty})$ equals the following top assertion:

$$\left(\begin{array}{l}
\nu X(s : D \cup \{\text{empty}\}) = \\
\quad (\forall s' : D \cup \{\text{empty}\}.[\mathsf{true}]X(s') \wedge X(s')) \\
\quad \wedge (\forall m : D\ . \\
\qquad [\text{in}(m)]Y(m, s) \\
\qquad \wedge (s \approx \text{empty} \Rightarrow\ [\tau]Y(m, m) \wedge Y(m, m)) \\
\quad) \\
\mu Y(m : D, s : D \cup \{\text{empty}\}) = \\
\quad [\neg\text{out}(m)]Y(m, s) \\
\quad \wedge (\forall d : D.s \approx \text{empty}\ \Rightarrow\ [\mathsf{true}]Y(m, d) \wedge Y(m, d)) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \not\approx m\ \Rightarrow\ [\mathsf{true}]Y(m, \text{empty}) \wedge Y(m, \text{empty})) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \approx m\ \Rightarrow\ [\neg\tau]Y(m, \text{empty}))
\end{array}\right) \downarrow X(\text{empty})$$

So, our model checking problem is now as follows:

$$\rho(P_\delta) \models \left(\begin{array}{l}
\nu X(s : D \cup \{\text{empty}\}) = \\
\quad (\forall s' : D \cup \{\text{empty}\}.[\mathsf{true}]X(s') \wedge X(s')) \\
\quad \wedge (\forall m : D\ . \\
\qquad [\text{in}(m)]Y(m, s) \\
\qquad \wedge (s \approx \text{empty} \Rightarrow\ [\tau]Y(m, m) \wedge Y(m, m)) \\
\quad) \\
\mu Y(m : D, s : D \cup \{\text{empty}\}) = \\
\quad [\neg\text{out}(m)]Y(m, s) \\
\quad \wedge (\forall d : D.s \approx \text{empty}\ \Rightarrow\ [\mathsf{true}]Y(m, d) \wedge Y(m, d)) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \not\approx m\ \Rightarrow\ [\mathsf{true}]Y(m, \text{empty}) \wedge Y(m, \text{empty})) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \approx m\ \Rightarrow\ [\neg\tau]Y(m, \text{empty}))
\end{array}\right) \downarrow X(\text{empty})$$

The solution to this model checking problem is equal to the solution of our original problem. Interestingly, we can solve this specific model checking problem very easily, because P_δ is such a simple process. As the deadlock process contains no transitions, the rename operator has no effect. Moreover, all box modalities are equivalent to true. From there, it is quickly shown that the model checking problem yields true. Of course, this exercise was intended to show a somewhat large example of the quotienting procedure, not to solve this trivial model checking problem.

4.4 Soundness

In this section, we show that the quotienting procedure on linear process equations in simple parallel composition, is sound. First, we show a relation between recursion variables before and after quotienting on assertions. Second, we use this to establish soundness on parameterized modal equation systems. Third, we prove soundness on top assertions and satisfaction, which follow directly from the second proof.

Relation between recursion variables on assertions

We show that the recursion variables before and after quotienting on assertions, are related.

Lemma 4.4.1 (Relation between recursion variables for quotienting on assertions)**.** For any two environments ρ_1, ρ_2 such that for any $s_1 \in S_{[\![P_1]\!]}$, $s_2 \in S_{[\![P_2]\!]}$ the relation

$$(\forall X \in \mathsf{occ}(\mathcal{A}) \,.\, (s_1, s_2) \in \rho_1(X)(d) \Leftrightarrow s_2 \in \rho_2(X')(d, s_1)) \tag{4.1}$$

holds, it also holds that for any data environment ε,

$$(s_1, s_2) \in [\![\mathcal{A}]\!]_{P_1 \,||\, P_2} \rho_1 \varepsilon(X)(d) \Leftrightarrow s_2 \in [\![\mathcal{A}/P_1(s_1)]\!]_{P_2} \rho_2 \varepsilon$$

Proof. By derivation of the definition of qoutienting on assertions, Definition 4.2.7 (Quotienting on assertions). □

Applicability of Knaster-Tarski's Theorem

It remains to show that the quotienting procedure is sound for parameterized modal equation systems and for top assertions. This leans heavily on Theorem 1.2.6 (Knaster-Tarski's theorem), so we will first establish that it is applicable. We need to show that environments form a complete lattice, and that the semantics of a parameterized modal equation system is a monotone function with respect to these environments.

We define an ordering on environments. Let the ordering $\leq$ on functions be defined as follows: Let $f, g : A \to B$ be functions from arbitrary domain A to an ordered B. Then $f \leq g$ if and only if for all $a \in A$ it holds that $f(a) \leq g(a)$. In our case, these functions are environments and thus $A = D$ and $B = 2^{\mathcal{S}}$. The ordering on $2^{\mathcal{S}}$ is simply $\subseteq$.

Lemma 4.4.2 (Environments are a complete lattice)**.** Environments are a complete lattice, i.e. the set $D \to 2^{\mathcal{S}}$ with pointwise ordering, form a complete lattice.

Proof. The set $2^{\mathcal{S}}$ forms a complete lattice along with the ordering $\subseteq$, as any set ordered by inclusion forms a complete lattice. This means also $D \to 2^{\mathcal{S}}$ ordered by pointwise inclusion is a complete lattice, as its codomain is a complete lattice. □

Lemma 4.4.3 (Assertions are monotone)**.** The semantics of assertions are monotone functions with respect to the environment, i.e. for any assertion $\mathcal{A}$, linear process equation P and data environment ε, if $\rho_1 \leq \rho_2$, then $[\![\mathcal{A}]\!]_P \rho_1 \varepsilon \subseteq [\![\mathcal{A}]\!]_P \rho_2 \varepsilon$.

Proof. By structural induction on $\mathcal{A}$.

- $X(e)$

$$\begin{aligned}
&[\![X(e)]\!]_P \rho_1 \varepsilon \\
&= \rho_1(X([\![e]\!]\varepsilon)) \\
&\subseteq \{\text{assumption}\} \\
&\quad \rho_2(X([\![e]\!]\varepsilon)) \\
&= [\![X(e)]\!]_P \rho_2 \varepsilon
\end{aligned}$$

- b

$$\begin{aligned}
&[\![b]\!]_P \rho_1 \varepsilon \\
&= \begin{cases} \mathsf{Act} & \text{, if } [\![b]\!]\varepsilon \\ \emptyset & \text{, otherwise} \end{cases} \\
&= [\![b]\!]_P \rho_2 \varepsilon
\end{aligned}$$

- $\mathcal{A}_1 \vee \mathcal{A}_2$

$$\begin{aligned}
&[\![\mathcal{A}_1 \vee \mathcal{A}_2]\!]_P \rho_1 \varepsilon \\
&= [\![\mathcal{A}_1]\!]_P \rho_1 \varepsilon \cup [\![\mathcal{A}_2]\!]_P \rho_1 \varepsilon \\
&\subseteq \{\text{Induction Hypothesis}\} \\
&\quad [\![\mathcal{A}_1]\!]_P \rho_2 \varepsilon \cup [\![\mathcal{A}_2]\!]_P \rho_2 \varepsilon \\
&= [\![\mathcal{A}_1 \vee \mathcal{A}_2]\!]_P \rho_2 \varepsilon
\end{aligned}$$

- $\langle af \rangle \mathcal{A}$

$$\begin{aligned}
&[\![\langle af \rangle \mathcal{A}]\!]_P \rho_1 \varepsilon \\
&= \left\{ s \;\middle|\; \bigvee_{i \in I} (\exists e : E_i . c_i(s,e) \wedge \alpha_i(s,e) \in [\![af]\!]\varepsilon \wedge g_i(s,e) \in [\![\mathcal{A}]\!]_P \rho_1 \varepsilon) \right\} \\
&\subseteq \{\text{Induction Hypothesis}\} \\
&\quad \left\{ s \;\middle|\; \bigvee_{i \in I} (\exists e : E_i . c_i(s,e) \wedge \alpha_i(s,e) \in [\![af]\!]\varepsilon \wedge g_i(s,e) \in [\![\mathcal{A}]\!]_P \rho_2 \varepsilon) \right\} \\
&= [\![\langle af \rangle \mathcal{A}]\!]_P \rho_2 \varepsilon
\end{aligned}$$

□

We also show monotonicity of parameterized modal equation systems:

Lemma 4.4.4 (Parameterized modal equation systems are monotone)**.** For any linear process equation P, data environment ε and parameterized modal equation system $\mathcal{E}$, if $\rho_1 \leq \rho_2$, also

$$[\![\mathcal{E}]\!]_P \rho_1 \varepsilon \leq [\![\mathcal{E}]\!]_P \rho_2 \varepsilon.$$

Proof. By structural induction on $\mathcal{E}$.

- ϵ:

$$\begin{aligned}
&[\![\epsilon]\!] \rho_1 \varepsilon \\
&= \rho_1 \varepsilon \\
&\leq \{\text{assumption}\} \\
&\quad \rho_2 \varepsilon \\
&= [\![\epsilon]\!] \rho_2 \varepsilon
\end{aligned}$$

- $(\sigma Y(d : D) = \mathcal{A})\mathcal{E}$:

$$
\begin{aligned}
& [\![(\sigma Y(d : D) = \mathcal{A})\mathcal{E}]\!]_P \rho_1 \varepsilon \\
= \; & [\![\mathcal{E}]\!]_P \rho_1 [Y := \sigma(\lambda F : D \to 2^{\mathcal{S}} \;.\; (\lambda v : D \;.\; [\![\mathcal{A}]\!]_P([\![\mathcal{E}]\!]_P \rho_1 [Y := F]\varepsilon)\varepsilon[d := v]))]\varepsilon \\
\leq \; & \{\text{Induction Hypothesis, Lemma 4.4.3 (Assertions are monotone)}\} \\
& [\![\mathcal{E}]\!]_P \rho_2 [Y := \sigma(\lambda F : D \to 2^{\mathcal{S}} \;.\; (\lambda v : D \;.\; [\![\mathcal{A}]\!]_P([\![\mathcal{E}]\!]_P \rho_2 [Y := F]\varepsilon)\varepsilon[d := v]))]\varepsilon \\
= \; & [\![(\sigma Y(d : D) = \mathcal{A})\mathcal{E}]\!]_P \rho_2 \varepsilon
\end{aligned}
$$

□

From the functions $D \to 2^{\mathcal{S}}$ forming a complete lattice, and the semantics of parameterized modal equation systems being a monotone function, we conclude that Theorem 1.2.6 (Knaster-Tarski's theorem) is applicable for parameterized modal equation systems.

Unbound variables remain equal

We also use the fact that the semantics of a parameterized modal equation system does not change the value of unbound variables. This proof is very simple:

Lemma 4.4.5 (Unbound variables remain equal)**.** For any parameterized modal equation system $\mathcal{E}$, process P, environment ρ and data environment ε, it holds that

$$(\forall X \notin \mathsf{bnd}(\mathcal{E}) \;.\; [\![\mathcal{E}]\!]_P \rho\varepsilon(X) = \rho(X)).$$

Proof. By induction on the structure of $\mathcal{E}$.

- ϵ:

$$
\begin{aligned}
& [\![\epsilon]\!]\rho\varepsilon(X) \\
= \; & \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& \rho(X)
\end{aligned}
$$

- $(\sigma Y(d : D) = \mathcal{A})\mathcal{E}$:
  Induction Hypothesis: for any environment η,

$$[\![\mathcal{E}]\!]_P \eta\varepsilon(X) = \eta(X).$$

$$
\begin{aligned}
& [\![(\sigma Y(d : D) = \mathcal{A})\mathcal{E}]\!]_P \rho\varepsilon(X) \\
= \; & \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& [\![\mathcal{E}]\!]_P \rho[Y := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D.[\![\mathcal{A}]\!]_P \rho[Y := F]\varepsilon[d := v]))]\varepsilon(X) \\
= \; & \{\text{Induction Hypothesis}\} \\
& \rho[Y := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D.[\![\mathcal{A}]\!]_P \rho[Y := F]\varepsilon[d := v]))]\varepsilon(X) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& \rho(X)
\end{aligned}
$$

□

Soundness of quotienting on parameterized modal equation systems

Now for the soundness proof for quotienting on parameterized modal equation systems. We assume a relation between variables which are not bound in $\mathcal{E}$ and prove that the variables which are bound in $\mathcal{E}$ are related in the same way. The theorem that quotienting is sound for top assertions, easily follows from this lemma.

Lemma 4.4.6 (Relating $\mathcal{E}$ and $\mathcal{E}/P$)**.** For parameterized modal equation system $\mathcal{E}$, $s_1 \in S_{\llbracket P_1 \rrbracket}$, $s_2 \in S_{\llbracket P_2 \rrbracket}$, and environments ρ_1, ρ_2 such that it holds that

$$(\forall X \in \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E}) \;.\; (s_1, s_2) \in \rho_1(X)(d) \Leftrightarrow s_2 \in \rho_2(X')(d, s_1)),$$

the following holds also:

$$(\forall X \in \mathsf{bnd}(\mathcal{E}) \;.\; (s_1, s_2) \in (\llbracket \mathcal{E} \rrbracket_{P_1 \,||\, P_2} \rho_1 \varepsilon)(X)(d) \Leftrightarrow s_2 \in (\llbracket \mathcal{E}/P_1(s_1) \rrbracket_{P_2} \rho_2 \varepsilon)(X')(d, s_1))$$

Proof. The proof is rather verbose and we note that the details can be freely skipped.

Proof by induction on the structure of $\mathcal{E}$.

- ϵ: trivial (empty universal quantification).
- $(\sigma Y(d : D) = \mathcal{A})\mathcal{E}$:

Induction Hypothesis: for environments ρ_1, ρ_2 such that for all $s_1 \in S_{\llbracket P_1 \rrbracket}$, $s_2 \in S_{\llbracket P_2 \rrbracket}$ and all $X \in \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$ it holds that

$$(s_1, s_2) \in \rho_1(X)(d) \Leftrightarrow s_2 \in \rho_2(X')(d, s_1), \tag{4.2}$$

it also holds that for all $X \in \mathsf{bnd}(\mathcal{E})$:

$$(s_1, s_2) \in (\llbracket \mathcal{E} \rrbracket_{P_1 \,||\, P_2} \rho_1 \varepsilon)(X)(d) \Leftrightarrow s_2 \in (\llbracket \mathcal{E}/P_1(s_1) \rrbracket_{P_2} \rho_2 \varepsilon)(X')(d, s_1). \tag{4.3}$$

Assume environments ρ_1, ρ_2 for which the assumption (4.2) holds. By Induction Hypothesis, equation (4.3) also holds for these environments. Furthermore, we assume the variables unbound in the next approximation are also related in this ρ_1, ρ_2, i.e. that equation (4.2) holds for all $X \in \mathsf{occ}((\sigma Y(d' : D) = \mathcal{A})\mathcal{E}) \setminus \mathsf{bnd}((\sigma Y(d' : D) = \mathcal{A})\mathcal{E})$. What is to show in the induction step is that for all $X \in \mathsf{bnd}((\sigma Y(d' : D) = \mathcal{A})\mathcal{E})$, it holds that

$$(s_1, s_2) \in (\llbracket (\sigma Y(d' : D) = \mathcal{A})\mathcal{E} \rrbracket_{P_1 \,||\, P_2} \rho_1 \varepsilon)(X)(d)$$
$$\Leftrightarrow s_2 \in (\llbracket (\sigma Y(d' : D) = \mathcal{A})\mathcal{E}/P_1(s_1) \rrbracket_{P_2} \rho_2 \varepsilon)(X')(d, s_1).$$

Take such $X \in \mathsf{bnd}((\sigma Y(d' : D) = \mathcal{A})\mathcal{E})$, call it X_0. There are three cases: X_0 is bound in the first equation, in an equation in $\mathcal{E}$, or neither. The first case is the most interesting, the second we handle using the Induction Hypothesis, and the third we handle using the assumption on unbound variables.

1. $X_0 = Y$

$$\begin{aligned}
& s_2 \in \llbracket ((\sigma Y(d' : D) = \mathcal{A})\mathcal{E})/P_1(s_1) \rrbracket_{P_2} \rho_2 \varepsilon(X_0')(d, s_1) \\
= \;& \{\text{Definition 4.2.8 (Quotienting on linear process equations)}\} \\
& s_2 \in \llbracket (\sigma Y(d' : D, s' : S) = \mathcal{A}/P_1(s'))(\mathcal{E}/P_1(s_1)) \rrbracket_{P_2} \rho_2 \varepsilon(X_0')(d, s_1) \\
= \;& \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& s_2 \in \llbracket \mathcal{E}/P_1(s_1) \rrbracket_{P_2} \rho_2 [Y' := \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad \llbracket \mathcal{A}/P_1(s') \rrbracket_{P_2} (\llbracket \mathcal{E}/P_1(s_1) \rrbracket_{P_2} \rho_2 [Y' := F] \varepsilon [d' := v, s' := s]) \varepsilon))] \varepsilon(X_0')(d, s_1) \\
= \;& \{\text{Lemma 4.4.5 (Unbound variables remain equal), Definition 1.1.1 (Assignment)}\} \\
& s_2 \in \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad \llbracket \mathcal{A}/P_1(s') \rrbracket_{P_2} (\llbracket \mathcal{E}/P_1(s_1) \rrbracket_{P_2} \rho_2 [Y' := F] \varepsilon [d' := v, s' := s]) \varepsilon))(d, s_1)
\end{aligned}$$

We want to show that this fixpoint is equal to the fixpoint before quotienting. We abbreviate these fixpoints with m and m'.

$$s_2 \in m'(d, s_1) = (s_1, s_2) \in m(d), \tag{4.4}$$

where

$$\begin{aligned} m' = \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\ [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon)) \\ m = \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\ [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_2[Y := F]\varepsilon[d' := v])\varepsilon)). \end{aligned}$$

By Theorem 1.2.6 (Knaster-Tarski's theorem), these fixpoints m and m' can be obtained by transfinite induction (cf. Section 1.3). We show that these two fixpoints are equal by proving the stronger statement that they are equal at every point in the induction, i.e. we show the following three statements:

- Base:

$$s_2 \in F'^0(d, s_1) \Leftrightarrow (s_1, s_2) \in F^0(d)$$

- Induction step:

assuming	$s_2 \in F'^n(d, s_1) \Leftrightarrow (s_1, s_2) \in F^n(d)$,
it follows that	$s_2 \in F'^{n+1}(d, s_1) \Leftrightarrow (s_1, s_2) \in F^{n+1}(d)$.

- Transfinite case: for any limit ordinal α,

assuming	$s_2 \in F'^{\beta}(d, s_1) \Leftrightarrow (s_1, s_2) \in F^{\beta}(d)$ for all $\beta < \alpha$,
it follows that	$s_2 \in F'^{\alpha}(d, s_1) \Leftrightarrow (s_1, s_2) \in F^{\alpha}(d)$.

As this induction converges on fixpoint values m and m', this proves equation (4.4).

1.1. Base. The definition of F^0 depends on the sign of σ. We only show $F^0_\mu = (\lambda v : D.\emptyset)$, the case $F^0_\nu = (\lambda v : D.\mathcal{S})$ is dual.

$$\begin{aligned} & s_2 \in F^0(d, s_1) \\ = \; & s_2 \in (\lambda v : D, s : S.\emptyset)(d, s_1) \\ = \; & s_2 \in \emptyset \\ = \; & (s_1, s_2) \in \emptyset \\ = \; & (s_1, s_2) \in (\lambda v : D.\emptyset)(d) \\ = \; & (s_1, s_2) \in F'^0(d) \end{aligned}$$

1.2. Induction step.

Induction Hypothesis:

$$(s_1, s_2) \in F^n(d) = s_2 \in F'^n(d, s_1) \tag{4.5}$$

The next step is obtained by applying the monotone function on the previous step:

$$\begin{aligned} F^{n+1} &= (\lambda v : D.[\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_2[Y' := F^n]\varepsilon[d' := v])\varepsilon) \\ F'^{n+1} &= (\lambda v : D, s : S.[\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := v, s' := s])\varepsilon) \end{aligned}$$

These functions are monotone, by Lemma 4.4.3 (Assertions are monotone) and Lemma 4.4.4 (Parameterized modal equation systems are monotone).

$$
\begin{aligned}
& s_2 \in F'^{n+1}(d, s_1) \\
= & \{\text{next step}\} \\
& s_2 \in (\lambda v : D, s : S.[\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := v, s' := s])\varepsilon)(d, s_1) \\
= & \{\text{function application}\} \\
& s_2 \in [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := d, s' := s_1])\varepsilon \\
\dagger = & \{\text{Lemma 4.4.1 (Relation between recursion variables for quotienting on assertions)}\} \\
& (s_1, s_2) \in [\![\mathcal{A}]\!]_{P_1 \,||\, P_1}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F^n]\varepsilon[d' := d])\varepsilon \\
= & \{\text{function application}\} \\
& (s_1, s_2) \in (\lambda v : D.[\![\mathcal{A}]\!]_{P_1 \,||\, P_1}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F^n]\varepsilon[d' := v]))\varepsilon(d) \\
= & \{\text{next step}\} \\
& (s_1, s_2) \in F^{n+1}(d)
\end{aligned}
$$

What remains to show is that the application of Lemma 4.4.1 (Relation between recursion variables for quotienting on assertions) at step † is valid, i.e. its assumption (4.1) is met.

To show: for all $X \in \mathsf{occ}(\mathcal{E})$:

$$
\begin{aligned}
& s_2 \in [\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := v, s' := s](X)(d, s_1) \\
\Leftrightarrow & (s_1, s_2) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F^n]\varepsilon[d' := v](X)(d, s_1)
\end{aligned}
$$

Take such $X \in \mathsf{occ}(\mathcal{E})$, call it X_1. There are three cases: X_1 is bound in the first equation, in an equation in $\mathcal{E}$, or neither. The first case follows from the Induction Hypothesis of the transfinite induction (4.5), the second from the Induction Hypothesis of the structural induction, and the third follows from the assumption on unbound variables.

1.2.1. $X_1 = Y$

$$
\begin{aligned}
& s_2 \in [\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := v, s' := s](X_1)(d, s_1) \\
= & \{\text{Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& s_2 \in \rho_2[Y' := F'^n](X_1)(d, s_1) \\
= & \{\text{Definition 1.1.1 (Assignment)}\} \\
& s_2 \in F'^n(d, s_1) \\
= & \{(4.5)\} \\
& (s_1, s_2) \in F^n(d) \\
= & \{\text{Definition 1.1.1 (Assignment)}\} \\
& (s_1, s_2) \in \rho_1[Y := F^n](X_1)(d) \\
= & \{\text{Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& (s_1, s_2) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F^n]\varepsilon[d' := v](X_1)(d)
\end{aligned}
$$

1.2.2. $X_1 \in \mathsf{bnd}(\mathcal{E})$

$$
\begin{aligned}
& s_2 \in [\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := v, s' := s](X_1)(d, s_1) \\
\ddagger = & \{\text{Induction Hypothesis}\} \\
& (s_1, s_2) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F^n]\varepsilon[d' := v](X_1)(d)
\end{aligned}
$$

We show that the application of the Induction Hypothesis at ‡ is valid, i.e. that its assumption is met.

To show: for all $X \in \mathsf{occ}(\mathcal{E})$,

$$s_2 \in \rho_2[Y' := F'^n](X)(d, s_1) \Leftrightarrow (s_1, s_2) \in \rho_1[Y := F^n](X)(d)$$

Take such $X \in \mathsf{occ}(\mathcal{E})$, call it X_2. There are only two cases:

1.2.2.1. $X_2 = Y$

This case is nearly identical to the case $X_1 = Y$ above (case 1.2.1):

$$\begin{aligned}
& s_2 \in \rho_2[Y' := F'^n](X_2)(d, s_1) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& s_2 \in F'^n(d, s_1) \\
= \; & \{\text{Induction Hypothesis for transfinite induction (4.5)}\} \\
& (s_1, s_2) \in F^n(d) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& (s_1, s_2) \in \rho_1[Y := F^n](X_2)(d)
\end{aligned}$$

1.2.2.2. $X_2 \neq Y$

$$\begin{aligned}
& s_2 \in \rho_2[Y' := F'^n](X_2')(d, s_1) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& s_2 \in \rho_2(X_2')(d, s_1) \\
= \; & \{\text{assumption on unbound variables (4.2)}\} \\
& (s_1, s_2) \in \rho_1(X_2)(d) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& (s_1, s_2) \in \rho_1[Y := F^n](X_2)(d)
\end{aligned}$$

This means step ‡ is valid and thus concludes the case $X_1 \in \mathsf{bnd}(\mathcal{E})$ (case 1.2.2).

1.2.3. $X_1 \neq Y$ and $X_1 \notin \mathsf{bnd}(\mathcal{E})$

$$\begin{aligned}
& s_2 \in [\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F'^n]\varepsilon[d' := v, s' := s](X_1)(d, s_1) \\
= \; & \{\text{Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& s_2 \in \rho_2[Y' := F'^n](X_1)(d, s_1) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& s_2 \in \rho_2(X_1)(d, s_1) \\
= \; & \{\text{assumption on unbound variables (4.2)}\} \\
& (s_1, s_2) \in \rho_1(X_1)(d) \\
= \; & \{\text{Definition 1.1.1 (Assignment)}\} \\
& (s_1, s_2) \in \rho_1[Y := F^n](X_1)(d) \\
= \; & \{\text{Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& (s_1, s_2) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F^n]\varepsilon[d' := v](X_1)(d)
\end{aligned}$$

This concludes the induction step of the transfinite induction (case 1.2).

1.3. Transfinite case. For limit ordinals α, there are two definitions for F^α, depending on the sign of σ:

- $F^{\alpha}_{\mu} = \mathsf{sup}(\{F^{\beta}_{\mu} \mid \beta < \alpha\})$
- $F^{\alpha}_{\nu} = \mathsf{inf}(\{F^{\beta}_{\nu} \mid \beta < \alpha\})$

We show only the case for μ, the other case is dual. To show: for any limit ordinal α,

assuming	$s_2 \in F'^{\beta}_{\mu}(d, s_1) \Leftrightarrow (s_1, s_2) \in F^{\beta}_{\mu}(d)$ for all $\beta < \alpha$,
it follows that	$s_2 \in F'^{\alpha}_{\mu}(d, s_1) \Leftrightarrow (s_1, s_2) \in F^{\alpha}_{\mu}(d)$.

$$
\begin{aligned}
& s_2 \in F'^{\alpha}_{\mu}(d, s_1) \\
= \; & \{\text{Definition 1.3.3 } (\sigma\text{-approximants})\} \\
& s_2 \in \mathsf{sup}(\{F'^{\beta}_{\mu} \mid \beta < \alpha\})(d, s_1)
\end{aligned}
$$

As the formula being iterated over is monotone, all these F'^{β}_{μ} are comparable. The supremum is defined as the pointwise supremum of the results of these functions. This means this supremum of these functions, applied to a value, is the supremum of these values:

$$
\begin{aligned}
= \; & s_2 \in \mathsf{sup}(\{F'^{\beta}_{\mu}(d, s_1) \mid \beta < \alpha\}) \\
= \; & \{\text{supremum of sets is their intersection}\} \\
& s_2 \in \bigcap_{\beta<\alpha} F'^{\beta}_{\mu}(d, s_1) \\
= \; & \{\text{set theory}\} \\
& \bigwedge_{\beta<\alpha} s_2 \in F'^{\beta}_{\mu}(d, s_1) \\
= \; & \{\text{Induction Hypothesis}\} \\
& \bigwedge_{\beta<\alpha} (s_1, s_2) \in F^{\beta}_{\mu}(d) \\
= \; & \{\text{same steps in reverse}\} \\
& (s_1, s_2) \in F^{\alpha}_{\mu}(d)
\end{aligned}
$$

By transfinite induction, equation (4.4) is valid.

From this, the result for the case $X_0 = Y$ (case 1) easily follows:

$$
\begin{aligned}
& s_2 \in [\![((\sigma Y'(d' : D) = \mathcal{A})\mathcal{E})/P_1(s_1)]\!]_{P_2}\rho_2\varepsilon(X_0')(d, s_1) \\
= \; & \{\text{Definition 4.2.8 (Quotienting on linear process equations)}\} \\
& s_2 \in [\![(\sigma Y'(d' : D, s' : S) = \mathcal{A}/P_1(s'))(\mathcal{E}/P_1(s_1))]\!]_{P_2}\rho_2\varepsilon(X_0')(d, s_1) \\
= \; & \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& s_2 \in [\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))]\varepsilon(X_0')(d, s_1) \\
= \; & \{\text{Lemma 4.4.5 (Unbound variables remain equal), Definition 1.1.1 (Assignment)}\} \\
& s_2 \in \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))(d, s_1) \\
= \; & \{\text{transfinite induction (4.4)}\} \\
& (s_1, s_2) \in \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_2[Y := F]\varepsilon[d' := v])\varepsilon))(d) \\
= \; & \{\text{Definition 1.1.1 (Assignment), Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& (s_1, s_2) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_2[Y := F]\varepsilon[d' := v])\varepsilon))]\varepsilon(X_0)(d) \\
= \; & \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& (s_1, s_2) \in [\![(\sigma Y(d' : D) = \mathcal{A})\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1\varepsilon(X_0)(d)
\end{aligned}
$$

2. $X_0 \in \mathsf{bnd}(\mathcal{E})$:

$$
\begin{aligned}
& s_2 \in [\![((\sigma Y(d' : D) = \mathcal{A})\mathcal{E})/P_1(s_1)]\!]_{P_2}\rho_2\varepsilon(X_0')(d, s_1) \\
= \; & \{\text{Definition 4.2.8 (Quotienting on linear process equations)}\} \\
& s_2 \in [\![(\sigma Y(d' : D, s' : S) = \mathcal{A}/P_1(s'))(\mathcal{E}/P_1(s_1))]\!]_{P_2}\rho_2\varepsilon(X_0')(d, s_1) \\
= \; & \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& s_2 \in [\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))]\varepsilon(d, s_1) \\
\ddagger = \; & \{\text{Induction Hypothesis}\} \\
& (s_1, s_2) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F]\varepsilon[d' := v])\varepsilon))]\varepsilon(d) \\
= \; & \{\text{Definition 3.1.7 (Parameterized modal equation system semantics)}\} \\
& (s_1, s_2) \in [\![(\sigma Y(d' : D) = \mathcal{A})\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1\varepsilon(X_0)(d)
\end{aligned}
$$

It remains to show that the application of the Induction Hypothesis at step † is valid, i.e. that the assumption (4.2) is met. To show: for all $X \in \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$ it holds that

$$
\begin{aligned}
& (s_1, s_2) \in \rho_1[Y := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F]\varepsilon[d' := v])\varepsilon))](X)(d) \\
\Leftrightarrow \; & s_2 \in \rho_2[Y' := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))](X')(d, s_1))
\end{aligned}
$$

Take such an $X \in \mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$, call it X_1. There are two cases: either X_1 is bound in the first equation or it is not bound in $(\sigma Y(d' : D) = \mathcal{A})\mathcal{E}$ at all. The first is equal to the case $X_0 = Y$ (case 1), the second follows from the assumption on unbound variables (4.2).

2.1. $X_1 = Y$
This case is equal to the case $X_0 = Y$ above (case 1):

$$
\begin{aligned}
& s_2 \in \rho_2[Y' := \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))](X_1')(d, s_1)) \\
= & \{\text{Definition 1.1.1 (Assignment)}\} \\
& s_2 \in \sigma(\lambda F : D' \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))(d, s_1) \\
& \{\text{equal to case } X_0 = Y \text{ (case 1)}\} \\
& (s_1, s_2) \in \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F]\varepsilon[d' :=])\varepsilon))](X_1)(d)) \\
= & \{\text{Definition 1.1.1 (Assignment)}\} \\
& (s_1, s_2) \in \rho_1[Y := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F]\varepsilon[d' := v])\varepsilon))](X)(d)
\end{aligned}
$$

2.2. $X_1 \neq Y$ and $X_1 \notin \mathsf{bnd}(\mathcal{E})$

$$
\begin{aligned}
& s_2 \in \rho_2[Y' := \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D, s : S. \\
& \quad [\![\mathcal{A}/P_1(s')]\!]_{P_2}([\![\mathcal{E}/P_1(s_1)]\!]_{P_2}\rho_2[Y' := F]\varepsilon[d' := v, s' := s])\varepsilon))](X_1')(d, s_1) \\
= & \{\text{Definition 1.1.1 (Assignment)}\} \\
& s_2 \in \rho_2(X_1')(d, s_1) \\
= & \{\text{assumption on unbound variables (4.2)}\} \\
& (s_1, s_2) \in \rho_1(X_1)(d) \\
= & \{\text{Definition 1.1.1 (Assignment)}\} \\
& (s_1, s_2) \in \sigma(\lambda F : D \to 2^{\mathcal{S}}.(\lambda v : D. \\
& \quad [\![\mathcal{A}]\!]_{P_1 \,||\, P_2}([\![\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1[Y := F]\varepsilon[d' := v])\varepsilon))(d)
\end{aligned}
$$

From this it follows that using the Induction Hypothesis at step † is valid and thus concludes the case $X \in \mathsf{bnd}(\mathcal{E})$ (case 2).

3. $X_0 \neq Y$ and $X_0 \notin \mathsf{bnd}(\mathcal{E})$

$$
\begin{aligned}
& s_2 \in [\![((\sigma Y(d' : D) = \mathcal{A})\mathcal{E})/P_1(s_1)]\!]_{P_2}\rho_2\varepsilon(X_0')(d, s_1) \\
= & \{\text{Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& s_2 \in \rho_2(X_0')(d, s_1) \\
= & \{\text{assumption on unbound variables (4.2)}\} \\
& (s_1, s_2) \in \rho_1(X_0)(d) \\
= & \{\text{Lemma 4.4.5 (Unbound variables remain equal)}\} \\
& (s_1, s_2) \in [\![(\sigma Y(d' : D) = \mathcal{A})\mathcal{E}]\!]_{P_1 \,||\, P_2}\rho_1\varepsilon(X_0)(d)
\end{aligned}
$$

These three cases prove the induction step. The lemma follows by structural induction on $\mathcal{E}$. □

Soundness of quotienting on top assertions

Finally, we prove soundness of quotienting on top assertions, which follows easily from Lemma 4.4.6 (Relating $\mathcal{E}$ and $\mathcal{E}/P$):

Theorem 4.4.7 (Quotienting on linear process equations is sound)**.** Quotienting on linear process equations is sound with respect to satisfaction. For any closed parameterized modal equation system $\mathcal{E}$ and any $X \in \mathsf{bnd}(\mathcal{E})$, it holds that

$$P_1(s_{i1}) \,||\, P_2(s_{i2}) \models \mathcal{E} \downarrow X(d), \quad \text{if and only if,} \quad P_2(s_{i2}) \models (\mathcal{E} \downarrow X(d))/P_1(s_{i1})$$

Proof. Since $\mathcal{E}$ is closed, the assumption of Lemma 4.4.6 (Relating $\mathcal{E}$ and $\mathcal{E}/P$) trivially holds.

$$\begin{aligned}
& P_2(s_{i2}) \models (\mathcal{E} \downarrow X(d))/P_1(s_{i1}) \\
= \; & \{\text{Definition 4.2.9 (Quotienting on top assertions)}\} \\
& P_2(s_{i2}) \models (\mathcal{E}/P_1(s_{i1})) \downarrow X'(d, s_{i1}) \\
= \; & \{\text{Definition 3.1.11 (Top assertion semantics)}\} \\
& s_{i2} \in [\![\mathcal{E}/P_1(s_{i1})]\!]_{P_2} \rho\varepsilon(X')(d, s_{i1}) && \text{for arbitrary } \rho, \varepsilon \\
= \; & \{\text{Lemma 4.4.6 (Relating } \mathcal{E} \text{ and } \mathcal{E}/P)\} \\
& (s_{i1}, s_{i2}) \in [\![\mathcal{E}]\!]_{P_1 \,||\, P_2} \rho\varepsilon(X)(d) && \text{for arbitrary } \rho, \varepsilon \\
= \; & \{\text{Definition 3.1.11 (Top assertion semantics)}\} \\
& P_1(s_{i1}) \,||\, P_2(s_{i2}) \models \mathcal{E} \downarrow X(d)
\end{aligned}$$

□

5 Quotienting extensions

In this section, we extend quotienting to process operators. What we are quotienting, is the actual operators themselves. For example, we define $\mathcal{E}/\Gamma_C$, such that $\Gamma_C(Q) \models \mathcal{E}$, if and only if, $Q \models \mathcal{E}/\Gamma_C$. Similarly for ∇_V, ρ_C and τ_H. This enabled decomposition of process description to their atomic linear process equations.

However, those definitions would not be complete. For example, quotienting the communication operators from the process $\Gamma_{C_1}(P_1) \,||\, \Gamma_{C_2}(P_2)$ would not be possible with this definition. We generalize the problem to finding a quotienting rule with the following specification:

$$\Gamma_C(Q_1) \,||\, Q_2 \models \mathcal{E}, \text{ if and only if, } Q_1 \,||\, \rho(Q_1) \models \mathcal{E}/\Gamma_C$$

The rationale behind this, we explain in Section 5.1.

Then, for each process operator, we derive the quotienting rule from this specification. Ideally, we would like this quotienting to be completely independent from Q, but it turns out that this is not possible.

The proof that quotienting is sound on parameterized modal equation system, follows from their soundness on assertions, as with quotienting on parallel linear process equations (cf. Section 4.4). We need to derive the definitions of quotienting on assertions. Because these operators only act on the labels of transitions or restrict what transitions can be taken, the only interesting cases are $\langle af \rangle \mathcal{A}$ and $[af]\mathcal{A}$. We derive the quotienting step for $\langle af \rangle \mathcal{A}$; the case $[af]\mathcal{A}$ is dual.

5.1 Quotienting process descriptions

We transform Grammar 2.6.1 (Process descriptions) to the following equivalent grammar, noting that the deadlock process P_δ is the unit element of parallel composition (cf. Corollary 2.7.6 (Parallel composition properties on linear process equations)):

Grammar 5.1.1 (Process descriptions, alternative grammar)**.** Let Q be a process description and let P be a linear process equation.

$$\begin{aligned}
Q ::= {} & P_\delta(i_{P_\delta}) && (1)\\
| \; & P(s) \,||\, Q && (2)\\
| \; & (Q \,||\, Q) \,||\, Q && (3)\\
| \; & \Gamma_C(Q) \,||\, Q && (4)\\
| \; & \nabla_V(Q) \,||\, Q && (5)\\
| \; & \rho_C(Q) \,||\, Q && (6)\\
| \; & \tau_H(Q) \,||\, Q && (7)
\end{aligned}$$

Here, C is a set of substitutions (cf. Definition 2.8.1 (Substitution function)) and V and I are sets of multi-action names.

By quotienting, we aim to reduce the complexity of the process description, which will increase the complexity of the property to be checked on it. A complete set of rules would enable the reduction of any process description to the base case, the deadlock process. So, we derive a quotienting rule for each of the other production rules in Grammar 5.1.1 (Process descriptions, alternative grammar). In Section 4.2, we already established quotienting for production rule (2). By the associativity of the parallel operator, Corollary 2.7.6 (Parallel composition properties on linear process equations), the quotienting rule for production rule (3) is trivial. In the following sections, we derive the quotienting rules for production rules (4) through (7).

The general pattern for these quotienting rules is as follows: let $O \in \{\Gamma_C, \nabla_V, \rho_C, \tau_H\}$ be a unary operator, with corresponding production rule $O(Q_1) \,||\, Q_2$. When quotienting the operator O, we need to ignore the parts of the multi-actions that come from Q_2, as they are not affected by that operator. To do this, we rename all action names in Q_2 to fresh action names. We remove this rename operator later. This leads to the following pattern:

$$O(Q_1) \,||\, Q_2 \models \mathcal{E} \downarrow X(d) \qquad \Leftrightarrow \qquad Q_1 \,||\, \rho(Q_2) \models (\mathcal{E} \downarrow X(d))/O$$

Before we define the quotient operator $/$ for each of the operators $\Gamma_C, \nabla_V, \rho_C$ and τ_H, we prove that a set of quotienting rules that follow this specification is sound and complete.

The soundness of quotienting on process descriptions is a direct consequence of the specification. In the following sections, we derive a quotienting rule from a different specification:

$$s \in [\![\mathcal{A}]\!]_{O(Q_1) \,||\, Q_2 \rho\varepsilon} \qquad \Leftrightarrow \qquad s \in [\![\mathcal{A}/O]\!]_{Q_1 \,||\, \rho(Q_2) \rho\varepsilon}$$

That quotienting rules which follow this specification, also obey the previous specification on satisfaction, can be shown very similarly to the soundness proof given in Section 4.4, using this specification instead of Lemma 4.4.1 (Relation between recursion variables for quotienting on assertions). Therefore, we omit the full proof here.

Theorem 5.1.2 (Quotienting on process descriptions is sound)**.** Let $\mathcal{E} \downarrow X(d)$ be a top assertion and $O(Q_1) \,||\, Q_2$ a process description, where $O \in \{\Gamma_C, \nabla_V, \rho_C, \tau_H\}$. Then,

$$O(Q_1) \,||\, Q_2 \models \mathcal{E} \downarrow X(d), \qquad \text{if and only if,} \qquad Q_1 \,||\, \rho(Q_2) \models (\mathcal{E} \downarrow X(d))/O.$$

Proof. Similar to the soundness proof given in Section 4.4, using soundness of quotienting the process operators, to be shown in Sections 5.2 through 5.5. □

We now show the completeness of quotienting on process descriptions:

Theorem 5.1.3 (Quotienting on process descriptions is complete)**.** Any model checking problem with process description Q and some property, can be rewritten to an equivalent problem with process description P_δ and some new property, in finitely many quotienting steps.

Proof. By induction on the number of operators and linear process equations in the left process of parallel composition. The process description can be one of the following forms, given by Grammar 5.1.1 (Process descriptions, alternative grammar):

- Base. P_δ: trivial, no rewriting steps necessary.
- Induction step. There are several induction steps:
  - $P(s) \,||\, Q$: quotienting $P(s)$ leads to one less linear process equation on the left process of this parallel composition.
  - $(Q_1 \,||\, Q_2) \,||\, Q_3$: by associativity of parallel composition, Corollary 2.7.6 (Parallel composition properties on linear process equations), this can be rewritten to $Q_1 \,||\, (Q_2 \,||\, Q_3)$. This leads to strictly fewer operators and linear process equations in the left process of this parallel composition, namely the operators and linear process equations of Q_2.
  - $O(Q_1) \,||\, Q_2$ for $O \in \{\Gamma_C, \nabla_V, \rho_C, \tau_H\}$. Quotienting O yields a property to be checked on $Q_1 \,||\, \rho(Q_2)$. The left process of this parallel composition contains one less operator, namely the operator O.

By induction, there are no operators or linear process equations in the left process of a parallel composition, meaning there is no meaningful parallelism at all. The process that remains, is of the form $\rho(\ldots(\rho(P_\delta)))$, which equals the deadlock process P_δ itself. □

We now define the quotient operator $/$ for each of the operators $\Gamma_C, \nabla_V, \rho_C$ and τ_H.

5.2 Quotienting the communication operator

We derive a quotient step for the communication operator from the following specification:

$$s \in [\![\mathcal{A}]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \Leftrightarrow s \in [\![\mathcal{A}/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon$$

The derivation is by structural induction on the assertion $\mathcal{A}$.

The interesting case is that of the modal operator $\langle\cdot\rangle$. For the other cases, the derivation is rather similar to the derivation of Definition 4.2.7 (Quotienting on assertions).

- $X(e)$

$$\begin{aligned}
& s \in [\![X(e)]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in \rho(X)([\![e]\!]\varepsilon) \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in [\![X(e)/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon
\end{aligned}$$

- b

$$\begin{aligned}
& s \in [\![b]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& [\![b]\!]\varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in [\![b/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon
\end{aligned}$$

- $\mathcal{A}_1 \vee \mathcal{A}_2$

$$\begin{aligned}
& s \in [\![\mathcal{A}_1 \vee \mathcal{A}_2]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in [\![\mathcal{A}_1]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \cup [\![\mathcal{A}_1]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \\
= \; & \{\text{Induction Hypothesis, twice}\} \\
& s \in [\![\mathcal{A}_1/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon \cup [\![\mathcal{A}_2/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in [\![(\mathcal{A}_1/\Gamma_C) \vee (\mathcal{A}_2/\Gamma_C)]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon
\end{aligned}$$

- $(\exists d : D.\mathcal{A})$

$$\begin{aligned}
& s \in [\![(\exists d : D \;.\; \mathcal{A})]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in \bigcup_{v:D} [\![\mathcal{A}]\!]_{\Gamma_C(P_1) \,||\, P_2}\rho\varepsilon[d := v] \\
= \; & \{\text{Induction Hypothesis, set theory}\} \\
& s \in \bigcup_{v:D} [\![\mathcal{A}/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon[d := v] \\
= \; & \{\text{Definition 3.1.3 (Assertion semantics)}\} \\
& s \in [\![(\exists d : D \;.\; \mathcal{A}/\Gamma_C)]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon
\end{aligned}$$

- $\langle af \rangle \mathcal{A}$

This is the interesting case. The communication operator applies the corresponding substitution operator to all transition labels. So, we try to find an action formula such that when the substitution operator is applied to all elements in its semantics, we get the semantics of the original action formula af. Formally, we define an action formula af/Γ_C such that

$$\gamma_C(\alpha) \in [\![af]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \qquad \Leftrightarrow \qquad \alpha \in [\![af/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon.$$

Quotienting for $\langle af \rangle \mathcal{A}$ is then as follows:

$$(\langle af \rangle \mathcal{A})/\Gamma_C = \langle af/\Gamma_C \rangle (\mathcal{A}/\Gamma_C)$$

Assume C contains a single substitution $\beta \to \beta'$. This can always be established using the rule that $\Gamma_{C_1 \cup C_2}(P) = \Gamma_{C_1}(\Gamma_{C_2}(P))$. We derive quotienting the communication operator on action formulae, by structural induction on the action formula af:

- true

$$\begin{aligned}
&\gamma_C(\alpha) \in [\![\mathsf{true}]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\mathsf{true} \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\alpha \in [\![\mathsf{true}]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon
\end{aligned}$$

- $af \wedge b$

$$\begin{aligned}
&\gamma_C(\alpha) \in [\![af \wedge b]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\gamma_C(\alpha) \in [\![af]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \cap [\![b]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\gamma_C(\alpha) \in [\![af]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \cap \begin{cases} \mathsf{Act} & \text{if } [\![b]\!]\varepsilon \\ \emptyset & \text{otherwise} \end{cases} \\
=\ &\{\text{Induction Hypothesis, Definition 2.2.13 (Action formulae semantics)}\} \\
&\alpha \in [\![af/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon \cap [\![b]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon \\
=\ &\{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\alpha \in [\![(af/\Gamma_C) \wedge b]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon
\end{aligned}$$

- $af \wedge \alpha$

For this case, we first construct an action formula that matches the multi-action α *before* the communication. This function is built up similarly to the function CAE in Definition 4.2.7 (Quotienting on assertions). For the multi-action α-action to be present in a process $\Gamma_{\{\beta \to \beta'\}}(P)$, we distinguish the case where an α-transition was already present in P and zero substitutions occurred, and the recursive case where α is the result of one or more substitutions occurring on some other transition in P.

For the case where there were no transitions, the α-transition must have been left untouched by the substitution operator. Using the function choose from Definition 4.2.7 (Quotienting on assertions), this can be expressed as the following action formula:

$$\bigwedge_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \neg(e_1 \approx \ldots \approx e_{|\beta|})$$

For the recursive case, the substitution must have a resulting β' with matching parameters contained in α. This can be expressed as

$$\bigvee_{\substack{(a_1(e_1)|\ldots|a_{|\beta'|}(e_{|\beta'|}),\alpha')\in \mathsf{choose}_{|\beta'|}(\alpha)\\ a_1|\ldots|a_{|\beta'|}=\beta'}} e_1 \approx \ldots \approx e_{|\beta'|}$$

This is the *recursive* case, because in the remainder α' there can again be a substitution result. No recursion is performed on $\beta(e_1)$, the part of the transition label before the substitution, because it cannot be the result of another substitution: substitution results cannot be substituted again.

So, we recursively define the function AF as follows:

$$\mathsf{AF}_{\beta,\beta'}(\alpha,\gamma) = \left(\left(\bigwedge_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha')\in \mathsf{choose}_{|\beta|}(\alpha)\\ a_1|\ldots|a_{|\beta|}=\beta}} \neg(e_1 \approx \ldots \approx e_{|\beta|})\right) \wedge \gamma|\alpha\right) \vee \bigvee_{\substack{(a_1(e_1)|\ldots|a_{|\beta'|}(e_{|\beta'|}),\alpha')\in \mathsf{choose}_{|\beta'|}(\alpha)\\ a_1|\ldots|a_{|\beta'|}=\beta'}} \left(e_1 \approx \ldots \approx e_{|\beta'|} \wedge \mathsf{AF}_{\beta,\beta'}(\alpha',\gamma|\beta(e_1))\right)$$

Initially, γ is the empty multi-action τ and α is the α of the original action formula $af \wedge \alpha$.

Using this function, we can now define the quotienting operator for $af \wedge \alpha$ as follows:

$$\begin{aligned}
&\gamma_C(\xi) \in [\![af \wedge \alpha]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \\
&= \{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\gamma_C(\xi) \in [\![af]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \cap [\![\alpha]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon
\end{aligned}$$

Part of α was performed by $\Gamma_C(P_1)$, part by P_2. These are ζ and η, respectively. We create a disjunction over all these possibilities. Since we rename all actions in P_2, we also need to rename η. Conveniently, as AF only passes its γ parameter on, we can use $\rho(\eta)$ as the initial value for γ.

$$\begin{aligned}
&= \{\text{Induction Hypothesis, splitting } \alpha \text{ into } \zeta \text{ and } \eta\} \\
&\xi \in [\![af/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon \cap \left[\!\!\left[\bigvee_{\zeta|\eta=\alpha} \mathsf{AF}(\zeta,\rho(\eta)) \right]\!\!\right]_{P_1 \,||\, \rho(P_2)} \varepsilon \\
&= \{\text{Definition 2.2.13 (Action formulae semantics)}\} \\
&\xi \in \left[\!\!\left[(af/\Gamma_C) \wedge \bigvee_{\zeta|\eta=\alpha} \mathsf{AF}_{\beta,\beta'}(\zeta,\rho(\eta)) \right]\!\!\right]_{P_1 \,||\, \rho(P_2)} \varepsilon
\end{aligned}$$

- $af \wedge \neg\alpha$

  This case is similar to the case $af \wedge \alpha$:

$$\begin{aligned}
&\gamma_C(\xi) \in [\![af \wedge \neg\alpha]\!]_{\Gamma_C(P_1) \,||\, P_2}\varepsilon \\
&= \{\text{similar to the case for } af \wedge \alpha\} \\
&\xi \in [\![(af/\Gamma_C) \wedge \neg \bigvee_{\zeta|\eta=\alpha} \mathsf{AF}_{\beta,\beta'}(\zeta,\rho(\eta))]\!]_{P_1 \,||\, \rho(P_2)}\varepsilon
\end{aligned}$$

So, we define quotienting the communication operator as follows:

Definition 5.2.1 (Quotienting the communication operator)**.** Let $\mathcal{A}$ be an assertion and let C be a set of substitutions. Then, quotienting out the operator Γ_C from assertion $\mathcal{A}$, denoted $\mathcal{A}/\Gamma_C$, is defined as follows:

If C is not a singleton:

$$\mathcal{A}/\Gamma_{C_1 \cup C_2} = \mathcal{A}/\Gamma_{C_1}/\Gamma_{C_2}$$

If C is a singleton $\beta \to \beta'$:

$$\begin{aligned} X(e)/\Gamma_C &= X(e) \\ b/\Gamma_C &= b \\ (\mathcal{A}_1 \vee \mathcal{A}_2)/\Gamma_C &= (\mathcal{A}_1/\Gamma_C) \vee (\mathcal{A}_2/\Gamma_C) \\ (\exists d : D.\mathcal{A})/\Gamma_C &= (\exists d : D.\mathcal{A}/\Gamma_C) \\ (\langle af \rangle \mathcal{A})/\Gamma_C &= \langle af/\Gamma_C \rangle (\mathcal{A}/\Gamma_C) \end{aligned}$$

Here, quotienting the communication operator on modal operators is defined as follows:

$$\begin{aligned} \mathsf{true}/\Gamma_C &= \mathsf{true} \\ (af \wedge b)/\Gamma_C &= af/\Gamma_C \wedge b \\ (af \wedge \alpha)/\Gamma_C &= af/\Gamma_C \wedge \bigvee_{\zeta|\eta=\alpha} \mathsf{AF}_{\beta,\beta'}(\zeta, \rho(\eta)) \\ (af \wedge \neg\alpha)/\Gamma_C &= af/\Gamma_C \wedge \neg \bigvee_{\zeta|\eta=\alpha} \mathsf{AF}_{\beta,\beta'}(\zeta, \rho(\eta)), \end{aligned}$$

where the action formula AF is defined as:

$$\begin{aligned} \mathsf{AF}_{\beta,\beta'}(\alpha,\gamma) = &\left(\left(\bigwedge_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \neg(e_1 \approx \ldots \approx e_{|\beta|}) \right) \wedge \gamma|\alpha \right) \\ &\vee \bigvee_{\substack{(a_1(e_1)|\ldots|a_{|\beta'|}(e_{|\beta'|}),\alpha') \in \mathsf{choose}_{|\beta'|}(\alpha) \\ a_1|\ldots|a_{|\beta'|}=\beta'}} \left(e_1 \approx \ldots \approx e_{|\beta'|} \wedge \mathsf{AF}_{\beta,\beta'}(\alpha', \gamma|\beta(e_1))\right) \end{aligned}$$

Theorem 5.2.2 (Quotienting the communication operator is sound)**.** Quotienting the communication operator is sound, i.e. for any assertion $\mathcal{A}$, processes P_1 and P_2, set of substitutions C, environment ρ, data environment ε and state s, it holds that

$$s \in [\![\mathcal{A}]\!]_{\Gamma_C(P_1) \,||\, P_2} \rho\varepsilon \Leftrightarrow s \in [\![\mathcal{A}/\Gamma_C]\!]_{P_1 \,||\, \rho(P_2)} \rho\varepsilon.$$

Proof. By derivation of Definition 5.2.1 (Quotienting the communication operator). □

Example 5.2.3 (Quotienting the communication operator)**.** Consider the process R of Example 2.8.6 (Communication operator on linear process equations), and the assertion "a $\mathsf{transfer}(\mathsf{message})$ action is enabled": $\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}$. This model checking problem can be expressed as $s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}]\!]_{\Gamma_{\{\mathsf{send}|\mathsf{receive} \to \mathsf{transfer}\}}(P \,||\, Q)} \rho\varepsilon$, for some environment ρ and data environment ε.

We rewrite this problem slightly, so that it fits the quotienting rule in Theorem 5.2.2 (Quotienting the communication operator is sound):

$$\begin{aligned} & s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}]\!]_{\Gamma_{\{\mathsf{send}|\mathsf{receive} \to \mathsf{transfer}\}}(P \,||\, Q)} \rho\varepsilon \\ = \; & \{\text{deadlock is unit element of parallel composition}\} \\ & s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}]\!]_{\Gamma_{\{\mathsf{send}|\mathsf{receive} \to \mathsf{transfer}\}}(P \,||\, Q) \,||\, P_\delta} \rho\varepsilon \end{aligned}$$

Now, we apply Theorem 5.2.2 (Quotienting the communication operator is sound) and obtain:

$$
\begin{aligned}
&= s \in [\![(\langle \mathsf{transfer}(\mathsf{message})\rangle \mathsf{true})/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}}]\!]_{(P \,||\, Q) \,||\, \rho(P_\delta)} \rho\varepsilon \\
&= \{\text{deadlock is unit element of parallel composition}\} \\
&\quad s \in [\![(\langle \mathsf{transfer}(\mathsf{message})\rangle \mathsf{true})/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}}]\!]_{P \,||\, Q} \rho\varepsilon
\end{aligned}
$$

We apply Definition 5.2.1 (Quotienting the communication operator):

$$
\begin{aligned}
&(\langle \mathsf{transfer}(\mathsf{message})\rangle \mathsf{true})/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}} \\
&= \langle \mathsf{transfer}(\mathsf{message})/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}}\rangle \mathsf{true}/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}} \\
&= \langle (\mathsf{true} \wedge \mathsf{transfer}(\mathsf{message}))/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}}\rangle \mathsf{true} \\
&= \langle \mathsf{true}/\Gamma_{\{\mathsf{send}|\mathsf{receive}\to\mathsf{transfer}\}} \wedge \bigvee_{\zeta|\eta=\mathsf{transfer}(\mathsf{message})} \mathsf{AF}_{\mathsf{send}|\mathsf{receive},}(\zeta|\rho(\eta))\rangle \mathsf{true} \\
&= \langle \mathsf{true} \wedge (\mathsf{AF}_{\mathsf{send}|\mathsf{receive},\mathsf{transfer}}(\mathsf{transfer}(\mathsf{message}), \tau) \vee \\
&\qquad (\mathsf{AF}_{\mathsf{send}|\mathsf{receive},transfer}(\tau, \mathsf{transfer}'(\mathsf{message}))))\rangle \mathsf{true} \\
&= \langle transfer(\mathsf{message}) \vee \mathsf{AF}_{\mathsf{send}|\mathsf{receive},\mathsf{transfer}}(\tau, \mathsf{send}(\mathsf{message})|\mathsf{receive}(\mathsf{message})) \vee \\
&\qquad \mathsf{transfer}'(\mathsf{message})\rangle \mathsf{true} \\
&= \langle \mathsf{transfer}(\mathsf{message}) \vee \mathsf{send}(\mathsf{message})|\mathsf{receive}(\mathsf{message}) \vee \mathsf{transfer}'(\mathsf{message})\rangle \mathsf{true}
\end{aligned}
$$

These disjuncts correspond neatly to the case where

- $\mathsf{transfer}(\mathsf{message})$ was already enabled in $P \,||\, Q$,
- $\mathsf{send}(\mathsf{message})|\mathsf{receive}(\mathsf{message})$ was enabled and communicated to $\mathsf{transfer}(\mathsf{message})$ (this can also be satisfied by some transition $\mathsf{send}(e_1)|\mathsf{receive}(e_2)$ and $e_1 \approx e_2 \approx \mathsf{message}$),
- $\mathsf{transfer}(\mathsf{message})$ was already enabled in P_δ.

Even though it is obvious that this last scenario cannot occur, we must keep this option open, as we wanted to avoid inspecting the underlying processes.

Finally, we reach the following equivalent model checking problem:

$$
s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \vee \mathsf{send}(\mathsf{message})|\mathsf{receive}(\mathsf{message}) \vee \mathsf{transfer}'(\mathsf{message})\rangle \mathsf{true}]\!]_{P \,||\, Q} \rho\varepsilon
$$

Another interesting example is the following:

Consider the same process R, with the assertion $\langle \mathsf{send}(\mathsf{message1})|\mathsf{receive}(\mathsf{message2})\rangle \mathsf{true}$, yields the following after quotienting:

$$
\begin{aligned}
s \in [\![\langle &(\mathsf{message1} \not\approx \mathsf{message2} \wedge \mathsf{send}(\mathsf{message1})|\mathsf{receive}(\mathsf{message2})) \vee \\
&\mathsf{send}(\mathsf{message1})|\mathsf{receive}'(\mathsf{message2}) \vee \\
&\mathsf{send}'(\mathsf{message1})|\mathsf{receive}(\mathsf{message2}) \vee \\
&\mathsf{send}'(\mathsf{message1})|\mathsf{receive}'(\mathsf{message2})\rangle \mathsf{true}]\!]_{P \,||\, Q} \rho\varepsilon
\end{aligned}
$$

The condition $\mathsf{message1} \not\approx \mathsf{message2}$ is necessary, as otherwise the label $\mathsf{send}(\mathsf{message1})|\mathsf{receive}(\mathsf{message2})$ would have communicated to a $transfer(\mathsf{message1})$ action and hence not satisfied the original assertion $\langle \mathsf{send}(\mathsf{message1})|\mathsf{receive}(\mathsf{message2})\rangle \mathsf{true}$.

5.3 Quotienting the allow operator

The interesting case is the one for the modal operator. For all other assertions, the derivation is identical to the derivation of Definition 5.2.1 (Quotienting the communication operator). The specification for this case is as follows:

$$s \in [\![(\langle af\rangle\mathcal{A})/\nabla_V]\!]_{P_1 \,||\, \rho(P_2)}\rho\varepsilon \Leftrightarrow s \in [\![\langle af\rangle\mathcal{A}]\!]_{\nabla_V(P_1) \,||\, P_2}\rho\varepsilon$$

The case $\langle \mathsf{true}\rangle\mathcal{A}$

The base case, where af equals true, is different from quotienting the other operators. As the allow operator actually prevents transitions from occurring, the set of actions that can occur after the allow operator, is only the actions in the allow set. So when we quotient out the allow operator, we only look at the actions that are allowed. For example, for the problem $\nabla_{\{a\}}(P) \models \langle\mathsf{true}\rangle\mathsf{true}$, it is not sufficient for P to be able to do a b-action, as it is blocked by the allow operator. Therefore, we restrict true in the modal operator to actions that are allowed. Because these multi-actions have parameters, we need a quantifier over the parameter sort of each action.

Moreover, these actions can occur simultaneously with actions from P_2. Consider

$$\nabla_{\{a\}}(P_1) \,||\, P_2 \models \langle\mathsf{true}\rangle\mathcal{A}.$$

It is not only sufficient for $\mathcal{A}$ to hold after some a action, but it may also hold after some $a|\alpha$, where α is an action of P_2. To distinguish between this α of P_2 and α's from P_1 that would be blocked, we use the renaming of multi-actions in P_2. Unfortunately, all this implies we have to enumerate all such possible multi-actions α that P_2 can perform, thus we need to inspect P_2. Small comfort is that we do not need to know *precisely* what actions P_2 can perform, we can do with an overapproximation.

Construct the set $Act(Q)$ of possible actions of Q. This is an overapproximation of the actual actions which can be performed by Q. These actions can be open expressions in the data parameters, e.g. $read(d)$.

Definition 5.3.1 (Overapproximation of multi-actions of a process description)**.** Let Q be a process description. The set $Act(Q)$ is an overapproximation of the multi-actions Q can perform.

$$\begin{aligned}
Act(P(s)) &= \{\tau\} \cup \bigvee_{i\in I} \alpha_i \\
Act(Q_1 \,||\, Q_2) &= Act(Q_1) \cup Act(Q_2) \cup \{\alpha|\beta \mid \alpha \in Act(Q_1), \beta \in Act(Q_2)\} \\
Act(\Gamma_C(Q)) &= \begin{cases} Act(\Gamma_{C_1}(\Gamma_{C_2}(Q))) & \text{if } C \text{ is not a singleton, } C = C_1 \cup C_2 \\ \{\Gamma Act_{\beta,\beta'}(\alpha,\tau) \mid \alpha \in Act(Q)\} & \text{if } C \text{ is a singleton } \beta \to \beta' \end{cases} \\
Act(\nabla_V(Q)) &= \{\alpha \in Act(Q) \mid \underline{\alpha} \in V \cup \{\tau\}\} \\
Act(\tau_H(Q)) &= \{\tau_H(\alpha) \mid \alpha \in Act(Q)\},
\end{aligned}$$

For the communication operator, we define the set ΓAct as follows:

$$\Gamma Act_{\beta,\beta'}(\alpha,\gamma) = \{\alpha|\gamma\} \cup \bigcup_{\substack{(a_1(e_1)|\ldots|a_{|\beta|}(e_{|\beta|}),\alpha') \in \mathsf{choose}_{|\beta|}(\alpha) \\ a_1|\ldots|a_{|\beta|}=\beta}} \Gamma Act_{\beta,\beta'}(\alpha', \gamma|\beta'(e_1))$$

The set $Act(Q)$ is clearly finite and all multi-actions that are enabled in any state in Q, are also present in $Act(Q)$.

We now have all the ingredients to quotient the allow operator for the assertion $\langle \mathsf{true} \rangle \mathcal{A}$:

- In the quotienting rule

$$s \in [\![(\langle af \rangle \mathcal{A}) /_{Act(P_2)} \nabla_V]\!]_{P_1 \,||\, \rho(P_2)} \rho\varepsilon \Leftrightarrow s \in [\![\langle af \rangle \mathcal{A}]\!]_{\nabla_V(P_1) \,||\, P_2} \rho\varepsilon,$$

  we calculate the set $Act(P_2)$ and pass it as a parameter to the quotient operator.

- In quotienting the assertion $\langle \mathsf{true} \rangle \mathcal{A}$, we use quantifiers over the possible parameters of the multi-action names in the allow set:

$$(\langle af \rangle \mathcal{A}) /_A \nabla_V = (\exists d_{v_1} : D_{v_1}, \ldots, d_{v_n} : D_{v_n} \;.\; \langle af /_A \nabla_V \rangle (\mathcal{A} /_A \nabla_V)),$$

  where D_v denotes the data parameter sort of the multi-action v for each $v \in V$.

- In quotienting the action formula true, we use a disjunction over all these multi-action names in the allow set, with the data parameter variables of the surrounding quantifier. These multi-actions can occur simultaneously with the actions of P_2, which we identified in the set $Act(P_2)$:

$$\mathsf{true} /_A \nabla_V = \bigvee_{v \in V} \bigvee_{\alpha \in A} v(d_v) | \rho(\alpha)$$

This concludes the case $\langle \mathsf{true} \rangle \mathcal{A}$.

The case $\langle af \wedge b \rangle \mathcal{A}$

The case $af \wedge b$, is equal to the case of the communication operator:

$$(af \wedge b) /_A \nabla_V = af /_A \nabla_V \wedge b$$

The case $\langle af \wedge \alpha \rangle \mathcal{A}$

A more interesting case is $af \wedge \alpha$. As with the communication operator, we construct an action formula that matches the multi-action α *before* the allow operator. This is simply α itself if $\underline{\alpha}$ is in the allow set $V \cup \{\tau\}$, or false otherwise. This is a simple case distinction, as we can determine whether α is allowed immediately. We also create a disjunction of splitting α in a part ζ and η, as with the communication operator. This means that for the case $af \wedge \alpha$, the quotient rule is simply

$$(af \wedge \alpha) / \nabla_V = \bigvee_{\zeta | \eta = \alpha} \begin{cases} af / \nabla_V \wedge \zeta | \rho(\eta) & \text{if } \underline{\zeta} \in V \cup \{\tau\} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The case $\langle af \wedge \neg\alpha \rangle \mathcal{A}$

For the case $af \wedge \neg\alpha$, we apply the same case distinction. In the case where α is in the allow set, the action formula $\neg\alpha$ simply remains $\neg\alpha$. In the case where α is *not* in the allow set, the requirement $\neg\alpha$ is already enforced by the allow operator and can be removed.

Conclusion

This leads to the following definition of quotienting the allow operator:

Definition 5.3.2 (Quotienting the allow operator)**.** Let $\mathcal{A}$ be an assertion and let V be a set of multi-action names. Then, quotienting out the operator ∇_V from assertion $\mathcal{A}$, denoted $\mathcal{A}/\nabla_V$, is defined as follows:

$$(\langle af \rangle \mathcal{A})/_A \nabla_V = (\exists d_{v_1} : D_{v_1}, \ldots, d_{v_n} : D_{v_n} \,.\, \langle af/_A \nabla_V \rangle (\mathcal{A}/_A \nabla_V)),$$

where D_v denotes the data parameter sort of the multi-action v for each $v \in V$, and quotienting the allow operator on action formulae is defined as:

$$\begin{aligned}
\mathsf{true}/_A \nabla_V &= \bigvee_{v \in V} \bigvee_{\alpha \in A} v(d_v)|\rho(\alpha) \\
(af \wedge b)/_A \nabla_V &= af/_A \nabla_V \wedge b \\
(af \wedge \alpha)/_A \nabla_V &= \bigvee_{\zeta|\eta=\alpha} \begin{cases} af/_A \nabla_V \wedge \zeta|\rho(\eta) & \text{if } \underline{\zeta} \in V \cup \{\tau\} \\ \mathsf{false} & \text{otherwise} \end{cases} \\
(af \wedge \neg\alpha)/_A \nabla_V &= \bigvee_{\zeta|\eta=\alpha} \begin{cases} af \wedge \zeta|\rho(\eta) & \text{if } \underline{\zeta} \in V \cup \{\tau\} \\ af/_A \nabla_V & \text{otherwise} \end{cases}
\end{aligned}$$

Theorem 5.3.3 (Quotienting the allow operator is sound)**.** Quotienting the allow operator is sound, i.e. for any assertion $\mathcal{A}$, processes P_1 and P_2, set of multi-action names V, environment ρ, data environment ε and state s, it holds that

$$s \in [\![\mathcal{A}]\!]_{\nabla_V(P_1) \,||\, P_2} \rho\varepsilon \Leftrightarrow s \in [\![\mathcal{A}/_{Act(P_2)} \nabla_V]\!]_{P_1 \,||\, \rho(P_2)} \rho\varepsilon.$$

Proof. By derivation of Definition 5.3.2 (Quotienting the allow operator). □

We explore an example:

Example 5.3.4 (Quotienting the allow operator)**.** Consider the process $\nabla_{\{\mathsf{transfer}\}}(R)$ of Example 2.9.5 (Allow operator on linear process equations) and the assertion $\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}$. The model checking problem is expressed as:

$$s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}]\!]_{\nabla_{\{\mathsf{transfer}\}}(R)} \rho\varepsilon,$$

where ρ is some environment and ε a data environment.

We rewrite this to match the quotienting rule in Theorem 5.3.3 (Quotienting the allow operator is sound):

$$\begin{aligned}
& s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}]\!]_{\nabla_{\{\mathsf{transfer}\}}(R)} \rho\varepsilon \\
= \; & \{\text{deadlock is unit element of parallel composition}\} \\
& s \in [\![\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true}]\!]_{\nabla_{\{\mathsf{transfer}\}}(R) \,||\, P_\delta} \rho\varepsilon
\end{aligned}$$

We calculate $Act(P_\delta)$, which is $\{\tau\}$. Now, we apply Theorem 5.3.3 (Quotienting the allow operator is sound) and obtain:

$$\begin{aligned}
= \; & s \in [\![(\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true})/_{\{\tau\}} \nabla_{\{\mathsf{transfer}\}}]\!]_{R \,||\, \rho(P_\delta)} \rho\varepsilon \\
= \; & \{\text{rename on deadlock process has no effect, deadlock is unit element of parallel composition}\} \\
& s \in [\![(\langle \mathsf{transfer}(\mathsf{message}) \rangle \mathsf{true})/_{\{\tau\}} \nabla_{\{\mathsf{transfer}\}}]\!]_R \rho\varepsilon
\end{aligned}$$

Quotienting the allow operator from the assertion is as follows:

$$
\begin{aligned}
&(\langle \mathsf{transfer(message)} \rangle \mathsf{true}) /_{\{\tau\}} \nabla_{\{\mathsf{transfer}\}} \\
&= (\exists d : D \;.\; \langle (\mathsf{true} \wedge \mathsf{transfer(message)}) /_{\{\tau\}} \nabla_{\{\mathsf{transfer}\}} \rangle \mathsf{true} /_{\{\tau\}} \nabla_{\{\mathsf{transfer}\}}) \\
&= (\exists d : D \;.\; \left\langle \bigvee_{\zeta|\eta = \mathsf{transfer(message)}} \begin{cases} \mathsf{true} /_{\{\tau\}} \nabla_{\{\mathsf{transfer}\}} \wedge \zeta|\rho(\eta) & \text{if } \underline{\zeta} \in \{\mathsf{transfer}, \tau\} \\ \mathsf{false} & \text{otherwise} \end{cases} \right\rangle \mathsf{true})
\end{aligned}
$$

There are two possibilities for ζ and η: either $\zeta = \mathsf{transfer(message)}$ and $\eta = \tau$, or vice-versa. As $\underline{\mathsf{transfer(message)}} = \mathsf{transfer}$ is in the allow set, we obtain:

$$
\begin{aligned}
&= (\exists d : D \;.\; \langle (\mathsf{true} \wedge \mathsf{transfer(message)}) \vee (\mathsf{true} \wedge \mathsf{transfer}'\mathsf{(message)}) \rangle \mathsf{true}) \\
&= (\exists d : D \;.\; \langle (\mathsf{transfer}(d) \wedge \mathsf{transfer(message)}) \vee (\mathsf{transfer}(d) \wedge \mathsf{transfer}'\mathsf{(message)}) \rangle \mathsf{true}
\end{aligned}
$$

Note that $\mathsf{transfer}(d)$ and $\mathsf{transfer}'\mathsf{(message)}$ cannot both be true, and that we can apply a one-point rule to the existential quantifier for $d \approx \mathsf{message}$. This yields:

$$= \langle \mathsf{transfer(message)} \rangle \mathsf{true}$$

So, after quotienting the allow operator, we get the following equivalent model checking problem:

$$s \in [\![\langle \mathsf{transfer(message)} \rangle \mathsf{true}]\!]_R \rho\varepsilon$$

Another interesting case is when transfer would *not* be in the allow set:

$$
\begin{aligned}
&\langle \mathsf{transfer(message)} \mathsf{true} \rangle /_{\{\tau\}} \nabla_{\{move\}} \\
&= \left\langle \bigvee_{\zeta|\eta = \mathsf{transfer(message)}} \begin{cases} \mathsf{true} /_{\{\tau\}} \nabla_{\{move\}} \wedge \zeta|\rho(\eta) & \text{if } \underline{\zeta} \in \{move, \tau\} \\ \mathsf{false} & \text{otherwise} \end{cases} \right\rangle \mathsf{true} /_{\{\tau\}} \nabla_{\{move\}} \\
&= \langle \mathsf{false} \rangle \mathsf{true} \\
&= \mathsf{false}
\end{aligned}
$$

The case where $\mathsf{transfer(message)}$ was enabled in R, is no longer available, as this would be blocked by the allow operator.

5.4 Quotienting the rename operator

Quotienting of the rename operator is the same as quotienting of the communication operator. However, is with the definition of the rename operator, the expressions can be simplified, noting that there is no need for parameter matching when there is only on parameter. This yields the following definition of quotienting the rename operator:

Definition 5.4.1 (Quotienting the rename operator)**.** Let $\mathcal{A}$ be an assertion and let C be a set of single substitutions. Then, quotienting out the operator ρ_C from assertion $\mathcal{A}$, denoted $\mathcal{A}/\rho_C$, is defined as follows:

If C is not a singleton:

$$\mathcal{A}/\rho_{C_1 \cup C_2} = \mathcal{A}/\rho_{C_1}/\rho_{C_2}$$

If C is a singleton $a \to \beta'$:

$$(\langle af \rangle \mathcal{A})/\rho_C = \langle af/\rho_C \rangle (\mathcal{A}/\rho_{\{b \to \beta'\}}),$$

where renameing on action formulae is defined as

$$\begin{aligned}
\mathsf{true}/\rho_C &= \mathsf{true}\\
(af \wedge b)/\rho_C &= af/\rho_C \wedge b\\
(af \wedge \alpha)/\rho_C &= af/\rho_C \wedge \bigvee_{\zeta|\eta=\alpha} \mathsf{AF}(\zeta,\rho(\eta))\\
(af \wedge \neg\alpha)/\rho_C &= af/\rho_C \wedge \neg \bigvee_{\zeta|\eta=\alpha} \mathsf{AF}(\zeta,\rho(\eta)),
\end{aligned}$$

and the action formula AF is defined as:

$$\mathsf{AF}(\alpha,\gamma) = \begin{cases} \gamma|\alpha & \text{if } a \not\sqsubseteq \alpha\\ \mathsf{false} & \text{otherwise}\end{cases} \vee \bigvee_{\substack{(a_1(e_1)|\ldots|a_{|\beta'|}(e_{|\beta'|}),\alpha')\in \mathsf{choose}_{|\beta'|}(\alpha)\\ a_1|\ldots|a_{|\beta'|}=\beta'}} \big(e_1 \approx \ldots \approx e_{|\beta'|} \wedge \mathsf{AF}(\alpha',\gamma|a)\big)$$

The soundness of quotienting of the rename operator, follows from this derivation.

Theorem 5.4.2 (Quotienting the rename operator is sound)**.** Quotienting the rename operator is sound, i.e. for any assertion $\mathcal{A}$, processes P_1 and P_2, set of single substitutions C, environment ρ, data environment ε and state s, it holds that

$$s \in [\![\mathcal{A}]\!]_{\rho_C(P_1)\,||\,P_2}\rho\varepsilon \Leftrightarrow s \in [\![\mathcal{A}/\rho_C]\!]_{P_1\,||\,\rho(P_2)}\rho\varepsilon.$$

Proof. By derivation of Definition 5.4.1 (Quotienting the rename operator). □

Example 5.4.3 (Quotienting the rename operator)**.** In Example 2.10.4 (Rename operator), we rename the send action to an out action in the process R of Definition 2.8.4 (Communication operator on linear process equations). This is the process $\rho_{\{\mathsf{send}\to out\}}(R)$. Suppose we were interested in whether this process can perform an out(1)-action, expressed as the assertion $\langle \mathsf{out}(1)\rangle\mathsf{true}$. The model checking problem is as follows:

$$s \in [\![\langle \mathsf{out}(1)\rangle\mathsf{true}]\!]_{\rho_{\{\mathsf{send}\to\mathsf{out}\}}(R)}\rho\varepsilon$$

By quotienting the rename operator, we can rewrite this to the following model checking problem:

$$s \in [\![(\langle \mathsf{out}(1)\rangle\mathsf{true})/\rho_{\{\mathsf{send}\to\mathsf{out}\}}]\!]_{R\,||\,\rho(P_\delta)}\rho\varepsilon$$

Now, we calculate the assertion $(\langle \mathsf{out}(1)\rangle\mathsf{true})/\rho_{\{\mathsf{send}\to\mathsf{out}\}}$.

$$\begin{aligned}
&(\langle \mathsf{out}(1)\rangle\mathsf{true})/\rho_{\{\mathsf{send}\to\mathsf{out}\}}\\
&= \langle \mathsf{true}/\rho_{\{\mathsf{send}\to\mathsf{out}\}} \wedge \mathsf{out}(1)/\rho_{\{\mathsf{send}\to\mathsf{out}\}}\rangle\mathsf{true}/\rho_{\{\mathsf{send}\to\mathsf{out}\}}\\
&= \langle \mathsf{true} \wedge \bigvee_{\zeta|\eta=\mathsf{out}(1)} \mathsf{AF}(\zeta,\rho(\eta))\rangle\mathsf{true}
\end{aligned}$$

There are two possibilities for ζ and η: one equals $\mathsf{out}(1)$ and the other τ, and vice-versa.

$$= \langle \mathsf{AF}(\mathsf{out}(1),\rho(\tau)) \vee \mathsf{AF}(\tau,\rho(\mathsf{out}(1)))\rangle\mathsf{true}$$

We expand the definition of AF. For the first one, as $\mathsf{out}(1)$ can be the result of a substitution, we do a recursive step. In the second, there is no recursive step.

$$\begin{aligned}
&= \langle (\mathsf{out}(1)|\rho(\tau) \vee \mathsf{AF}(\tau,\mathsf{move}(1)|\rho(\tau))) \vee \mathsf{out}'(1)\rangle\mathsf{true}\\
&= \langle \mathsf{out}(1) \vee \mathsf{move}(1) \vee \mathsf{out}'(1)\rangle\mathsf{true}
\end{aligned}$$

So, we obtain the following model checking problem:

$$s \in [\![\langle \mathsf{move}(1) \vee \mathsf{out}'(1)\rangle \mathsf{true}]\!]_{R \,||\, \rho(P_\delta)} \rho\varepsilon$$

These three disjuncts neatly correspond to

- the $\mathsf{out}(1)$ action was already enabled and the rename had no effect on it,
- a $\mathsf{move}(1)$ action was enabled and it was renamed to $\mathsf{out}(1)$, and
- the $\mathsf{out}(1)$ action was enabled in the remainder process $\rho(P_\delta)$ and R stays idle.

5.5 Quotienting the abstraction operator

The interesting case is the one for the modal operator. For the other assertions, the derivation is again identical to the derivation of Definition 5.2.1 (Quotienting the communication operator). The specification for this case is as follows:

$$s \in [\![(\langle af\rangle \mathcal{A})/\tau_H]\!]_{P_1 \,||\, \rho(P_2)} \rho\varepsilon \Leftrightarrow s \in [\![\langle af\rangle \mathcal{A}]\!]_{\tau_H(P_1) \,||\, P_2} \rho\varepsilon$$

We construct an action formula that matches a given multi-action α *before* the abstraction operator was applied. However, the set $\{\beta \mid \tau_H(\beta) = \alpha\}$ turns out to infinitely large, which can be easily seen by an example: suppose $\alpha = a$ and H is the singleton b. Then, a should be in this set, as does $a|b$, $a|b|b$, etcetera. There is no finite action formula that can describe this set. We solve this problem by overapproximating what possible actions are actually present in the process the abstraction operator was applied to. We use the overapproximation Act we defined for the allow operator, Definition 5.3.1 (Overapproximation of multi-actions of a process description).

Now we can simply enumerate the possibilities, i.e. we create a disjunction over all β in a set A, where $\tau_\beta = \alpha$. Finally, we create a disjunction of splitting α in a part ζ and η, as with the other process operators. This leads to the following definition of quotienting the abstraction operator:

Definition 5.5.1 (Quotienting the abstraction operator)**.** Let $\mathcal{A}$ be an assertion, H a set of multi-action names and A a set of multi-actions. Then, quotienting out the operator τ_H from assertion $\mathcal{A}$, where A is the overapproximation of actions that are enabled, denoted $\mathcal{A}/_A\tau_H$, is defined as follows:

$$\langle af\rangle \mathcal{A}/_A\tau_H = \langle af/_A\tau_H\rangle(\mathcal{A}/_A\tau_H),$$

where quotienting the abstraction operator from a action formulae is defined as follows:

$$\begin{aligned}
\mathsf{true}/_A\tau_H &= \mathsf{true} \\
(af \wedge b)/_A\tau_H &= af/_A\tau_H \wedge b \\
(af \wedge \alpha)/_A\tau_H &= af/_A\tau_H \wedge \left(\bigvee_{\zeta|\eta=\alpha} \bigvee_{\substack{\beta\in A \\ \tau_H(\beta)=\zeta}} \beta|\rho(\eta) \right) \\
(af \wedge \neg\alpha)/_A\tau_H &= af/_A\tau_H \wedge \neg \left(\bigvee_{\zeta|\eta=\alpha} \bigvee_{\substack{\beta\in A \\ \tau_H(\beta)=\alpha}} \neg\beta|\rho(\eta) \right)
\end{aligned}$$

The use of $Act(Q_2)$ can be seen in the following quotienting rule:

Theorem 5.5.2 (Quotienting the abstraction operator is sound)**.** Quotienting the abstraction operator is sound, i.e. for any assertion $\mathcal{A}$, processes P_1 and P_2, set of multi-action names H, environment ρ, data environment ε and state s, it holds that

$$s \in [\![\mathcal{A}]\!]_{\tau_H(P_1) \,||\, P_2 \rho\varepsilon} \Leftrightarrow s \in [\![\mathcal{A}/_{Act(P_1)\tau_H}]\!]_{P_1 \,||\, \rho(P_2)\rho\varepsilon}.$$

Proof. By derivation of Definition 5.5.1 (Quotienting the abstraction operator). □

Example 5.5.3 (Quotienting the abstraction operator)**.** In Example 2.11.6 (Abstraction operator on linear process equations), we applied abstraction to the process R of Example 2.8.6 (Communication operator on linear process equations). We consider the process $\tau_{\{\mathsf{send},\mathsf{receive}\}}(R)$, where R is defined as:

$$\begin{aligned} R = &\sum_{d:D} \mathsf{true} \to \mathsf{send}(d) \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{receive}(d) \;.\; R \\ &+ \sum_{d:D} \sum_{d':D} d \not\approx d' \to \mathsf{send}(d)|\mathsf{receive}(d') \;.\; R \\ &+ \sum_{d:D} \mathsf{true} \to \mathsf{transfer}(d) \;.\; R \end{aligned}$$

Suppose that we are interested in whether $\tau_{\{\mathsf{send},\mathsf{receive}\}}(R)$ satisfies the property that a $\mathsf{transfer}(1)$-action is possible: $\langle \mathsf{transfer}(1) \rangle \mathsf{true}$. The model checking problem is:

$$s \in [\![\langle \mathsf{transfer}(1) \rangle \mathsf{true}]\!]_{\tau_{\{\mathsf{send},\mathsf{receive}\}}(R) \,||\, P \rho\varepsilon}$$

By quotienting out the abstraction operator, we can express this as a property on only R and P:

$$s \in [\![(\langle \mathsf{transfer}(1) \rangle \mathsf{true})/_A \tau_{\{\mathsf{send},\mathsf{receive}\}}]\!]_{R \,||\, \rho(P)\rho\varepsilon}$$

Unfortunately, we do need to inspect R to see what multi-actions are possible, or at least obtain a finite overapproximation of those possible multi-actions. This is the set of multi-actions A. We calculate it to be $A = \{\mathsf{send}(d), \mathsf{receive}(d), \mathsf{send}(d)|\mathsf{receive}(d'), \mathsf{transfer}(d)\}$.

We calculate $(\langle \mathsf{transfer} \rangle \mathsf{true})/_A \tau_{\{\mathsf{send},\mathsf{receive}\}}$:

$$\begin{aligned} &(\langle \mathsf{transfer}(1) \rangle \mathsf{true})/_A \tau_{\{\mathsf{send},\mathsf{receive}\}} \\ &= \langle \mathsf{transfer}(1)/_A \tau_{\{\mathsf{send},\mathsf{receive}\}} \rangle (\mathsf{true}/_A \tau_{\{\mathsf{send},\mathsf{receive}\}}) \\ &= \langle \mathsf{true}/_A \tau_{\{\mathsf{send},\mathsf{receive}\}} \wedge \bigvee_{\eta|\zeta = \mathsf{transfer}(1)} \bigvee_{\substack{\beta \in A \\ \tau_{\{\mathsf{send},\mathsf{receive}\}}(\beta) = \zeta}} \beta|\rho(\eta) \rangle \mathsf{true} \end{aligned}$$

We expand the possibilities for η and ζ: one equals $\mathsf{transfer}(1)$ and the other is τ, and vice-versa.

$$= \left\langle \begin{array}{ll} \displaystyle\bigvee_{\substack{\beta \in A \\ \tau_{\{\mathsf{send},\mathsf{receive}\}}(\beta) = \tau}} & \beta|\rho(\mathsf{transfer}(1)) \\ \vee & \displaystyle\bigvee_{\substack{\beta \in A \\ \tau_{\{\mathsf{send},\mathsf{receive}\}}(\beta) = \mathsf{transfer}(1)}} \beta|\rho(\tau) \end{array} \right\rangle \mathsf{true}$$

There are three possibilities for β in the first disjunction, and only one in the second. To have β equal $\mathsf{transfer}(1)$, we require that the parameter d in $\mathsf{transfer}(d)$, matches the parameter 1. We expand the possibilities for β and apply the renaming to fresh action names. We obtain the following model checking problem:

$$\begin{aligned} s \in [\![& \langle \mathsf{send}(d) | \mathsf{transfer}'(1) \\ & \vee \mathsf{receive}(d) | \mathsf{transfer}'(1) \\ & \vee \mathsf{send}'(d) | \mathsf{receive}'(d') | \mathsf{transfer}'(1) \\ & \vee \mathsf{transfer}(1) \rangle \mathsf{true}]\!]_{R \,||\, \rho(P)\rho\varepsilon} \end{aligned}$$

These four possibilities correspond to three possible ways in which R does a transition that is hidden by the abstraction and P does a $\mathsf{transfer}(1)$ transition, and the one possibility where R does the $\mathsf{transfer}(1)$ transition.

5.6 Buffer example (5)

We revisit the example with an n-place buffer constructed of n one-place buffers. Using quotienting, we can remove one of the parallel one-place buffers from the model side of the model checking equation, to the property side. In other words, we express the property that we would like to check against the $n+1$-place buffer, as a property we still have to check on the n-place buffer, by quotienting out one of the parallel one-place buffers. Of course, as these one-place buffers are linked together with process operators, this means we need to quotient out these process operators as well.

The $n+1$-place buffer is constructed from a one-place buffer B_1 and an n-place buffer B_n as follows (cf. Section 4.3):

$$\tau_{\{\mathrm{move}\}}(\nabla_{\{\mathrm{in},\mathrm{out},\mathrm{move}\}}(\Gamma_{\{\mathrm{out}'|\mathrm{in}'\to\mathrm{move}\}}(\rho_{\{\mathrm{out}\to\mathrm{out}'\}}(B_1) \,||\, \rho_{\{\mathrm{in}\to\mathrm{in}'\}}(B_n))))$$

The property we would like to check on this process, is the property that every data element that goes in the buffer, eventually comes out (cf. Section 3.2):

$$\mathcal{E} \downarrow X \quad = \quad \begin{pmatrix} \nu X = [\mathsf{true}]X \wedge (\forall m : D \;.\; [\mathrm{in}(m)]Y(m)) \\ \mu Y(m : D) = [\neg \mathrm{out}(m)]Y(m) \wedge \langle \mathsf{true} \rangle \mathsf{true} \end{pmatrix} \downarrow X$$

Intuitively, this should be true.

Using quotienting, we can rewrite the question of whether the $n+1$-place buffer satisfies this property, to whether the one-place buffer and the n-place buffer satisfy it. We quotient the process operators from this top assertion, one by one. The original model checking problem is as follows:

$$\tau_{\{\mathrm{move}\}}(\nabla_{\{\mathrm{in},\mathrm{out},\mathrm{move}\}}(\Gamma_{\{\mathrm{out}'|\mathrm{in}'\to\mathrm{move}\}}(\rho_{\{\mathrm{out}\to\mathrm{out}'\}}(B_1) \,||\, \rho_{\{\mathrm{in}\to\mathrm{in}'\}}(B_n)))) \models \mathcal{E} \downarrow X$$

To fit this to the definition of Definition 5.5.1 (Quotienting the abstraction operator), we use the fact that the deadlock process is the unit of parallel composition:

$$\tau_{\{\mathrm{move}\}}(\nabla_{\{\mathrm{in},\mathrm{out},\mathrm{move}\}}(\Gamma_{\{\mathrm{out}'|\mathrm{in}'\to\mathrm{move}\}}(\rho_{\{\mathrm{out}\to\mathrm{out}'\}}(B_1) \,||\, \rho_{\{\mathrm{in}\to\mathrm{in}'\}}(B_n)))) \,||\, P_\delta \models \mathcal{E} \downarrow X$$

Quotienting abstraction

From this equation, we can quotient the abstraction operator. The first step is to build the overapproximation of the set of actions that are enabled:

$$A = Act(\nabla_{\{\mathrm{in},\mathrm{out},\mathrm{move}\}}(\Gamma_{\{\mathrm{out}'|\mathrm{in}'\to\mathrm{move}\}}(\rho_{\{\mathrm{out}\to\mathrm{out}'\}}(B_1) \,||\, \rho_{\{\mathrm{in}\to\mathrm{in}'\}}(B_n))))$$

Fortunately, because of the allow operator, we can easily see that A only contains in, out and move actions, as well as the τ action which the allow operator never blocks. We do not explicitly build Act in this example, as the intermediate expressions are quite large. Also, we cannot inspect B_n in this example, nor do we need to.

$$\nabla_{\{\text{in,out,move}\}}(\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}(\rho_{\{\text{out}\to\text{out}'\}}(B_1) \,||\, \rho_{\{\text{in}\to\text{in}'\}}(B_n))) \,||\, \rho(P_\delta) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}$$

We calculate $(\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}$. First, we distribute the quotienting operation over equations and boolean connectives:

$$\begin{aligned} &\mathcal{E}/_A\tau_{\{\text{move}\}} \\ &= \begin{pmatrix} \nu X = [\mathsf{true}/_A\tau_{\{\text{move}\}}]X/_A\tau_{\{\text{move}\}} \wedge (\forall m : D \,.\, [\text{in}(m)/_A\tau_{\{\text{move}\}}]Y(m)/_A\tau_{\{\text{move}\}}) \\ \mu Y(m : D) = [(\neg\text{out}(m))/_A\tau_{\{\text{move}\}}]Y(m)/_A\tau_{\{\text{move}\}} \wedge \langle \mathsf{true}/_A\tau_{\{\text{move}\}}\rangle \mathsf{true}/_A\tau_{\{\text{move}\}} \end{pmatrix} \end{aligned}$$

We can immediately reduce most of it, as $\mathsf{true}/_A\tau_H = \mathsf{true}$ for both assertions and action formulae, and recursion variables such as X are also not affected:

$$= \begin{pmatrix} \nu X = [\mathsf{true}]X \wedge (\forall m : D \,.\, [\text{in}(m)/_A\tau_{\{\text{move}\}}]Y(m)) \\ \mu Y(m : D) = [(\neg\text{out}(m))/_A\tau_{\{\text{move}\}}]Y(m) \wedge \langle \mathsf{true}\rangle \mathsf{true} \end{pmatrix}$$

The two quotienting expressions on action formulae are the interesting cases. We immediately expand the large disjunctions in their definition: for each, ζ and η have two possibilites: one is the entire action and the other is τ, and vice-versa. Since A only contains in, out, move and τ actions, the requirement that $\tau_{\{\text{move}\}}(\beta)$ equals $\text{in}(m)$ or $\text{out}(m)$ is simply that β equals it directly. For the case $\tau_{\{\text{move}\}}(\beta) = \tau$, the multi-action β can equal τ directly or equal move which is abstracted to τ.

$$= \begin{pmatrix} \nu X = [\mathsf{true}]X \wedge (\forall m : D \,.\, [\text{in}(m) \vee \text{in}'(m) \vee \text{move}|\text{in}'(m)]Y(m)) \\ \mu Y(m : D) = [\neg(\text{out}(m) \vee \text{out}'(m) \vee \text{move}|\text{out}'(m))]Y(m) \wedge \langle \mathsf{true}\rangle \mathsf{true} \end{pmatrix}$$

Note that the deadlock process keeps collecting rename operators which have no effect, as the deadlock process has no transitions to rename. This is common. From now on, we keep the deadlock process implicit.

Because we would like to keep our intermediate results small, we immediately try to reduce this property. We already noted that the deadlock process cannot do any actions and that it accumulates rename operators which have no effect. From this, we can also see that actions which are renamed in the property, e.g. $\text{in}'(m)$ or the out' action in $\text{move}|\text{out}'(m)$, are never performed by this deadlocked process. Therefore, we can remove these actions. Perhaps not very surprisingly, this takes us back to the starting point $\mathcal{E}$. We conclude that in for particular process and this particular property, the abstraction operator had no effect on the property's validity.

We have calculated that $(\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}} = \mathcal{E} \downarrow X$.

Quotienting allow

Next, we quotient out the allow operator, to obtain:

$$\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}(\rho_{\{\text{out}\to\text{out}'\}}(B_1) \,||\, \rho_{\{\text{in}\to\text{in}'\}}(B_n)) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}/_{Act(P_\delta)}\nabla_{\{\text{in,out,move}\}}$$

The set $Act(P_\delta)$ is simply $\{\tau\}$. We calculate $(\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}$:

$$\begin{aligned} &\mathcal{E}/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}} \\ &= \mathcal{E}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}} \end{aligned}$$

The in, out and move actions take a parameter of type D, which we quantify over:

$$= \left(\begin{array}{l} \nu X = \\ \quad (\forall d : D \,.\, [\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau] X \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}) \\ \quad \wedge (\forall m : D \,.\, (\forall d : D \,. \\ \qquad [(\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau) \wedge \mathrm{in}(m) \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}] \\ \qquad\quad Y(m) \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}) \\ \quad) \\ \mu Y(m : D) = \\ \quad (\forall d : D \,. \\ \qquad [(\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau) \wedge (\neg \mathrm{out}(m)) \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}] \\ \qquad\quad Y(m) \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}} \\ \quad) \\ \quad \wedge (\exists d : D \,.\, \langle \mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau \rangle \mathsf{true} \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}) \end{array}\right)$$

The recursion variables X and $Y(m)$ are left unchanged by quotienting the allow operator. Because $\underline{\mathrm{in}(m)} = in$ is in the allow set, it is also unchanged. Similarly, $\neg\mathrm{out}(m)$ is unchanged. We also note that the action formula $(\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau) \wedge \mathrm{in}(m)$ is equal to $\mathrm{in}(m)$ because there are no further restrictions on d.

$$= \left(\begin{array}{l} \nu X = \\ \quad (\forall d : D \,.\, [\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau] X) \\ \quad \wedge (\forall m : D \,.\, [\mathrm{in}(m)] Y(m)) \\ \mu Y(m : D) = \\ \quad (\forall d : D \,.\, [(\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau) \wedge \neg \mathrm{out}(m)] Y(m)) \\ \quad \wedge (\exists d : D \,.\, \langle \mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau \rangle \mathsf{true} \end{array}\right)$$

Quotienting communication

So far, we have calculated $\mathcal{E} \mathbin{/_A} \tau_{\{\mathrm{move}\}} \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}$ and found that the original model checking problem is equivalent to the following problem:

$$\Gamma_{\{\mathrm{out'|in'} \to \mathrm{move}\}}(\rho_{\{\mathrm{out} \to \mathrm{out'}\}}(B_1) \,||\, \rho_{\{\mathrm{in} \to \mathrm{in'}\}}(B_n)) \models (\mathcal{E} \downarrow X) \mathbin{/_A} \tau_{\{\mathrm{move}\}} \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}},$$

where

$$(\mathcal{E} \downarrow X) \mathbin{/_A} \tau_{\{\mathrm{move}\}} \mathbin{/_{\{\tau\}}} \nabla_{\{\mathrm{in,out,move}\}}$$
$$= \left(\begin{array}{l} \nu X = \\ \quad (\forall d : D \,.\, [\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau] X) \\ \quad \wedge (\forall m : D \,.\, [\mathrm{in}(m)] Y(m)) \\ \mu Y(m : D) = \\ \quad (\forall d : D \,.\, [(\mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau) \wedge \neg \mathrm{out}(m)] Y(m)) \\ \quad \wedge (\exists d : D \,.\, \langle \mathrm{in}(d) \vee \mathrm{out}(d) \vee \mathrm{move}(d) \vee \tau \rangle \mathsf{true} \end{array}\right) \downarrow X.$$

The next step is to quotient the communication operator. This is more straightforward. Note that move can be the result of a substitution, so we account for two possibilities: either move indeed is the result of a substitution, or there was already a move transition.

We calculate $\mathcal{E}/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}/\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}$.

$$
\begin{aligned}
&\mathcal{E}/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}/\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}} \\
&= \left(\begin{array}{l}
\nu X = \\
\quad (\forall d : D \;.\; [\text{in}(d) \vee \text{out}(d) \vee \text{move}(d) \vee \tau]X) \\
\quad \wedge (\forall m : D \;.\; [\text{in}(m)]Y(m)) \\
\mu Y(m : D) = \\
\quad (\forall d : D \;.\; [(\text{in}(d) \vee \text{out}(d) \vee \text{move}(d) \vee \tau) \wedge \neg\text{out}(m)]Y(m)) \\
\quad \wedge (\exists d : D \;.\; \langle \text{in}(d) \vee \text{out}(d) \vee \text{move}(d) \vee \tau\rangle \mathsf{true})
\end{array}\right) /\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}
\end{aligned}
$$

We do the usual disjunction over ζ and η, again noting that for these single actions, one equals the entire action and the other is τ, and vice-versa. For each of these possibilities, there is an action formula AF:

$$
\begin{aligned}
&= \left(\begin{array}{l}
\nu X = \\
\quad (\forall d : D \;.\; \left[\begin{array}{l}
(AF_{in'|out',move}(\text{in}(d), \tau') \vee AF_{in'|out',move}(\tau, in''(d)) \\
\vee AF_{in'|out',move}(\text{out}(d), \tau') \vee AF_{in'|out',move}(\tau, out'(d)) \\
\vee AF_{in'|out',move}(\text{move}(d), \tau') \vee AF_{in'|out',move}(\tau, move'(d)) \\
\vee AF_{in'|out',move}(\tau, \tau') \vee AF_{in'|out',move}(\tau, \tau')
\end{array}\right] X) \\
\quad \wedge (\forall m : D \;.\; [AF_{in'|out',move}(\text{in}(m), \tau') \vee AF_{in'|out',move}(\tau, \text{in}'(m))]Y(m)) \\
\mu Y(m : D) = \\
\quad (\forall d : D \;.\; \left[\begin{array}{l}
(AF_{in'|out',move}(\text{in}(d), \tau') \vee AF_{in'|out',move}(\tau, in''(d)) \\
\vee AF_{in'|out',move}(\text{out}(d), \tau') \vee AF_{in'|out',move}(\tau, out'(d)) \\
\vee AF_{in'|out',move}(\text{move}(d), \tau') \vee AF_{in'|out',move}(\tau, move'(d)) \\
\vee AF_{in'|out',move}(\tau, \tau') \vee AF_{in'|out',move}(\tau, \tau') \\
) \wedge \neg(AF_{in'|out',move}(\text{out}(m), \tau') \vee AF_{in'|out',move}(\tau, out'(m)))
\end{array}\right] Y(m)) \\
\quad \wedge (\exists d : D \;.\; \left[\begin{array}{l}
(AF_{in'|out',move}(\text{in}(d), \tau') \vee AF_{in'|out',move}(\tau, in''(d)) \\
\vee AF_{in'|out',move}(\text{out}(d), \tau') \vee AF_{in'|out',move}(\tau, out'(d)) \\
\vee AF_{in'|out',move}(\text{move}(d), \tau') \vee AF_{in'|out',move}(\tau, move'(d)) \\
\vee AF_{in'|out',move}(\tau, \tau') \vee AF_{in'|out',move}(\tau, \tau')
\end{array}\right] \mathsf{true})
\end{array}\right)
\end{aligned}
$$

Expanding the definition of AF is simple:

$$= \left(\begin{array}{l} \nu X = \\ \quad (\forall d : D \;.\; \left[\begin{array}{l} \mathrm{in}(d) \vee in''(d) \\ \vee\, \mathrm{out}(d) \vee out'(d) \\ \vee\, (\mathrm{move}(d) \vee AF_{in'|out',move}(\tau, in'(d)|out'(d)|\tau')) \vee move'(d) \\ \vee\, \tau \end{array} \right] X) \\ \quad \wedge\, (\forall m : D \;.\; [\mathrm{in}(m) \vee in'(m)]Y(m)) \\ \mu Y(m : D) = \\ \quad (\forall d : D \;.\; \left[\begin{array}{l} (\mathrm{in}(d) \vee in''(d) \\ \vee\, \mathrm{out}(d) \vee out'(d) \\ \vee\, (\mathrm{move}(d) \vee AF_{in'|out',move}(\tau, in'(d)|out'(d)|\tau')) \vee move'(d) \\ \vee\, \tau \\ ) \wedge \neg(\mathrm{out}(m) \vee out'(m)) \end{array} \right] Y(m)) \\ \quad \wedge\, (\exists d : D \;.\; \left[\begin{array}{l} \mathrm{in}(d) \vee in''(d) \\ \vee\, \mathrm{out}(d) \vee out'(d) \\ \vee\, (\mathrm{move}(d) \vee AF_{in'|out',move}(\tau, in'(d)|out'(d)|\tau')) \vee move'(d) \\ \vee\, \tau \end{array} \right] \mathsf{true}) \end{array} \right)$$

Finally, we expand the recursive step of AF. Without knowledge of the second parallel process, in this case P_δ, the calculation would end there. We would keep open all possibilities of parts of the original actions being performed by this parallel process. However, we can go a bit further and use this information anyway. Since we know the parallel process is the deadlock process and thus does not do any actions, we can remove all those possibilities where the parallel process takes a transition. What remains is very similar to before quotienting the communication operator, with an additional possibility for move(d) to occur: it was the result of a communication of $in'(d)|out'(d)$.

$$= \left(\begin{array}{l} \nu X = \\ \quad (\forall d : D \;.\; [\mathrm{in}(d) \vee \mathrm{out}(d) \vee (\mathrm{move}(d) \vee in'(d)|out'(d)) \vee \tau]X) \\ \quad \wedge\, (\forall m : D \;.\; [\mathrm{in}(m)]Y(m)) \\ \mu Y(m : D) = \\ \quad (\forall d : D \;.\; [(\mathrm{in}(d) \vee \mathrm{out}(d) \vee (\mathrm{move}(d) \vee in'(d)|out'(d)) \vee \tau) \wedge \neg\mathrm{out}(m)]Y(m)) \\ \quad \wedge\, (\exists d : D \;.\; [\mathrm{in}(d) \vee \mathrm{out}(d) \vee (\mathrm{move}(d) \vee in'(d)|out'(d)) \vee \tau]\mathsf{true}) \end{array} \right)$$

This is the property $\mathcal{E}/_A\tau_{\{\mathrm{move}\}}/_{\{\tau\}}\nabla_{\{\mathrm{in,out,move}\}}/\Gamma_{\{\mathrm{out'|in'\to move}\}}$. Thus far, we have the following model checking problem, which is equivalent to the original model checking problem:

$$\rho_{\{\mathrm{out}\to\mathrm{out'}\}}(B_1) \,||\, \rho_{\{\mathrm{in}\to\mathrm{in'}\}}(B_n) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\mathrm{move}\}}/_{\{\tau\}}\nabla_{\{\mathrm{in,out,move}\}}/\Gamma_{\{\mathrm{out'|in'\to move}\}}$$

Continuing quotienting

From here, we can continue quotienting down to B_n. These steps are all similar to ones we have seen previously. Quotienting the rename operator is very similar to quotienting the communication operator. We do not go into details for this example, only show the quotienting steps without calculating the expression on the property side of the model checking equation.

When we quotient out the rename of B_1, we put another a rename over the B_n part. We remove this later.

$$B_1 \,||\, \rho(\rho_{\{\mathrm{in}\to\mathrm{in'}\}}(B_n)) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\mathrm{move}\}}/_{\{\tau\}}\nabla_{\{\mathrm{in,out,move}\}}/\Gamma_{\{\mathrm{out'|in'\to move}\}}/\rho_{\{\mathrm{out}\to\mathrm{out'}\}}$$

We quotient the linear process equation B_1. We have already seen an example of quotienting a linear process equation in Section 4.3.

$$\rho(\rho_{\{\text{in}\to\text{in}'\}}(B_n)) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}/\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}/\rho_{\{\text{out}\to\text{out}'\}}/B_1$$

The rename to B_n that was the byproduct of quotienting the rename operator around B_1, can now be quotiented out. Again, we use an implicit deadlock parallel process that accumulates rename operators that have no effect:

$$\rho(\rho_{\{\text{in}\to\text{in}'\}}(B_n) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}/\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}/\rho_{\{\text{out}\to\text{out}'\}}/B_1$$

Finally, we quotient the rename operator around B_n.

$$\rho_{\{\text{in}\to\text{in}'\}}(B_n) \models (\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}/\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}/\rho_{\{\text{out}\to\text{out}'\}}/B_1/\rho$$

We can simply forget about the implicit deadlock process with all the renames, as the renames do not do anything and the deadlock process is the unit element of parallel composition. The model checking problem we eventually obtain, which is equivalent to the original model checking problem, is:

$$B_n \models (\mathcal{E} \downarrow X)/_A\tau_{\{\text{move}\}}/_{\{\tau\}}\nabla_{\{\text{in,out,move}\}}/\Gamma_{\{\text{out}'|\text{in}'\to\text{move}\}}/\rho_{\{\text{out}\to\text{out}'\}}/B_1/\rho/\rho_{\{\text{in}\to\text{in}'\}}$$

6 Property minimization

In this section, we explore some techniques to reduce the complexity of the parameterized modal equation systems obtained through quotienting. The quotienting step only *moved* part of the problem's complexity from the model side of the satisfaction equation to the property side. In this section, we reduce the property's complexity and actually make the satisfaction problem easier to solve.

Because parameterized modal equation systems are equally expressive as the modal μ-calculus, we know that it is EXPTIME-hard to solve (Schneider, [10]). Therefore, we resort to heuristic reductions: there are no guarantees that a given parameterized modal equation system can be reduced in complexity. Some of these techniques can be rather expensive operations; sometimes it is more efficient to not reduce the property than to calculate if it can be reduced. Nonetheless, reducing the property's complexity means reducing the overall model checking problem's complexity, which is the aim of the compositional model checking approach.

Admittedly, the reductions we discuss in this section are a rather arbitrary selection of the many heuristic reduction schemes that can be devised to reduce the complexity of parameterized modal equation systems. Nonetheless, this set of reductions should form a good starting point. It incorporates the reductions from Andersen, which were powerful enough in his experiments to dramatically decrease the complexity of the model checking problem. Further research can focus on designing better heuristics to reduce the complexity of parameterized modal equation systems.

The reduction techniques we discuss in this section are:

1. *Simple evaluation*
   Simplification of assertions, e.g. $\mathsf{false} \wedge X(\mathsf{true}) = X(\mathsf{true})$.

2. *Reachability analysis*
   Removes equations of variables on which the top variable does not depend, e.g. $(\mu X = X)\ (\nu Y = X) \downarrow X$ reduces to $(\mu X = X) \downarrow X$, as X does not depend on Y.

3. *Constant propagation*
   Replaces occurrences of a variable by its defining assertion, when that defining assertion is a constant, e.g. $(\mu X = Y)\ (\nu Y = \mathsf{true})$ reduces to $(\mu X = \mathsf{true})\ (\nu Y = \mathsf{true})$.

4. *Unguardedness removal*
   Replaces occurrences of a variable with its defining assertion in assertions where it appears unguarded and which appear before its defining equation, e.g. $(\mu X = Y)\ (\nu Y = X)$ reduces to $(\mu X = X)\ (\nu Y = X)$.

5. *Trivial equation elimination*
   Solves trivial equations, such as $(\mu X = \langle af \rangle X)$ reduces to $(\mu X = \mathsf{false})$.

6. *Action formulae simplification*
   Simplification of action formulae, e.g. $in(message) \wedge \mathsf{true} = in(message)$.

7. *Parameter elimination*
   Removes parameters that do not occur in the defining equation, e.g. $(\mu X(b : \mathsf{Bool}, n : \mathsf{Nat}) = b)$ reduces to $(\mu X(b : \mathsf{Bool}) = b)$.

The naming and definitions of reductions 1 through 5 are taken from Andersen [1]. The equivalence reduction and implication reduction from Andersen are not discussed in detail, as these pertain to reducing the number of equations. Since, contrary to quotienting on labeled transition systems, quotienting on process descriptions does not introduce any new equations, we expect equivalence reduction and implication reduction to have little or no impact. We do discuss parameter elimination, as laid out on parameterized boolean equation systems by Orzan *et al.* [9]. Also, we discuss reducing the complexity of action formulae.

We do not go into details about the soundness of these reductions. Most of these proofs would be rather similar to the proof in Section 4.4. We do provide an argument why we believe these reductions are valid.

6.1 Simple evaluation

Simple evaluation (Andersen [1]) is the simplest of the reductions. It is essentially boolean simplification on the right-hand sides of the equations in a parameterized modal equation system. We rewrite some assertion $\mathcal{A}$ to some equivalent $\mathcal{A}'$, i.e. $[\![\mathcal{A}]\!]_t \rho\varepsilon = [\![\mathcal{A}']\!]_t \rho\varepsilon$ holds. An example could be reducing $\mathsf{true} \wedge X$ to simply X. From Definition 3.1.3 (Assertion semantics), we can immediately infer the following reductions:

$\mathsf{false} \wedge \mathcal{A}$	reduces to	false
$\mathsf{true} \wedge \mathcal{A}$	reduces to	$\mathcal{A}$
$\langle\alpha\rangle\mathsf{false}$	reduces to	false
$(\exists d : D.\mathcal{A})$	reduces to	e , provided $d \notin FV(\mathcal{A})$

And similar for $\vee$, $[\cdot]$ and $\forall$. Here, $FV(\mathcal{A})$ is the set of free data variables in $\mathcal{A}$.

This is slightly different from the definition of Andersen, in that the reductions are applied recursively to the assertions, i.e. not only can we reduce $X \vee \mathsf{false}$ to X, but also $X \vee (Y \vee \mathsf{true})$ to $X \vee Y$. This is rather straightforward and we believe recursive application is what Andersen intended.

The definition of simple evaluation is as follows:

Definition 6.1.1 (Simple evaluation)**.** Let $\mathcal{E} \downarrow X(d)$ be a top assertion. *Simple evaluation*, applied to this top assertion, denoted $\mathsf{SE}(\mathcal{E} \downarrow X(d))$, is defined as follows:

$$\mathsf{SE}(\mathcal{E} \downarrow X(d)) = \mathsf{SE}(\mathcal{E}) \downarrow X(d),$$

where simple evaluation on parameterized modal equation systems is defined as

$$\begin{aligned} \mathsf{SE}(\epsilon) &= \epsilon \\ \mathsf{SE}((\sigma X(d : D) = \mathcal{A})\ \mathcal{E}) &= (\sigma X(d : D) = \mathsf{SE}(\mathcal{A}))\ \mathsf{SE}(\mathcal{E}), \end{aligned}$$

and simple evaluation on assertions is defined as

$$\begin{aligned} \mathsf{SE}(\mathsf{false} \wedge \mathcal{A}) &= \mathsf{false} \\ \mathsf{SE}(\mathsf{false} \vee \mathcal{A}) &= \mathsf{SE}(\mathcal{A}) \\ \mathsf{SE}(\mathsf{true} \wedge \mathcal{A}) &= \mathsf{SE}(\mathcal{A}) \\ \mathsf{SE}(\mathsf{true} \vee \mathcal{A}) &= \mathsf{true} \\ \mathsf{SE}(\langle af\rangle\mathsf{false}) &= \mathsf{false} \\ \mathsf{SE}([af]\mathsf{true}) &= \mathsf{true} \\ \mathsf{SE}((\exists d : D.\mathcal{A})) &= \mathsf{SE}(\mathcal{A}) && \text{if } d \notin FV(\mathcal{A}) \\ \mathsf{SE}((\forall d : D.\mathcal{A})) &= \mathsf{SE}(\mathcal{A}) && \text{if } d \notin FV(\mathcal{A}) \\ \mathsf{SE}(\mathcal{A}) &= \mathcal{A} && \text{in all other cases.} \end{aligned}$$

Here, $FV(\mathcal{A})$ denotes the set of free variables occurring in $\mathcal{A}$.

6.2 Reachability analysis

The reachability analysis reduction (Andersen [1]) aims to remove parts of parameterized modal equation systems that do not influence the recursion variable we are interested in. For a top assertion $\mathcal{E} \downarrow X(d)$, if a variable Y is bound in $\mathcal{E}$ but is seen not to influence the value of X, we can remove its defining equation from $\mathcal{E}$. We approximate "influence" by a dependency graph: a variable Y *can* influence a variable X, if Y occurs in X's defining equation, or if Y can influence a variable Z which can influence X (transitivity).

The dependency graph of a parameterized modal equation system is a graph with the recursion variables as vertices and there is an edge from X to Y if Y occurs in the defining equation for X.

Definition 6.2.1 (Dependency graph)**.** Let $(\sigma_1 X_1(d_1 : D_1) = \mathcal{A}_1) \dots (\sigma_n X_n(d_n : D_n) = \mathcal{A}_n)$ be a parameterized modal equation system $\mathcal{E}$. The *dependency graph* of $\mathcal{E}$ is a graph (V, E), where $V = \{X_1, \dots, X_n\}$ and $(X_i, X_j) \in E$, if and only if, $X_j \in \mathsf{occ}(\mathcal{A}_i)$.

A variable's value can depend on the values of the variables in its defining equation. In turn, those may depend on the variables in their defining equations, and so on. These form a path in the dependency graph. Observe that variables to which there is *no* path from a given variable X, cannot influence X's value. So, we can reduce a parameterized modal equation system when variables in its dependency graph are *unreachable* from the top variable. These parts can be removed from the parameterized modal equation system without influencing the top variable's value.

Definition 6.2.2 (Reachability analysis)**.** Let $\mathcal{E} \downarrow X(d)$ be a top assertion. Let (V, E) be the dependency graph of $\mathcal{E}$ and let U be the unreachable variables in that graph: the set of vertices for which there is no path from X to that variable. Applying the *reachability analysis reduction* on $\mathcal{E} \downarrow X(d)$, denoted $\mathsf{RA}(\mathcal{E} \downarrow X(d))$, is defined as $\mathsf{rm}(U, \mathcal{E}) \downarrow X(d)$, where the function $\mathsf{rm}(U, \mathcal{E})$ removes the defining equations for a set of variables U from a parameterized modal equation system $\mathcal{E}$. It is defined as follows:

$$\mathsf{rm}(U, \epsilon) = \epsilon$$

$$\mathsf{rm}(U, (\sigma X(d : D) = \mathcal{A})\ \mathcal{E}) = \begin{cases} \mathsf{rm}(U, \mathcal{E}) & \text{if } X \in U \\ (\sigma X(d : D) = \mathcal{A})\ \mathsf{rm}(U, \mathcal{E}) & \text{otherwise} \end{cases}$$

6.3 Constant propagation

Constant propagation (Andersen [1]) replaces all occurrences of a variable with that variable's definition, if that definition is a constant. Since the definition is a constant, the variable and the definition are always equal and thus we may replace one with the other.

We expand this principle to include all assertions with no recursion variables in them, as these assertions are independent of the recursion variable environment ρ.

Constant propagation is formally defined as follows:

Definition 6.3.1 (Constant propagation)**.** Let $(\sigma X(d : D) = \mathcal{A})$ be an equation in parameterized modal equation system $\mathcal{E}$, where no recursion variables occur in $\mathcal{A}$: $occ(\mathcal{A}) = \emptyset$ holds. The application of *constant propagation* for this equation to $\mathcal{E}$, denoted $\mathsf{CP}(X(d), \mathcal{A}, \mathcal{E})$, is defined as follows:

$$\mathsf{CP}(X(d), \mathcal{A}, \epsilon) = \epsilon$$

$$\mathsf{CP}(X(d), \mathcal{A}, (\sigma' Y(e : E) = \mathcal{A}')\ \mathcal{E}) = (\sigma' Y(e : E) = \mathsf{CP}(X(d), \mathcal{A}, \mathcal{A}')\ \mathsf{CP}(X(d), \mathcal{A}, \mathcal{E}),$$

where constant propagation on assertions is defined as:

$$\begin{aligned}
\mathsf{CP}(X(d), \mathcal{A}, Y(e)) &= \begin{cases} \mathcal{A}[e/d] & \text{if } X = Y \\ Y(e) & \text{otherwise} \end{cases} \\
\mathsf{CP}(X(d), \mathcal{A}, b) &= b \\
\mathsf{CP}(X(d), \mathcal{A}, \mathcal{A}_1 \vee \mathcal{A}_2) &= \mathsf{CP}(X(d), \mathcal{A}, \mathcal{A}_1) \vee \mathsf{CP}(X(d), \mathcal{A}, \mathcal{A}_2) \\
\mathsf{CP}(X(d), \mathcal{A}, \langle af \rangle \mathcal{A}') &= \langle af \rangle \mathsf{CP}(X(d), \mathcal{A}, \mathcal{A}') \\
\mathsf{CP}(X(d), \mathcal{A}, (\exists e : E \,.\, \mathcal{A}')) &= (\exists e : E \,.\, \mathsf{CP}(X(d), \mathcal{A}, \mathcal{A}'))
\end{aligned}$$

6.4 Unguardedness removal

Unguardedness removal (Andersen [1]) is similar to constant propagation, in that it replaces occurrences of a variable with its definition. The main difference is that the defining equation of that variable can be anything and there is a restriction on which assertions it may be replaced in. In constant propagation, one may replace every occurrence of some variable X once it is established that X's defining assertion is constant. In unguardedness removal, one may replace occurrences of *any* X by its defining assertion, but *only* in assertions in which X does not occur after a modal operator, and only in equations *before* the defining equation of X.

We call the set of variables occurring after a modal operator, the "guarded variables".

Definition 6.4.1 (Guarded variables)**.** Let $\mathcal{A}$ be an assertion. The set of *guarded variables* of $\mathcal{A}$, denoted $\mathsf{guarded}(\mathcal{A})$, is defined as follows:

$$\begin{aligned}
\mathsf{guarded}(X(d)) &= \emptyset \\
\mathsf{guarded}(b) &= \emptyset \\
\mathsf{guarded}(\mathcal{A}_1 \vee \mathcal{A}_2) &= \mathsf{guarded}(\mathcal{A}_1) \cup \mathsf{guarded}(\mathcal{A}_2) \\
\mathsf{guarded}(\exists d : D \,.\, \mathcal{A}) &= \mathsf{guarded}(\mathcal{A}) \\
\mathsf{guarded}(\langle af \rangle \mathcal{A}) &= \mathsf{occ}(\mathcal{A})
\end{aligned}$$

Now, the unguardedness removal reduction replaces unguarded variables with their definitions:

Definition 6.4.2 (Unguardedness removal)**.** Let $(\sigma X(d : D) = \mathcal{A})$ and $(\sigma' Y(e : E) = \mathcal{A}')$ be equations in parameterized modal equation system $\mathcal{E}$, where X's defining equation occurs *after* Y's, and X does not occur guarded in $\mathcal{A}'$: $X \notin \mathsf{guarded}(\mathcal{A}')$ holds. The application of *unguardedness removal* for the defining equation of X to the defining equation for Y in $\mathcal{E}$, denoted $\mathsf{UR}(X(d), \mathcal{A}, Y, \mathcal{E})$, is defined as follows:

$$\begin{aligned}
\mathsf{UR}(X(d), \mathcal{A}, Y, \epsilon) &= \epsilon \\
\mathsf{UR}(X(d), \mathcal{A}, Y, (\sigma'' Z(f : F) = \mathcal{A}'')\ \mathcal{E}) &= \begin{cases} (\sigma' Y(e : E) = \mathsf{UR}(X(d), \mathcal{A}, \mathcal{A}'))\ \mathcal{E} & \text{if } Y = Z \\ (\sigma'' Z(f : F) = \mathcal{A}'')\ \mathsf{UR}(X(d), \mathcal{A}, Y) & \text{otherwise,} \end{cases}
\end{aligned}$$

where unguardedness removal on assertions is defined as:

$$\begin{aligned}
\mathsf{UR}(X(d), \mathcal{A}, Y(e)) &= \begin{cases} \mathcal{A}[e/d] & \text{if } X = Y \\ Y(e) & \text{otherwise} \end{cases} \\
\mathsf{UR}(X(d), \mathcal{A}, b) &= b \\
\mathsf{UR}(X(d), \mathcal{A}, \mathcal{A}_1 \vee \mathcal{A}_2) &= \mathsf{UR}(X(d), \mathcal{A}, \mathcal{A}_1) \vee \mathsf{UR}(X(d), \mathcal{A}, \mathcal{A}_2) \\
\mathsf{UR}(X(d), \mathcal{A}, \langle af \rangle \mathcal{A}') &= \langle af \rangle \mathcal{A}' \\
\mathsf{UR}(X(d), \mathcal{A}, (\exists e : E \,.\, \mathcal{A}')) &= (\exists e : E \,.\, \mathsf{UR}(X(d), \mathcal{A}, \mathcal{A}'))
\end{aligned}$$

This is rather similar to Definition 6.3.1 (Constant propagation). A notable difference is that we do not automatically replace all occurrences of X in *every* assertion where X does not occur guarded – we leave some control to some higher decision procedure, as $\mathcal{A}$ may be rather complex.

Note that it is possible for infinitely many unfoldings to occur, when the variables being replaced occur in each other's definitions in a cyclical manner. A trivial example is a parameterized modal equation system containing an equation $(\sigma X(d : D) = X(d))$, where X may be replaced by itself. We use the dependency graph we introduced in section 6.2 and maintain the set of edges $E' \subseteq E$ where $(X_i, X_j) \in E'$ if X_j occurs in the defining assertion of X_i and also X_j does *not* occur guarded in the defining equation of X_i. When E' contains a cycle, we must take care not to do infinite unfoldings. Detection of a cycle in a graph can be done by any Strongly Connected Component (SCC) algorithm.

In the paper by Andersen, the condition that substitution is only allowed *backwards*, is not explicitly stated. This is no small detail, as forward substitution is not generally sound. We show this using an example:

Example 6.4.3 (Unguardedness removal, forward substitution counterexample)**.** Let $(\mu X = Y)\ (\nu Y = X)$ be the modal equation system $\mathcal{E}$. The semantics of $\mathcal{E}$ for any environment ρ and linear process equation P, is the environment $\rho[X := \mathsf{false}][Y := \mathsf{false}]$. Using forward substitution, however, we could rewrite $\mathcal{E}$ to $(\mu X = Y)\ (\nu Y = Y)$, the semantics of which is $\rho[X := \mathsf{true}][Y := \mathsf{true}]$. This reduction is therefore unsound.

Note that with *backwards* substitution, the semantics of the reduced modal equation system is indeed invariant: $\mathcal{E}$ reduces to $(\mu X = X)\ (\nu Y = X)$, which has equal semantics.

This reduction is similar to the substitution step in Gauss elimination on boolean equation systems (Mader [8]).

6.5 Trivial equation elimination

Trivial equation elimination (Andersen [1]) reduces equations which do not depend on other recursion variables than the one it binds and thus can be solved by themselves. For example, the equation $(\mu X = X)$ can be reduced to $(\mu X = \mathsf{false})$. There are four such rules that Andersen notes on modal equation systems, leading to the following reduction rules on parameterized modal equation systems:

$$\begin{aligned}
(\mu X(d : D) = X(e)) &\quad \text{reduces to} \quad (\mu X(d : D) = \mathsf{false}) \\
(\mu X(d : D) = \langle af \rangle X(e)) &\quad \text{reduces to} \quad (\mu X(d : D) = \mathsf{false}) \\
(\nu X(d : D) = X(e)) &\quad \text{reduces to} \quad (\mu X(d : D) = \mathsf{true}) \\
(\nu X(d : D) = [af] X(e)) &\quad \text{reduces to} \quad (\mu X(d : D) = \mathsf{true})
\end{aligned}$$

This yields equations with constant defining assertions, enabling constant propagation reduction (cf. Section 6.3). Also, the defining assertions do not depend on the data parameter, enabling parameter elimination (cf. Section 6.6).

6.6 Parameter elimination

Parameter elimination for parameterized modal equation systems is inspired by parameter elimination on parameterized boolean equation systems (Orzan *et al.* [9]). It aims to remove a data parameter from a recursion variable that does not occur in its defining equation, e.g. $(\mu X(b :$

$\mathsf{Bool}, n : \mathsf{Nat}) = b)$ can be reduced to $(\mu X(b : \mathsf{Bool}) = b)$, as the parameter n does not influence the assertion b.

Formally, we define the parameter elimination reduction as follows:

Definition 6.6.1 (Parameter elimination)**.** Let $(\sigma X(d : D, d' : D') = \mathcal{A})$ be an equation in the parameterized modal equation system $\mathcal{E}$, where $d \notin FV(\mathcal{A})$. Here, $FV(\mathcal{A})$ denotes the set of free data variables occurring in assertion $\mathcal{A}$.

The parameter elimination reduction of X's d parameter applied to $\mathcal{E}$, denoted $\mathsf{PE}(X, d, \mathcal{E})$, is defined as:

$$\mathsf{PE}(X, d, \epsilon) = \epsilon$$

$$\mathsf{PE}(X, d, (\sigma' Y(e : E) = \mathcal{A}')\ \mathcal{E}) = \begin{cases} (\sigma X(d' : D') = \mathcal{A})\ \mathcal{E} & \text{if } X = Y \\ (\sigma' Y(e : E) = \mathcal{A}')\ \mathsf{PE}(X, d, \mathcal{E}) & \text{otherwise} \end{cases}$$

6.7 Action formulae simplification

The action formulae simplification is essentially boolean simplification. Because of the large similarity between action formulae and boolean expressions, we can apply common boolean reductions to action formulae, treating multi-actions and boolean expressions on user-defined data as variables. Most notably, the quotienting technique requires action formulae to be in disjunctive normal form (cf. Definition 2.2.14 (Action formulae disjunctive normal form)). To rewrite assertions to disjunctive normal form, we can use algorithms designed for boolean expressions, which usually also employ some boolean reduction techniques.

Conclusion

We conclude that we have successfully overcome the limitation of earlier compositional model checking procedures, that the processes at hand have enumerable state spaces. Not only did we generalize assertions to include quantifiers and modal operators with action formulae instead of single actions, we most importantly extended the quotienting procedure to linear process equations, including infinite state spaces.

The heart of this thesis is in quotienting linear process equations and process operators, which is to move part of the model checking problem from the model side of the model checking equation to the property side. This was successful: we have defined how to calculate quotienting a linear process equation or process operator from a property. This yields a new model checking problem, where the linear process equation or process operator has been removed from the model side of the model checking equation. We defined how to calculate this new model checking equation.

A very important part is to show that the quotienting procedure is sound: the model checking problem we obtain by quotienting out a process operator or a linear process equation, has the same solution as the original model checking problem. In Section 4.4, we show this is indeed true.

We have not yet quantified how good our compositional model checking procedure is at reducing the complexity of model checking problems. We recommend further research be done to provide a working implementation, more research on reducing the properties it produces, and experiments to the effectiveness of the compositional model checking procedure.

Future research

The most important areas for future research are to implement the technique we described, and to improve the property reduction step of compositional model checking.

Building an efficient implementation is not only the first step to quantify how useful this compositional model checking technique is in the real world, it should also serve as an inspiration to property reduction techniques. The compositional model checking technique is only as useful as the reductions it can perform on the property side of the model checking equations. The quotienting procedure merely moves the problem from the model side of the equation to the property side, where property reductions do the actual work of making the model checking problem smaller and thus easier to solve. Properties obtained by quotienting components of real-life systems from real-life properties, should provide a good indication what reductions are useful.

We also found that quotienting the allow operator and the abstraction operator, requires building a set of actions that can be performed by the process P_2 which runs independently from the process description the process operator surrounds. This was something we wanted to avoid, but we were unable to express this in action formulae otherwise. We wanted to express that, say, some action a occurs simultaneously with any action α, where a is performed by one parallel process and α by the other. We did this by renaming the actions in α, but in the case of the allow and abstraction operators, this meant we had to enumerate all these possible α's, and thus inspect P_2. Some way of expressing these actions without inspecting P_2 would be more true to the spirit of compositional model checking, where we *only* inspect one component at a time.

Finally, the quotienting procedure could be expanded to work on the full range of action formulae, including quantifiers over data. Though it proved very convenient to require action formulae to be in disjunctive normal form, we hope future research will enable the quotienting procedure to work with arbitrary action formulae.

References

[1] H. R. Andersen, *Partial model checking (extended abstract)*, In Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1995, pp. 398–407.

[2] J. F. Groote, A. H. J. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. J. Van Weerdenburg, *The formal specification language mCRL2*, In Proceedings of the Dagstuhl Seminar, MIT Press, 2007.

[3] J. F. Groote and M. A. Reniers, *Modelling and analysis of communicating systems*, rev. 2601, Eindhoven University of Technology, Eindhoven, The Netherlands, 2009.

[4] J. F. Groote and T. A. C. Willemse, *Parameterised boolean equation systems*, In Theoretical Computer Science, Elsevier, 2004, pp. 332–369.

[5] ———, *Model-checking processes with data*, In Science of Computer Programming, Elsevier, 2005, pp. 251–273.

[6] F. Lang and R. Mateescu, *Partial Model Checking using Networks of Labelled Transition Systems and Boolean Equation Systems*, Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2012.

[7] J.-L. Lassez, V. L. Nguyen, and E. A. Sonenberg, *Fixed point theorems and semantics: A folk tale*, Inf. Process. Lett. **14** (1982), no. 3, 112–116.

[8] A. Mader, *Verification of modal properties using boolean equation systems*, Edition versal 8, Bertz Verlag, Berlin, Germany, 1997.

[9] S. Orzan, W. Wesselink, and T. A. C. Willemse, *Static analysis techniques for parameterised boolean equation systems*, Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Berlin, Heidelberg), TACAS '09, Springer-Verlag, 2009, pp. 230–245.

[10] K. Schneider, *Verification of reactive systems: Formal methods and algorithms*, SpringerVerlag, 2004.

[11] A. Tarski, *A Lattice-Theoretical Fixpoint Theorem and its Applications*, Pacific J. Math. **5** (1955), no. 2, 285–309.

[12] K. van der Pol, *Adapting the partial model checking technique for use in the mCRL2 toolkit*, 2012.

A Buffer example (4) calculations

This section shows the calculation of quotienting on the running example of the buffer.

$$(\mathcal{E} \downarrow X)/B_1(\text{empty})$$

The first steps are quote simple. The state parameter of the process we are quotienting out, becomes a parameter of each equation of $\mathcal{E}$. The quotienting simply distributes over boolean connectives and quantifiers.

$$\left(\begin{array}{rl} \nu X(s : D \cup \{\text{empty}\}) = & ([\mathsf{true}]X)/B_1(s) \\ & \wedge\ (\forall m : D\ .\ ([\text{in}(m)]Y(m))/B_1(s)) \\ \mu Y(m : D, s : D \cup \{\text{empty}\}) = & ([\neg\text{out}(m)]Y(m))/B_1(s) \\ & \wedge\ (\langle\mathsf{true}\rangle\mathsf{true})/B_1(s) \end{array}\right) \downarrow X(\text{empty})$$

The box modalities give rise to the box variant of Quotient2, denoted $\mathsf{Quotient2}_{[]}$.

$$\left(\begin{array}{l} \nu X(s : D \cup \{\text{empty}\}) = \\ \quad \bigwedge_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{in}(d),\mathsf{true})} \mathsf{Quotient2}_{[]}(b_1,b_2,b_3,af_2,X,B_1(s),1) \\ \quad \wedge \bigwedge_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{out}(s),\mathsf{true})} \mathsf{Quotient2}_{[]}(b_1,b_2,b_3,af_2,X,B_1(s),2) \\ \quad \wedge (\forall m : D\ . \\ \qquad \bigwedge_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{in}(s),\text{in}(m))} \mathsf{Quotient2}_{[]}(b_1,b_2,b_3,af_2,Y(m),B_1(s),1) \\ \qquad \wedge \bigwedge_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{out}(s),\text{in}(m))} \mathsf{Quotient2}_{[]}(b_1,b_2,b_3,af_2,Y(m),B_1(s),2) \\ \quad) \\ \mu Y(m : D, s : D \cup \{\text{empty}\}) = \\ \quad \bigwedge_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{in}(d),\neg\text{out}(m))} \mathsf{Quotient2}_{[]}(b_1,b_2,b_3,af_2,Y(m),B_1(s),1) \\ \quad \wedge \bigwedge_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{out}(s),\neg\text{out}(m))} \mathsf{Quotient2}_{[]}(b_1,b_2,b_3,af_2,Y(m),B_1(s),2) \\ \quad \wedge (\bigvee_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{in}(d),\mathsf{true})} \mathsf{Quotient2}(b_1,b_2,b_3,af_2,\mathsf{true},B_1(s),1) \\ \qquad \vee \bigvee_{(b_1,b_2,b_3,af_2)\in\mathsf{split2}(\text{out}(s),\mathsf{true})} \mathsf{Quotient2}(b_1,b_2,b_3,af_2,\mathsf{true},B_1(s),2) \\ \quad) \end{array}\right) \downarrow X(\text{empty})$$

We calculate these sets split2, Definition 4.2.5 (split2). For single actions, there are two cases: one process does the action and the remainder does τ or nothing, and vice-versa. This will become apparent when we expand the definition of Quotient2, Definition 4.2.6 (Quotient2).

$$
\begin{pmatrix}
\nu X(s : D \cup \{\text{empty}\}) = \\
\quad \mathsf{Quotient2}_{[]}(\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, X, B_1(s), 1) \\
\quad \wedge \mathsf{Quotient2}_{[]}(\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, X, B_1(s), 2) \\
\quad \wedge (\forall m : D \ . \\
\qquad \mathsf{Quotient2}_{[]}(\tau = \text{in}(d), \tau = \text{in}(m), \tau = \tau, \text{in}(m), Y(m), B_1(s), 1) \\
\qquad \wedge \mathsf{Quotient2}_{[]}(\text{in}(m) = \text{in}(d), \text{in}(m) = \text{in}(m), \text{in}(m) = \tau, \tau, Y(m), B_1(s), 1) \\
\qquad \wedge \mathsf{Quotient2}_{[]}(\tau = \text{out}(s), \tau = \text{in}(m), \tau = \tau, \text{in}(m), Y(m), B_1(s), 2) \\
\qquad \wedge \mathsf{Quotient2}_{[]}(\text{in}(m) = \text{out}(s), \text{in}(m) = \text{in}(m), \text{in}(m) = \tau, \tau, Y(m), B_1(s), 2) \\
\quad) \\
\mu Y(m : D, s : D \cup \{\text{empty}\}) = \\
\quad \mathsf{Quotient2}_{[]}(\text{in}(d) \not\sqsubseteq \text{out}(m), \mathsf{true}, \mathsf{false}, \mathsf{true}, Y(m), B_1(s), 1) \\
\quad \wedge \mathsf{Quotient2}_{[]}(\tau = \text{in}(d), \tau \neq \text{out}(m), \tau = \tau, \neg\text{out}(m), Y(m), B_1(s), 1) \\
\quad \wedge \mathsf{Quotient2}_{[]}(\text{out}(m) = \text{in}(d), \text{out}(m) \neq \text{out}(m), \text{out}(m) = \tau, \neg\tau, Y(m), B_1(s), 1) \\
\quad \wedge \mathsf{Quotient2}_{[]}(\text{out}(s) \not\sqsubseteq \text{out}(m), \mathsf{true}, \mathsf{false}, \mathsf{true}, Y(m), B_1(s), 2) \\
\quad \wedge \mathsf{Quotient2}_{[]}(\tau = \text{out}(s), \tau = \tau, \tau \neq \text{out}(m), \neg\text{out}(m), Y(m), B_1(s), 2) \\
\quad \wedge \mathsf{Quotient2}_{[]}(\text{out}(m) = \text{out}(s), \text{out}(m) \neq \text{out}(m), \text{out}(m) = \tau, \neg\tau, Y(m), B_1(s), 2) \\
\quad \wedge (\mathsf{Quotient2}(\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, B_1(s), 1) \\
\qquad \vee \mathsf{Quotient2}(\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true}, B_1(s), 2)) \\
\quad)
\end{pmatrix}
\downarrow X(\text{empty})
$$

We expand these Quotient2 expressions. Each gives rise to three disjuncts or conjuncts: one for when B_1 does part of the transition and the reaminder process does the rest, one for when B_1 does the entire transition and the remainder process stays idle, and one when B_1 itself stays idle and the remainder process does the entire transition.

$$
\begin{array}{l}
\left(\begin{array}{l}
\nu X(s : D \cup \{\text{empty}\}) = \\
\quad (\forall d : D.s \approx \text{empty} \wedge \mathsf{true} \;\Rightarrow\; [\mathsf{true}](X/B_1(d))) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \mathsf{true} \wedge \mathsf{true} \;\Rightarrow\; (X/B_1(d))) \\
\quad\quad \wedge (\mathsf{true} \;\Rightarrow\; [\mathsf{true}](X/B_1(s))) \\
\quad \wedge (s \not\approx \text{empty} \wedge \mathsf{true} \;\Rightarrow\; [\mathsf{true}](X/B_1(\text{empty}))) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge \mathsf{true} \wedge \mathsf{true} \;\Rightarrow\; (X/B_1(\text{empty}))) \\
\quad\quad \wedge (\mathsf{true} \;\Rightarrow\; [\mathsf{true}](X/B_1(s))) \\
\quad \wedge (\forall m : D\ . \\
\quad\quad (\forall d : D.s \approx \text{empty} \wedge \tau = \text{in}(d) \;\Rightarrow\; [\text{in}(m)](Y(m)/B_1(d))) \\
\quad\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \tau = \text{in}(d) \wedge \tau = \text{in}(m) \;\Rightarrow\; (Y(m)/B_1(d))) \\
\quad\quad\quad \wedge (\tau = \tau \;\Rightarrow\; [\text{in}(m)](Y(m)/B_1(s))) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \text{in}(m) = \text{in}(d) \;\Rightarrow\; [\tau](Y(m)/B_1(d))) \\
\quad\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \text{in}(m) = \text{in}(d) \wedge \text{in}(m) = \text{in}(m) \;\Rightarrow\; (Y(m)/B_1(d))) \\
\quad\quad\quad \wedge (\text{in}(m) = \tau \;\Rightarrow\; [\tau](Y(m)/B_1(s))) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge \tau = \text{out}(s) \;\Rightarrow\; [\text{in}(m)](Y(m)/B_1(\text{empty}))) \\
\quad\quad\quad \wedge (s \not\approx \text{empty} \wedge \tau = \text{out}(s) \wedge \tau = \text{in}(m) \;\Rightarrow\; (Y(m)/B_1(\text{empty}))) \\
\quad\quad\quad \wedge (\tau = \tau \;\Rightarrow\; [\text{in}(m)](Y(m)/B_1(s))) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge \text{in}(m) = \text{out}(s) \;\Rightarrow\; [\tau](Y(m)/B_1(\text{empty}))) \\
\quad\quad\quad \wedge (s \not\approx \text{empty} \wedge \text{in}(m) = \text{out}(s) \wedge \text{in}(m) = \text{in}(m) \;\Rightarrow\; (Y(m)/B_1(\text{empty}))) \\
\quad\quad\quad \wedge (\text{in}(m) = \tau \;\Rightarrow\; [\tau](Y(m)/B_1(s))) \\
\quad)
\end{array}\right) \\
\mu Y(m : D, s : D \cup \{\text{empty}\}) = \\
\quad (\forall d : D.s \approx \text{empty} \wedge \text{in}(d) \not\sqsubseteq \text{out}(m) \;\Rightarrow\; [\mathsf{true}](Y(m)/B_1(d))) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \text{in}(d) \not\sqsubseteq \text{out}(m) \wedge \mathsf{true} \;\Rightarrow\; (Y(m)/B_1(d))) \\
\quad\quad \wedge (\mathsf{false} \;\Rightarrow\; [\mathsf{true}](Y(m)/B_1(s))) \\
\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \tau = \text{in}(d) \;\Rightarrow\; [\neg\text{out}(m)](Y(m)/B_1(d))) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \tau = \text{in}(d) \wedge \tau \neq \text{out}(m) \;\Rightarrow\; (Y(m)/B_1(d))) \\
\quad\quad \wedge (\tau = \tau \;\Rightarrow\; [\neg\text{out}(m)](Y(m)/B_1(s))) \\
\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \text{out}(m) = \text{in}(d) \;\Rightarrow\; [\neg\tau](Y(m)/B_1(d))) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge \text{out}(m) = \text{in}(d) \wedge \text{out}(m) \neq \text{out}(m) \;\Rightarrow\; (Y(m)/B_1(d))) \\
\quad\quad \wedge (\text{out}(m) = \tau \;\Rightarrow\; [\neg\tau](Y(m)/B_1(s))) \\
\quad \wedge (s \not\approx \text{empty} \wedge \text{out}(s) \not\sqsubseteq \text{out}(m) \;\Rightarrow\; [\mathsf{true}](Y(m)/B_1(\text{empty}))) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge \text{out}(s) \not\sqsubseteq \text{out}(m) \wedge \mathsf{true} \;\Rightarrow\; (Y(m)/B_1(\text{empty}))) \\
\quad\quad \wedge (\mathsf{false} \;\Rightarrow\; [\mathsf{true}](Y(m)/B_1(s))) \\
\quad \wedge (s \not\approx \text{empty} \wedge \tau = \text{out}(s) \;\Rightarrow\; [\neg\text{out}(m)](Y(m)/B_1(\text{empty}))) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge \tau = \text{out}(s) \wedge \tau \neq \text{out}(m) \;\Rightarrow\; (Y(m)/B_1(\text{empty}))) \\
\quad\quad \wedge (\tau = \tau \;\Rightarrow\; [\neg\text{out}(m)](Y(m)/B_1(s))) \\
\quad \wedge (s \not\approx \text{empty} \wedge \text{out}(m) = \text{out}(s) \;\Rightarrow\; [\neg\tau](Y(m)/B_1(\text{empty}))) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge \text{out}(m) = \text{out}(s) \wedge \text{out}(m) \neq \text{out}(m) \;\Rightarrow\; (Y(m)/B_1(\text{empty}))) \\
\quad\quad \wedge (\text{out}(m) = \tau \;\Rightarrow\; [\neg\tau](Y(m)/B_1(s))) \\
\quad \wedge ((\exists d : D.s \approx \text{empty} \wedge \mathsf{true} \;\wedge\; \langle\mathsf{true}\rangle(\mathsf{true}/B_1(d))) \\
\quad\quad\quad \vee (\exists d : D.s \approx \text{empty} \wedge \mathsf{true} \wedge \mathsf{true} \;\wedge\; (\mathsf{true}/B_1(d))) \\
\quad\quad\quad \vee (\mathsf{true} \;\wedge\; \langle\mathsf{true}\rangle(\mathsf{true}/B_1(s))) \\
\quad\quad \vee (s \not\approx \text{empty} \wedge \mathsf{true} \;\wedge\; \langle\mathsf{true}\rangle(\mathsf{true}/B_1(\text{empty}))) \\
\quad\quad\quad \vee (s \not\approx \text{empty} \wedge \mathsf{true} \wedge \mathsf{true} \;\wedge\; (\mathsf{true}/B_1(\text{empty}))) \\
\quad\quad\quad \vee (\mathsf{true} \;\wedge\; \langle\mathsf{true}\rangle(\mathsf{true}/B_1(s))) \\
\quad)
\end{array}
$$

$$\downarrow X(\text{empty})$$

We reduce this by some simple boolean simplification. In the previous expression, all expressions which are obviously equal to false, are highlighted in red. We remove conjuncts which equal true (especially the many occurrences of expressions similar to $(\forall d : D.\mathsf{false} \Rightarrow \dots)$) and we remove duplicate expressions. We reduce the multi-action equality $a(d) = a(d')$ to $d \approx d'$, by unfolding the definition of multi-action equality.

$$
\left(\begin{array}{l}
\nu X(s : D \cup \{\text{empty}\}) = \\
\quad (\forall d : D.s \approx \text{empty} \;\Rightarrow\; [\mathsf{true}]X(d)) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \;\Rightarrow\; X(d)) \\
\quad\quad \wedge [\mathsf{true}]X(s) \\
\quad \wedge (s \not\approx \text{empty} \;\Rightarrow\; [\mathsf{true}]X(\text{empty})) \\
\quad\quad \wedge (s \not\approx \text{empty} \;\Rightarrow\; X(\text{empty})) \\
\quad \wedge (\forall m : D\ . \\
\quad\quad [\text{in}(m)]Y(m, s) \\
\quad\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge m \approx d \;\Rightarrow\; [\tau]Y(m, d)) \\
\quad\quad\quad \wedge (\forall d : D.s \approx \text{empty} \wedge m \approx d \;\Rightarrow\; Y(m, d)) \\
\quad) \\
\mu Y(m : D, s : D \cup \{\text{empty}\}) = \\
\quad (\forall d : D.s \approx \text{empty} \;\Rightarrow\; [\mathsf{true}]Y(m, d)) \\
\quad\quad \wedge (\forall d : D.s \approx \text{empty} \Rightarrow\; Y(m, d)) \\
\quad\quad \wedge [\neg\text{out}(m)]Y(m, s) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \not\approx m \;\Rightarrow\; [\mathsf{true}]Y(m, \text{empty})) \\
\quad\quad \wedge (s \not\approx \text{empty} \wedge s \not\approx m \;\Rightarrow\; Y(m, \text{empty})) \\
\quad \wedge (s \not\approx \text{empty} \wedge m \approx s \;\Rightarrow\; [\neg\tau]Y(m, \text{empty})) \\
\quad \wedge ((\exists d : D.s \approx \text{empty} \wedge \langle\mathsf{true}\rangle\mathsf{true}) \\
\quad\quad\quad \vee (\exists d : D.s \approx \text{empty}) \\
\quad\quad\quad \vee \langle\mathsf{true}\rangle\mathsf{true} \\
\quad\quad \vee (s \not\approx \text{empty}) \\
\quad\quad\quad \vee (s \not\approx \text{empty}) \\
\quad\quad\quad \vee \langle\mathsf{true}\rangle\mathsf{true} \\
\quad)
\end{array}\right) \downarrow X(\text{empty})
$$

With a little more insight, we can reduce this even more.

$$
\left(\begin{array}{l}
\nu X(s : D \cup \{\text{empty}\}) = \\
\quad (\forall s' : D \cup \{\text{empty}\}.[\mathsf{true}]X(s') \wedge X(s')) \\
\quad \wedge (\forall m : D\ . \\
\quad\quad [\text{in}(m)]Y(m, s) \\
\quad\quad\quad \wedge (s \approx \text{empty} \Rightarrow\; [\tau]Y(m, m) \wedge Y(m, m)) \\
\quad) \\
\mu Y(m : D, s : D \cup \{\text{empty}\}) = \\
\quad [\neg\text{out}(m)]Y(m, s) \\
\quad \wedge (\forall d : D.s \approx \text{empty} \;\Rightarrow\; [\mathsf{true}]Y(m, d) \wedge Y(m, d)) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \not\approx m \;\Rightarrow\; [\mathsf{true}]Y(m, \text{empty}) \wedge Y(m, \text{empty})) \\
\quad \wedge (s \not\approx \text{empty} \wedge s \approx m \;\Rightarrow\; [\neg\tau]Y(m, \text{empty}))
\end{array}\right) \downarrow X(\text{empty})
$$

Interestingly, the part that correspond to not being deadlocked, has reduces to true. This was to be expected, as the process B_1 we have quotiented out, contains no deadlocks. Also, we see some expected case distinction: the buffer can be empty or full, and if it is full, it can be filled with the element m or not. In this last case, the step where B_1 performs the out(m) transition is hidden from view. What remains is that the remainder process does not do any steps simultaneously, or at most only τ-steps, so as not to interfere with the out(m) action occurring.