Decomposing monolithic processes in a process algebra with multi-actions

Citation for published version (APA):
Laveaux, M., & Willemse, T. A. C. (2023). Decomposing monolithic processes in a process algebra with multi-actions. Journal of Logical and Algebraic Methods in Programming, 132, Article 100858. https://doi.org/10.1016/j.jlamp.2023.100858

Document license:
CC BY

DOI:
10.1016/j.jlamp.2023.100858

Document status and date:
Published: 01/04/2023

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 01. Dec. 2024
Decomposing monolithic processes in a process algebra with multi-actions

Maurice Laveaux\textsuperscript{a,\ast}, Tim A.C. Willemse\textsuperscript{a,b}

\textsuperscript{a} Eindhoven University of Technology, De Groene Loper 5, 5612 AE, Eindhoven, the Netherlands
\textsuperscript{b} ESI (TNO), Eindhoven, the Netherlands

\begin{abstract}
A monolithic process is a single recursive equation with data parameters, which only uses non-determinism, action prefixing, and recursion. We present a technique that decomposes such a monolithic process into multiple processes where each process defines behaviour for a subset of the parameters of the monolithic process. For this decomposition we can show that a composition of these processes is strongly bisimilar to the monolithic process under a suitable synchronisation context. Minimising the resulting processes before determining their composition can be used to derive a state space that is smaller than the one obtained by a monolithic exploration. We apply the decomposition technique to several specifications to show that this works in practice. Finally, we prove that state invariants can be used to further improve the effectiveness of this decomposition technique.
\end{abstract}

\section{1. Introduction}

The mCRL2 language \cite{mCRL2} is a process algebra that can be used to specify the behaviour of communicating processes with data parameters. It has the usual ACP-style operators for modelling non-deterministic choice, sequential composition, parallel composition and recursion. A powerful yet somewhat unconventional language construct of mCRL2 is the \textit{multi-action}, which allows for specifying that atomic actions can happen simultaneously.

Specifications written in mCRL2 can be analysed using the corresponding mCRL2 toolset \cite{mCRL2toolset}. The mCRL2 toolset \cite{mCRL2toolset} translates a process specification to an equivalent monolithic recursive process, replacing all interleaving parallelism by non-determinism, action prefixing and recursion.

Translating a complicated process specification into a simpler normal form, in this case the monolithic process, has several advantages due to the fact that we only have to deal with a simple structure instead of the full process algebra in all subsequent analyses. First of all, the design and implementation of state space exploration algorithms can be greatly simplified. Furthermore, the design of effective static analysis techniques on the global behaviour of the specification is also easier.

Such static analysis techniques can help to reduce the \textit{size} of the state space underlying the monolithic process, where the size is the sum of the number of states and transitions. One example is a static analysis to detect live variables as presented...
in [3], where variables are live whenever their values influence the behaviour. Other static analysis are the elimination of constant parameters as presented in [4]. However, the static analysis techniques available at the moment are not always strong enough to mitigate the state space explosion problem for this monolithic process even though its state space can often be minimised with respect to some equivalence relation after state space exploration.

In this paper, we define a decomposition technique (which we refer to as a cleave) of a monolithic process. Our technique takes as input such a process and a partitioning of its data parameters, and it produces two new processes. To illustrate the idea, consider a machine that alternates between two modes, where switching modes has a certain delay. The behaviour of this machine is modelled by the labelled transition system in Fig. 1. Assume that this machine is described by a single recursive mCRL2 process with two parameters: a natural number representing the counter for the delay and a Boolean for representing the mode of the machine. Using the partition that ‘splits’ these two parameters, our technique will decompose this machine into two recursive processes (components) with their respective behaviour shown in Fig. 2. Observe that indeed the states of a component rely on only one of the two parameters. Note that the transition systems of both components include sync and tag actions that do not occur in the transition system of the original machine. These are generated by our technique and are needed to model the interface between the two components such that under a suitable synchronisation context the parallel composition of these components is equivalent (strongly bisimilar) to the original monolithic process.

Decomposing a monolithic process may help to partly sidestep the state space explosion that is due to the interleaving of parallel processes that is encoded in the monolithic process. This follows from the observation that the state spaces of the components resulting from a decomposition can be (much) smaller than the state space of the monolithic process; these may therefore be easier to obtain. By first minimising the state spaces of these components with respect to bisimilarity before computing their composition, also the state space of the latter remains (much) smaller than that of the monolithic process. Since strong bisimilarity is a congruence for all operators of mCRL2, the resulting state space is still strongly bisimilar to that of the monolithic process, meaning that no information is lost.

Theoretically, the main challenge in defining a decomposition technique is to ensure that it results in components that, when combined appropriately, behave indistinguishably from the monolithic process from which they were derived. This is the problem of finding a valid decomposition. We illustrate that there may be multiple valid decompositions of a monolithic process. The main practical challenge is therefore to identify a universally applicable decomposition technique that yields valid decompositions, and which is capable of sidestepping the state space explosion problem. The current paper extends on the previously published conference version [5] by showing that the techniques that we develop can, to a large extent, be automated. Moreover, we provide detailed proofs of the main results and include a more extensive set of experiments.

Summarising the contributions of our work are as follows:

- we formalise the notion of a decomposition (see Section 3) and the notion of validity of a decomposition,
- we present a generally applicable decomposition technique and provide sufficient conditions for this decomposition to be valid (see Section 4),
- we show that state invariants [6] can be used to obtain even smaller state spaces by restricting the interfaces of the components resulting from the decomposition (Section 5),
- we provide algorithms that, based on a user-defined selection of process parameters, extract two components from a monolithic process (see Section 6),
- we confirm the practical applicability of our techniques on several cases in (see Section 7).

Related work Several different techniques are related to this type of decomposition. Most notably, the work on decomposing Petri nets into a set of automata [7] also aims to speed up state space exploration by means of decomposition. However, Petri nets have a clear structure and lack (possibly complex) data expressions that must be taken into account. The work on functional decomposition [8] describes a technique to decompose a specification based on a partitioning of the action labels

![Fig. 1. Behaviour of a machine that alternates between two modes of operation. The transitions are labelled with multi-sets, indicated as \( \{ \} \), of actions.](image1)

![Fig. 2. Behaviour of the decomposition processes.](image2)
for a basic fragment of LOTOS, where processes have no data parameters. In [9] it was shown how this type of decomposition can be achieved for mCRL2 processes. These works use the structure of the original process specification to perform a decomposition with the intention to distribute the components, and are not necessarily interested in compositional verification techniques and it is not clear whether reductions can be achieved in this way. Furthermore, a decomposition technique was used in [6] to improve the efficiency of equivalence checking. However, that work considers processes that are already in parallel composition and further decomposes them based on the actions that occur in each component.

Decompositional minimisation is also related to compositional minimisation, in which the objective is to replace the state space of each component in a (given) parallel composition by an equivalent, smaller state space, while preserving the behaviour of the original specification [10,11]. A problem that is common to compositional minimisation and decompositional minimisation is that the size of the state spaces belonging to individual components summed together might still exceed the size of the original state space [12]. One way to (partly) avoid this is by specifying interface constraints (also known as environmental constraints or context constraints), see [13,14]. The state invariants in our work serve a similar purpose, but the mechanism is different since interface constraints are action-based whereas invariants are state-based. Another possibility is to find a more suitable order in which components are explored and minimised, since the order heavily influences the size of the intermediate state spaces. Heuristics for determining this order can be very effective in practice [15]; such heuristics are also relevant for the application of our decomposition technique.

One advantage of the decomposition technique over compositional minimisation is that our interfaces can be derived from the conditions present in the monolithic process. These interfaces can also be further strengthened with state invariants. Secondly, the components resulting from the decomposition are not limited to the user-defined processes present in the specification. Our decomposition technique is thus more flexible, and may yield more optimal compositions. Indeed, the case studies on which we report support both observations.

2. Preliminaries

We assume the existence of an abstract data theory that describes data sorts, where sorts are sometimes referred to as types in other contexts. Each sort $D$ has an associated non-empty semantic domain denoted by $\mathbb{D}$. The existence of sorts $\text{Bool}$ and $\text{Nat}$ with their associated Boolean ($\mathbb{B}$) and natural number ($\mathbb{N}$) semantic domains respectively, with standard operators is assumed. Furthermore, we assume the existence of an infinite set of sorted variables. We use $e : D$ to indicate that $e$ is an expression (or sort) of sort $D$. The set of expressions of an expression $e$ is denoted $\text{FV}(e)$, and a variable that is not free is called bound. An expression $e$ is closed iff $\text{FV}(e) = \emptyset$. A substitution $\sigma$ is a total function from variables to closed data expressions of their corresponding sort. For a substitution $\sigma$, we write $\sigma[x ← e]$ to denote the substitution $\sigma'$ such that $\sigma'(x) = e$ and for all $y \neq x$, we have $\sigma'(y) = \sigma(y)$. We use $\sigma(e)$ to denote the syntactic replacement of variables in expression $e$ by their substituted expression.

An interpretation function, denoted by $[\cdot]$, maps syntactic objects to values within their corresponding semantic domain. We assume that $[e]$ for closed expressions $e$ is already defined. Semantic objects are typeset in boldface to differentiate them from syntax, e.g., the semantics of expression $1 + 1$ is $2$. We denote data equivalence by $e \equiv f$, which is true iff $[e] = [f]$; for other operators we use the same symbol in both syntactic and semantic domains. Data equivalence is lifted to equivalence of substitutions in the usual way, i.e., $\sigma \equiv \sigma'$ iff for all variables $x$ we have $\sigma(x) \equiv \sigma'(x)$. We adopt the usual principle of substitutivity; i.e., for all substitutions $\sigma, \sigma'$ and expressions $e$ it holds that if $\sigma \equiv \sigma'$ then $\sigma(e) \equiv \sigma'(e)$.

We denote a vector of length $n + 1$ by $d = (d_0, \ldots, d_n)$. The empty vector is denoted by $. Two vectors are equivalent, denoted by $(d_0, \ldots, d_n) \equiv (e_0, \ldots, e_n)$, iff their elements are pairwise equivalent, i.e., $d_i \equiv e_i$ for all $0 \leq i \leq n$. Given a vector $(d_0, \ldots, d_n)$ and a subset $I \subseteq \mathbb{N}$, we define the projection, denoted by $(d_0, \ldots, d_n)[I]$, as the vector $(d_0, \ldots, d_n)$ for the largest $l \in \mathbb{N}$ such that $i_0 < i_1 < \ldots < i_l \leq n$ and $i_k \in I$ for $0 \leq k \leq l$. For a vector $(d_0 : D_0, \ldots, d_n : D_n)$ we write $d : D$ and denote the projection for a subset of indices $I \subseteq \mathbb{N}$ by $d_I : D_I$. Finally, we define $\text{Vars}(d) = \{d_0, \ldots, d_n\}$.

A multi-set over a set $A$ is a total function $m : A \rightarrow \mathbb{N}$; we refer to $m(a)$ as the multiplicity of $a$ and we write $\{\ldots\}$ for a multi-set where the multiplicity of each element is either written next to it or omitted when it is one. For instance, $\{a : 2, b : 3\}$ has elements $a$ and $b$ with multiplicity two and one respectively, and all other elements have multiplicity zero. For multi-sets $m, m' : A \rightarrow \mathbb{N}$, we write $m \subseteq m'$ iff $m(a) \leq m'(a)$ for all $a \in A$. Multi-sets $m + m'$ and $m - m'$ are defined pointwise: $(m + m')(a) = m(a) + m'(a)$ and $(m - m')(a) = \max(m(a) - m'(a), 0)$ for all $a \in A$.

2.1. Labelled transition systems

Let $\Lambda$ be the set of (sorted) action labels. We use $\text{D}_a$ to indicate the sort of action label $a \in \Lambda$. The set of all multi-sets over $\{a(e) \mid a \in \Lambda, e \in \text{D}_a\}$ is denoted $\Omega$. Note that $\text{D}_a$ is the semantic domain of $\text{D}_a$. In examples we often omit the expression and parentheses whenever $\text{D}_a$ consists of a single element.

**Definition 2.1.** A labelled transition system with multi-actions, abbreviated LTS, is a tuple $\mathcal{L} = (S, \text{Act}, \rightarrow)$ where $S$ is a set of states; $\text{Act} \subseteq \Omega$ and $\rightarrow \subseteq S \times \text{Act} \times S$ is a labelled transition relation.

We typically use $\omega$ to denote an element of $\text{Act}$ and we write $s \xrightarrow{\omega} t$ whenever $(s, \omega, t) \in \rightarrow$. As usual, a finite LTS can be depicted as an edge-labelled directed graph, where vertices represent states, the labelled edges represent the transitions.
The left graph of Fig. 2 (see page 2) depicts an LTS with four states and five transitions, which are labelled with multi-actions \( \{\text{count}, \text{tag}\} \) and \( \{\text{sync}_1^\text{true}, \text{false}\} \). The size of a labelled transition system is given by the number of states and transitions combined.

We recall the well-known strong bisimulation equivalence relation on states of an LTS [16].

**Definition 2.2.** Let \( \mathcal{L} = (S, \text{Act}, \rightarrow) \) be an LTS. A binary relation \( R \subseteq S \times S \) is a (strong) bisimulation relation iff for all \( s, t \):

- if \( s \overset{a}{\rightarrow} s' \) then there is a state \( t' \in S \) such that \( t \overset{a}{\rightarrow} t' \) and \( s' R t' \), and
- if \( t \overset{a}{\rightarrow} t' \) then there is a state \( s' \in S \) such that \( s \overset{a}{\rightarrow} s' \) and \( s' R t' \).

States \( s \) and \( t \) are *bisimilar*, denoted \( s \overset{}{=} t \), iff \( s R t \) for a bisimulation relation \( R \).

2.2. Linear process equations

We draw inspiration from the process algebra mCRL2 [1] to describe the elements of an LTS; similar language concepts and constructs may appear in other shapes elsewhere; for example ACP [17], CCS [18] and SCCS [16].

**Definition 2.3.** Multi-actions are defined as follows:

\[
\alpha ::= \tau \mid a(e) \mid \alpha|\alpha
\]

Constant \( \tau \) represents the empty multi-action and \( a \in \Lambda \) is an action label with an expression \( e \) of sort \( D_a \). The semantics of a multi-action \( \alpha \), given a substitution \( \sigma \), is denoted by \( \llbracket \alpha \rrbracket_\sigma \) and is an element of \( \Omega \). It is defined inductively as follows:

\[
\llbracket \tau \rrbracket_\sigma = \{ \}, \quad \llbracket a(e) \rrbracket_\sigma = \llbracket a(\sigma(e)) \rrbracket_\sigma \quad \text{and} \quad \llbracket \alpha|\beta \rrbracket_\sigma = \llbracket \alpha \rrbracket_\sigma + \llbracket \beta \rrbracket_\sigma .
\]

If \( \alpha \) is a closed expression then the substitution is typically omitted.

**Example 2.4.** Consider the multi-action \( \text{toggle}\{\text{sync}_1^\text{false}\} \). Since this is a closed expression, the semantics \( \llbracket \text{toggle}\{\text{sync}_1^\text{false}\} \rrbracket \) of the multi-action is independent of a substitution. The semantics of the multi-action \( \llbracket \text{toggle}\{\text{sync}_1^\text{false}\}(x) \rrbracket \), in the context of substitution \( \sigma \) satisfying \( \sigma(x) = \text{true} \) is \( \llbracket \text{toggle}\{\text{sync}_1^\text{true}\} \rrbracket \).

The states and transitions of an LTS are described by means of monolithic processes called **linear process equations**, which consist of a number of condition-action-effect statements, referred to as *summands*. Each summand symbolically represents a partial transition relation between the current and the next state for a multi-set of action labels. Let \( PN \) be a set of (sorted) process names.

**Definition 2.5.** A **linear process equation** (LPE) is an equation of the form:

\[
P(d : D) = \sum_{e_0 : E_0} c_0 \rightarrow \alpha_0 \cdot P(g_0) + \ldots + \sum_{e_n : E_n} c_n \rightarrow \alpha_n \cdot P(g_n)
\]

Where \( P \in PN \) is the process name, \( d \) is the process parameter, and each:

- \( E_i \) is a sort ranged over by sum variable \( e_i \) (where \( e_i \neq d \)),
- \( c_i \) is the enabling condition, a boolean expression so that \( \text{FV}(c_i) \subseteq [d, e_i] \),
- \( \alpha_i \) is a multi-action \( \tau \) or \( a_1^i(f_1^i) \ldots a_n^i(f_n^i) \) such that each \( a_j^i \in \Lambda \) and \( f_j^i \) is an expression of sort \( D_{a_j^i} \) such that \( \text{FV}(f_j^i) \subseteq [d, e_i] \),
- \( g_i \) is an update expression of sort \( D \), satisfying \( \text{FV}(g_i) \subseteq [d, e_i] \).

The \( + \)-operator denotes a non-deterministic choice among the summands of a given LPE; the \( \sum \)-operator describes a non-deterministic choice among the possible values of the associated sum variable bound by the \( \sum \)-operator. We omit the \( \sum \)-operator when the sum variable does not occur freely within the condition, action and update expressions. We use \( +_{i \in I} \) for a finite set of indices \( I \subseteq \mathbb{N} \) as a shorthand for a number of summands.

Note that similar structures also occur elsewhere. For example in Extended Finite State Machines [19] or Symbolic Transition Graphs [20], process parameters are sometimes called state variables and summands are referred to as transitions. We often consider LPEs where the parameter sort \( D \) represents a vector; in that case we write \( d_0 : D_0, \ldots, d_n : D_n \) to indicate that there are \( n + 1 \) parameters where parameter \( d_i \) has sort \( D_i \). Similarly, we also generalise the action sorts and the sum operator in LPEs, where we permit ourselves to write \( a(f_0, \ldots, f_k) \) and \( \sum_{e_0 : E_0, \ldots, e_i : E_i} \) respectively.

Let \( P \) be the set of expressions \( P(i), \) where \( P \in PN \) and \( i \) is a closed expression of sort \( D \) (the sort of \( P \)). The labelled transition system induced by a (set of) LPE(s) is then formally defined as follows.
**Definition 2.6.** The operational semantics associated with expressions of \( P \) is the LTS \((P, \Omega, \rightarrow)\), where the transition relation \( \rightarrow \) is defined as the smallest relation obtained as follows: for each LPE \( P(d : D) = \bigvee_{i \in I} \sum_{c_i \in C_i} c_i \rightarrow \alpha_i \cdot P(g_i) \) and for all indices \( i \in I \), closed expressions \( \iota : D \) and substitutions \( \sigma \) such that \( \sigma(d) = \iota \) there is a transition \( P(\iota) \xrightarrow{[\sigma]} P(\sigma(g)) \) iff \([\sigma(c)] = \text{true}\).

For a given expression \( P(\iota) \), we refer to the part of the LTS that is reachable from \( P(\iota) \) as the state space. Note that in defining the transition system in Definition 2.6, in the interpretation of an LPE a syntactic substitution is applied to the update expressions to define the reached state. This means that different closed syntactic expressions that correspond to the same semantic object, e.g., \( 1 + 1 \) and \( 2 \) for our assumed sort \( \text{Nat} \), result in different states. As stated by the lemma below, such states are always bisimilar.

**Lemma 2.7.** For all closed expressions \( e, e' : D \) such that \([e \approx e'] = \text{true}\) we have \( P(e) = P(e') \).

For any given state space we can therefore consider a representative state space where for each state a unique closed expression is chosen that is data equivalent. In examples we always consider the representative state space.

**Example 2.8.** Consider the following LPE, modelling a machine that alternates between two modes. The event that signals a switch between these two modes is modelled by action \( \text{toggle} \); switching between modes happens after a number of clock cycles and is dependent on the mode in which the machine is running (3 cycles for one mode, 1 cycle for the other). The machine keeps track of its mode using a Boolean parameter \( s \), and parameter \( n \) keeps track of the number of cycles left before switching modes.

\[
\text{Machine}(n : \text{Nat}, s : \text{Bool}) = (n > 0) \rightarrow \text{count} \cdot \text{Machine}(n - 1, s) \\
+ (n \approx 0) \rightarrow \text{toggle} \cdot \text{Machine}(\text{if}(\neg s, 3, 1), \neg s)
\]

Note that the expression \( \text{if}(\neg s, 3, 1) \) models the reset of the clock cycle count upon switching modes. A representative state space of \( \text{Machine}(0, \text{false}) \), where the machine is initially off, is shown in Fig. 1 (see page 2).

2.3. A process algebra of communicating linear process equations

We define a minimal language to express parallelism and interaction of LPEs; the operators are taken from mCRL2 [1] and similar-styled process algebras. Let \( \text{Comm} \) be the set of communication expressions of the form \( a_0 \ldots a_n \rightarrow c \) where \( a_0, \ldots, a_n, c \in \Lambda \ (n > 0) \) are action labels.

**Definition 2.9.** The process algebra is defined as follows:

\[ S ::= \nabla_A(S) \mid \tau_H(S) \mid \Gamma_C(S) \mid S \parallel S \mid P(\iota) \]

Here, \( A \subseteq 2^\Lambda \times \mathbb{N} \) is a non-empty finite set of finite multi-sets of action labels, \( H \subseteq \Lambda \) is a non-empty finite set of action labels and \( C \subseteq \text{Comm} \) is a finite set of communication expressions. Finally, we have \( P(\iota) \in \text{P} \).

The set \( S \) contains all expressions of the process algebra. Intuitively, the action allowing operator \( \nabla_A(S) \) blocks exactly those multi-actions of \( S \) whose multi-set of action labels are not in \( A \). The action hiding operator \( \tau_H(S) \) effectively replaces each multi-action in \( S \) by a new one, obtained by renaming all actions \( a(d) \) of the multi-action to \( \tau \) when \( a \in H \). Parallel composition \( S \parallel S' \) is used to specify that processes \( S \) and \( S' \) execute their (multi-)actions in parallel. The communication operator \( \Gamma_C(S) \) describes that actions that happen in parallel can synchronise. Intuitively, a communication expression \( a_0 \ldots a_n \rightarrow c \in C \) specifies that when actions \( a_0(d_0), \ldots, a_n(d_n) \) run in parallel in \( S \), they can synchronise to action \( c(d) \), provided that \( d, d_0, \ldots, d_n \) specify the same values.

Not all sets of communication expressions \( C \) are sensible. First, we require that the left-hand sides of the communications do not share labels. Second, the action label on the right-hand side must not occur in any other left-hand side. More formally, we henceforward impose the following (syntactic) restrictions on the set of communication expressions.

**Definition 2.10.** A set \( C \subseteq \text{Comm} \) is valid iff for all pairs of distinct communication expressions \( a_0 \ldots a_n \rightarrow c \in C \) and \( b_0 \ldots b_m \rightarrow d \in C \):

- \([a_0, \ldots, a_n] \cap [b_0, \ldots, b_m] = \emptyset \), and
- \( d \notin [a_0, \ldots, a_n] \), and, symmetrically, \( c \notin [b_0, \ldots, b_m] \).
Observe that by these rules, neither \([a|b \rightarrow c, a|d \rightarrow c]\) nor \([a|b \rightarrow c, c \rightarrow d]\) are valid sets of communication expressions. We remark that communication operators can be stacked; for instance, one can write \(\Gamma_{|c \rightarrow d}(\Gamma_{|a|b \rightarrow c}(S))\) to ensure that multi-actions involving both action labels \(a\) and \(b\), are transformed into multi-actions involving \(c\) instead (provided, of course, the parameters of \(a\) and \(b\) match); in the resulting multi-actions further synchronisation between the \(c\) and \(e\) may then take place.

The operational semantics of expressions in \(S\) are defined in Definition 2.15. We introduce three auxiliary functions on \(\Omega\) that are used to define the operational semantics of the communication operator, the hiding operator and the action allowing operator. First of all, the auxiliary function \(\gamma_C\) defined below specifies the result of applying a set \(C\) of communication expressions to a multi-action \(\omega \in \Omega\). Since multiple communication expressions may be applicable at the same time, definition \(\gamma\) is recursive in nature. The validity requirements on the set of communication expressions guarantee that it is a well-defined function.

**Definition 2.11.** Given \(\omega \in \Omega\) we define \(\gamma_C\), where \(C \subseteq \text{Comm}\) is valid, as follows:

\[
\begin{align*}
\gamma_H(\omega) &= \omega \\
\gamma_C(\omega) &= \gamma_C(C_1)(\gamma_C(\omega)) \quad \text{for } C_1 \subseteq C \\
\gamma(\omega|a|b|\rightarrow c)(\omega) &= \begin{cases} 
\gamma(\omega|a|b|\rightarrow c)(\omega) = \gamma(\omega|a|b|\rightarrow c)(\omega|a|b|\rightarrow c) & \text{if } \gamma(\omega|a|b|\rightarrow c)(\omega|a|b|\rightarrow c) \subseteq \omega \\
\omega & \text{ otherwise}
\end{cases}
\end{align*}
\]

**Example 2.12.** We have \(\gamma(\omega|\rightarrow c)(\gamma(\omega|a|b|\rightarrow c)(\omega)) = \gamma(\omega|a|b|\rightarrow c)(\omega|a|b|\rightarrow c)\). On the other hand, \(\gamma(\omega|\rightarrow c)(\gamma(\omega|a|b|\rightarrow c)(\omega|a|b|\rightarrow c)) = \gamma(\omega|a|b|\rightarrow c)(\omega|a|b|\rightarrow c)\) since \(a\) and \(b\)’s parameters do not match. Furthermore, \(\gamma(\omega|\rightarrow c, d|\rightarrow e)(\gamma(\omega|a|b|\rightarrow c, d|\rightarrow e)(\omega))\) results in \(\gamma(\omega|a|b|\rightarrow c, d|\rightarrow e)(\omega)\)

Next, we define an auxiliary function \(\theta_H(\omega)\) that defines the transformation that takes place when using the hiding operator \(\tau_H(S)\). This function yields a multi-action in which all the action labels that occur in \(H\) are removed. Note that the multi-action becomes \(\tau\) when all action labels are hidden.

**Definition 2.13.** Let \(\omega \in \Omega\) and \(H \subseteq \Lambda\). We define \(\theta_H(\omega)\) as the multi-action \(\omega' \in \Omega\) defined as:

\[
\omega'(\omega(d)) = \begin{cases} 
0 & \text{if } a \in H \\
\omega(\omega(d)) & \text{otherwise}
\end{cases}
\]

**Example 2.14.** Consider the multi-action \(\gamma(\omega|a|b|\rightarrow c, d|\rightarrow e)(\omega)|a|b|\rightarrow c, d|\rightarrow e)(\omega)\) that resulted from applying function \(\gamma\) in the previous example. The action \(c\) that is obtained by synchronisation can be hidden using the auxiliary function \(\theta\): we have \(\theta(\gamma(\omega|a|b|\rightarrow c, d|\rightarrow e)(\omega)|a|b|\rightarrow c, d|\rightarrow e)(\omega)) = \gamma(\omega|a|b|\rightarrow c, d|\rightarrow e)(\omega))\). Similarly, the multi-action \(\gamma(\omega|a|b|\rightarrow c, e|\rightarrow f)(\omega)|a|b|\rightarrow c, e|\rightarrow f)(\omega)\), which was the result from applying two communication expressions to a multi-action, can be hidden by hiding the respective actions \(c\) and \(e\): \(\theta(\gamma(\omega|a|b|\rightarrow c, e|\rightarrow f)(\omega)) = \gamma(\omega|a|b|\rightarrow c, e|\rightarrow f)(\omega))\).

Finally, given a multi-action \(\alpha\) we write \(\alpha\) to denote the multi-set of action labels that occur in \(\alpha\), e.g., \(a(3)|a(2)|b(5) = \{a : 2, b\}\). Formally, \(a(e) = \{a\}\), \(\tau = \{\}\) and \(a|b = a + b\). We overload this notation to also apply to elements \(\omega \in \Omega\), allowing us to reason about \(\omega\) in a similar way. For instance, \(\gamma(\omega|a(3), a(2), b(5)) = \gamma(\omega|a(3), a(2), b(5))\).

**Definition 2.15.** The LTS \((\mathcal{S}, \Omega, \rightarrow)\) associated with expressions of \(S\) is defined by the rules below and the transition relation given in Definition 2.6 for each expression in \(P\). For any \(\omega, \omega' \in \Omega\), expressions \(P, P', Q, Q'\) of \(S\) and sets \(C \subseteq \text{Comm}, A \subseteq \mathbb{2}^\Lambda \rightarrow \mathbb{N}\) and \(H \subseteq \Lambda\):
Note that for \textsf{Allow} the condition $\omega \in A \cup \{\text{\bare}\}$ must be satisfied in order for the rule to be applicable.

Observe that \textsf{PARL} and \textsf{PARR} are distinct rules from \textsf{Par}. Rule \textsf{Par} expresses that a transition from both $P$ and $Q$ happen simultaneously, whereas rules \textsf{PARR} and \textsf{PARL} cover the case in which only one of the two processes involved in the parallel composition executes a transition. Furthermore, observe that it is not possible to disallow the occurrence of a $t$-transition, because the condition $\tau \in A \cup \{\text{\bare}\}$ rule \textsf{Allow} is true for any $A$ since $\tau = \text{\bare}$.

**Example 2.16.** Consider the following LPE that models a drill component in which each toggle action leads to a drill action.

$$\text{Drill}(t : \text{Bool}) = (\neg t) \rightarrow \text{toggle} \cdot \text{Drill(true)}$$

$$+ (t) \rightarrow \text{drill} \cdot \text{Drill(false)}$$

Suppose that we wish to study the interaction of LPEs Machine of Example 2.8 and Drill, assuming that their toggle actions must synchronise, resulting in a toggle action. Let $C = \{\text{toggle} \rightarrow \text{toggle}'\}$ be the communication that specifies this synchronisation, and let $A = \{\text{toggle}', \text{\bare}, \text{drill}, \text{\count}\}$ be the set of multi-action labels we allow. The interaction between LPEs Machine and Drill can be specified by the expression $\forall A(\Gamma_C(\text{Machine}(0, \text{false}) \parallel \text{Drill(false)})))$ in the algebra. An example derivation is depicted below, invoking (from top to bottom) rule \textsf{Par} and finally \textsf{Allow}:

\[
\begin{align*}
\text{Machine}(0, \text{false}) & \xrightarrow{\text{toggle}} \text{Machine}(3, \text{true}) & \text{Drill}(\text{false}) & \xrightarrow{\text{toggle}} \text{Drill}(\text{true}) \\
\text{Machine}(0, \text{false}) \parallel \text{Drill}(\text{false}) & \xrightarrow{\text{toggle:2}} \text{Machine}(3, \text{true}) \parallel \text{Drill}(\text{true}) \\
\forall A(\Gamma_C(\text{Machine}(0, \text{false}) \parallel \text{Drill}(\text{false}))) & \xrightarrow{\text{toggle:1}} \forall A(\Gamma_C(\text{Machine}(3, \text{true}) \parallel \text{Drill}(\text{true})))
\end{align*}
\]

Observe that in the last step in the derivation, rule \textsf{Allow} is applicable since $\text{\bare} \in A \cup \{\text{\bare}\}$. Note that we cannot derive $\forall A(\Gamma_C(\text{Machine}(0, \text{false}) \parallel \text{Drill}(\text{false})))$, even though by, e.g., \textsf{PARL}, we can derive $\text{Machine}(0, \text{false}) \parallel \text{Drill}(\text{false}) \xrightarrow{\text{toggle}} \text{Machine}(3, \text{true}) \parallel \text{Drill}(\text{false})$. The reason for this is that $\text{\bare} \notin A \cup \{\text{\bare}\}$. \hfill \Box

We wrap up this section by remarking that the operators we introduced in this section are fairly standard in ACP-style process algebras. Some process algebras, such as CCS, have a parallel composition operator with an implicit communication function that hides the resulting synchronisations. Such a parallel composition can be described by nesting a parallel composition, communication operator, allow operator and hiding operator. By nesting multiple communication operators, more intricate synchronisation schemes such as broadcast communication between a finite set of processes can be described. For example $\Gamma_Aa \rightarrow b(\Gamma_Aa \rightarrow b((P \parallel Q) \parallel R))$ describes that multi-actions of $(P \parallel Q) \parallel R$ involving three $a$-action labels, but also those containing only two $a$-action labels, synchronise and yield a single $b$-action label instead.

Finally, we remark that communication and synchronisation schemes also exist in, e.g., networks of LTSs [21], Exp.Open [22] of CADP [23] and pNets [24]. At first glance, such schemes are more liberal and expressive than the communication operator we have introduced. However, by employing renaming (using communication expressions of the shape $a \rightarrow b$) and by nesting communication and allowing operators, in practice one is able to express most synchronisation schemes in a rather straightforward way. The question whether all synchronisation schemes can be expressed in our language (and vice versa) is interesting but beyond the scope of the current work. Regardless of the potentially different expressive power of the various formalisms, the techniques we outline in this paper should be applicable to any language containing multi-actions and a synchronisation scheme that allows for expressing at least some form of CCS-style communication on multi-actions.

### 3. The decomposition problem

The state space of a monolithical LPE may grow quite large and generating that state space may either take too long or require too much memory. We are therefore interested in decomposing an LPE into two or more LPEs, where the latter are referred to as components, such that the state spaces of the resulting components are smaller than that of the original state space. Such a decomposition is considered valid iff the original state space is strongly bisimilar to the state space of these components when combined under a suitable context (i.e., an expression with a ‘hole’) that specifies how to combine the components. We formalise this problem as follows.

**Definition 3.1.** Let $P(\overline{d} : \overline{D}) = \phi$ be an LPE and $\overline{t} : \overline{D}$ a closed expression. The LPEs $P_0(\overline{d}_{I_0} : \overline{D}_{I_0}) = \phi_0$ to $P_N(\overline{d}_{I_N} : \overline{D}_{I_N}) = \phi_N$, for sets of indices $I_0, \ldots, I_N \subseteq \mathbb{N}$, are a valid decomposition of $P$ and $\overline{t}$ iff there is a context $\Gamma$ such that:

$$P(\overline{t}) = \Gamma[P_0(\overline{d}_{I_0}) \parallel \ldots \parallel P_N(\overline{d}_{I_N})]$$
where $C[P_0(\hat{\iota}_{I_0}) \parallel \ldots \parallel P_n(\hat{\iota}_{I_n})]$ is an expression in $S$. We refer to the expression $C[P_0(\hat{\iota}_{I_0}) \parallel \ldots \parallel P_n(\hat{\iota}_{I_n})]$ as the composition.

In the next sections, we will show that a suitable context $C$ can be constructed using the operators from $S$, and we define a decomposition technique that results in exactly two components (a cleave). The technique can, in principle, be applied recursively to these two components. The primary benefit of a valid decomposition is that a state space that is equivalent to the original state space can be obtained using compositional minimisation, which can result in a state space that is immediately be significantly smaller than the state space resulting from monolithic exploration. First, the state space of each component is derived separately. The composition can then be derived from the component state spaces, exploiting the rules of the operational semantics. The component state spaces can be minimised modulo bisimilarity, which is a congruence with respect to the operators of $S$ before deriving the results of the composition expression.

4. A solution to the decomposition problem

A basic observation that we exploit in our solution to the decomposition problem is that when hiding label $c$ in a multi-action $\alpha|c$, we are left with multi-action $\alpha$, provided that $c$ does not occur in $\alpha$. When the multi-action $\alpha$ is an event that is possible in a monolithic LPE and the label $c$ is the result of a communication between two components, we can effectively exchange information between multiple components, without this information becoming visible externally. The example below illustrates the idea using a naive but valid solution to the decomposition technique on the LPE of Example 2.8.

Example 4.1. Reconsider the LPE Machine we defined earlier, and consider the two components depicted below.

$$ Machine_V(n : Nat) = \sum_{s : \text{Bool}} (n > 0) \rightarrow \text{count}_{\hat{s}}^0(n, s). Machine_V(n - 1) $$

$$ + \sum_{s : \text{Bool}} (n \approx 0) \rightarrow \text{sync}_V(n, s). Machine_V(\text{if}(\neg s, 3, 1)) $$

$$ Machine_W(s : \text{Bool}) = \sum_{n : \text{Nat}} (n > 0) \rightarrow \text{sync}_W^0(n, s). Machine_W(s) $$

$$ + \sum_{n : \text{Nat}} (n \approx 0) \rightarrow \text{toggle}_s\text{sync}_W^1(n, s). Machine_W(\neg s) $$

Each component describes part of the behaviour and knows the value of parameter $n$ or $s$, but not the other. To cater for this, it is ‘over-approximated’ by a sum variable. The state space of Machine$_V(0)$ is shown below. The synchronisation actions sync expose the non-deterministically chosen values of the unknown parameters.

![Diagram of Machine V and Machine W](attachment:image.png)

Enforcing synchronisation of the sync actions, the context $C$ can be chosen as follows to achieve a valid decomposition:

$$ \forall \{\text{toggle}, \text{count}\} (\tau_{\text{sync}^0, \text{sync}^1} (\Gamma_{\text{sync}^0 \text{sync}^0} \rightarrow \text{sync}^0, \text{sync}^1 | \text{sync}^1 \rightarrow \text{sync}^1) (Machine_V(0) \parallel Machine_W(\text{false})))) $$

Unfortunately the state space of Machine$_W(\text{false})$ in the above example is infinitely branching and it has no finite state space that is strongly bisimilar to it, rendering the decomposition useless in practice. We will subsequently develop a more robust solution.

4.1. Separation tuples

To obtain a useful decomposition it can be beneficial to reduce the number of parameters that occur in the synchronisation actions, because these then become a visible part of the transitions in the state spaces of the individual components. In the worst case, as illustrated by LPE Machine$_W$ of Example 4.1, synchronisation actions lead to a component having an infinite state space despite the fact that the state space of the original LPE is finite.

One observation we exploit is that in some cases we can actually remove the synchronisation for summands completely. For instance, in the first summand of Machine in Example 2.8, the value of parameter $s$ remains unchanged and the
condition is only an expression containing parameter n. We refer to summands with such a property as independent summands, whereas the other summands are dependent. When defining the context C, we can allow a component to execute multi-actions of its independent summands without enforcing a synchronisation with the other component. This allows, for instance, component Machine_V to independently execute (multi-)action count without synchronising the values of s and n with Machine_W.

We must ensure that each dependent summand of the monolithic LPE is covered by both components that we extract from the LPE. However, an independent summand of one component does not need a corresponding summand in the other component. Therefore, the summands that we extract for a given component are identified by a set of indices J of the summands of the monolithic LPE. Of these, we furthermore can identify summands that are dependent and summands that are independent. The indices for the latter are collected in a set K.

A third observation that can be utilised is that for the dependent summands, there is some degree of flexibility for deciding which component will contribute to which part of the summand of the monolithic LPE. More specifically, by carefully distributing the enabling condition c and action expression α of a summand of the monolithic LPE over the two components, the amount of information that needs to be exchanged between these two components when they execute their respective summands, can be minimised. That is, the synchronisation of ‘missing’ parameters, i.e., the parameters of the other component, might be avoided when the condition and action expressions of one component no longer contain that parameter.

The final observation is that the synchronisation actions of dependent summands can be used to synchronise the result of arbitrary expressions instead of only process parameters. This can be used when distributing equality conditions over the two components, i.e., conditions of the form \( e = e' \), where one side of the equality (e.g., expression e) can be put into the synchronisation action of one component and the other side of the equality (expression e') into the other component. We use the term synchronisation expression to refer to the expression passed along an argument to the synchronisation action.

Note that the way we distribute the list of process parameters of the monolithic LPE over the two components may affect which summands can be considered independent. For instance, had we decided to assign the (multi-)action count to Machine_W and toggle to Machine_V, we would not be able to declare count’s summand independent. Consequently, the set of process parameter indices U, assigned to a component, and the set K are mutually dependent.

To capture this relation, we introduce the concept of a separation tuple. The concept of a separation tuple, a 6-tuple which we introduce below, formalises the required relation between the sets of indices for independent summands K, summands J and process parameters U, and the conditions c, and action α and a vector of data expressions \( h \) that we call synchronisation expressions of a component. We use indexed sets to define the condition, action and update expressions for every dependent summand. For elements in an indexed set we use subscript notation to indicate the index of that element.

**Definition 4.2.** Let \( P(\vec{d} : \vec{D}) = \sum_{i \in I} \sum_{e_i \in E_i} c_i \rightarrow \alpha_i \cdot P(\vec{g}_i) \) be an LPE. A separation tuple for P is a 6-tuple \((U, K, J, c^U, \alpha^U, h^U)\) where \( U \subseteq \mathbb{N} \) is a set of parameter indices, \( K \subseteq J \subseteq I \) are two sets of summand indices, and \( c^U, \alpha^U \) and \( h^U \) are sets of condition, action and synchronisation expressions respectively, indexed by indices from \( J \setminus K \). We require that for all \( i \in (J \setminus K) \) it holds that \( \text{FV}(c_i^U) \cup \text{FV}(\alpha_i^U) \cup \text{FV}(h_i^U) \subseteq \text{Vars}(\vec{d}) \cup \{e_i\} \), and for all \( i \in K \) it holds that \( \text{FV}(c_i) \cup \text{FV}(\alpha_i) \cup \text{FV}(\vec{g}_i) \subseteq \text{Vars}(\vec{d}) \cup \{e_i\} \).

A separation tuple induces an LPE, where \( U' = \mathbb{N} \setminus U \), as follows:

\[
P_U(\vec{d}_U : \vec{D}_{U}) = \sum_{i \in (I \setminus K)} \sum_{e_i \in E_i} c_i \rightarrow \alpha_i^U \cdot \text{sync}_i^U(h_i^U) \cdot P_U(\vec{g}_i) + \sum_{i \in K} c_i \rightarrow \alpha_i^U \cdot \text{tag} \cdot P_U(\vec{g}_i)
\]

We assume that action label \( \text{sync}_i^U \), for any \( i \in I \), and label tag does not occur in \( \alpha_j \), for any \( j \in I \), to ensure that these action labels are fresh.

Observe that for independent summands the action label is extended with a tag action in Definition 4.2. This label is (only) needed to properly deal with overlapping multi-actions, as we illustrate below in Example 4.3.

**Example 4.3.** Consider the following LPE.

\[
P(\text{Bool} : \text{Bool}) = x \rightarrow a \cdot P(\text{false}, y) + y \rightarrow b \cdot P(\text{x}, \text{false}) + (x \land \neg y) \rightarrow a | b \cdot P(\text{false}, \text{false})
\]

Suppose we decompose LPE \( P \) with indices \( I = \{0, 1, 2\} \) using the tuples \((V, \{0\}, \{0, 2\}, \{x_2\}, \{a_2\}, \{\langle \rangle 2\})\) and \((W, \{1\}, \{1, 2\}, \{\langle \neg y \rangle 2\}, \{b_2\}, \{\langle \rangle 2\})\), where \( V = \{0\} \) and \( W = \{1\} \), and assuming that the summands of \( P \) are indexed from top to bottom.
by 0, 1 and 2 respectively. Now assume that we had omitted the tag action in Definition 4.2, in which case these separation tuples would induce the following LPEs:

\[ P_V(x : \text{Bool}) = x \rightarrow a . P_V(\text{false}) \]
\[ + x \rightarrow a (\text{sync}_V^2 . P_V(\text{false}) \]
\[ P_W(y : \text{Bool}) = y \rightarrow b . P_W(\text{false}) \]
\[ + (\neg y) \rightarrow b (\text{sync}_W^2 . P_W(\text{false}) \]

Observe that both \( P_V(\text{true}) \) and \( P_W(\text{true}) \) are transitions for these components. This also means that due to (among others) rule \( \text{PAR} \), \( P_V(\text{true}) \) makes it possible to perform actions \( a, b \). Note that \( P(\text{true}, \text{true}) \) does not have an outgoing transition labelled \( a, b \), but (the reachable) process \( P(\text{true}, \text{false}) \) does have an outgoing \( a, b \) transition. There is, however, no composition expression that prevents \( a, b \) in \( P_V(\text{true}) \) and \( P_W(\text{true}) \) and allows \( a, b \) in \( P_V(\text{true}) \) or \( P_W(\text{false}) \). The tag label provides the tools for making this distinction by only allowing at most one tag action to be present, and therefore disallowing \( a, b, \text{tag}, \text{tag} \).

The components, induced by two separation tuples, can be (re)combined in a context that ensures synchronisation of the sync events and which hides their communication trace. This ensures that all actions left can be traced back to the monolithic LPE from which the components are derived. Under specific conditions, this is achieved by the following context.

**Definition 4.4.** Let \( P(\hat{d} : \hat{D}) = \sum_{i \in I} c_i \rightarrow a_i . P(\hat{g}_i) \) be an LPE with the separation tuples \( (V, K^V, J^V, c^V, a^V, h^V) \) and \( (W, K^W, J^W, c^W, a^W, h^W) \) for \( P \). Let \( P_V(\hat{d}_{\mid V} : \hat{D}_V) = \phi_V \) and \( P_W(\hat{d}_{\mid W} : \hat{D}_W) = \phi_W \) be the induced LPEs according to Definition 4.2. Let \( \tau : \hat{D} \) be a closed expression. Then the composition expression is defined as:

\[ \tau(\phi_V) (\phi_W(\tau) | \phi_V(\tau) | \phi_W(\tau) | \phi_W(\tau) | \phi_W(\tau)) \]

Before we proceed to identify the conditions under which two separation tuples induce a valid decomposition using the above context, we revisit Example 2.8 to illustrate the concepts introduced so far.

**Example 4.5.** Reconsider the LPE presented in Example 2.8 with \( V = \{0\} \) and \( W = \{1\} \). The separation tuple \( (V, \{0\}, \{1\}, \{n \approx 0\}, \{\tau_1\}, \{\tau_2\}) \) and the tuple \( (W, \emptyset, \{1\}, \{\text{true}_1\}, \{\text{false}_1\}, \{\text{false}_2\}) \) for Machine induce component \( \text{Machine}_V \) and \( \text{Machine}_W \) respectively.

\[ \text{Machine}_V(n : \text{Nat}) = \begin{cases} \text{count} & | \text{tag} . \text{Machine}_V(n - 1) \\
\sum_{s : \text{bool}} (n \approx 0) & \rightarrow \tau | \text{sync}_V^1(s) . \text{Machine}_V(\text{if}(\neg s, 3, 1)) \end{cases} \]

\[ \text{Machine}_W(s : \text{Bool}) = \text{true} \rightarrow \text{toggle} | \text{sync}_W^1(s) . \text{Machine}_W(\neg s) \]

Note that we omitted the \( \sum \)-operator in the first summand of \( \text{Machine}_V \) since sum variable \( s \) does not occur as a free variable in the expressions; for similar reasons, the \( \sum \)-operator is omitted in \( \text{Machine}_W \). The state spaces of components \( \text{Machine}_V(0) \) and \( \text{Machine}_W(\text{false}) \) are shown in Fig. 2. We obtain the following composition according to Definition 4.4:

\[ \tau(\text{tag}) (\tau(\text{count}) \cup \text{toggle} \cup \text{count} . \text{tag}) (\tau(\text{sync}_V^0, \text{sync}_V^1) \cup \tau(\text{sync}_W^0, \text{sync}_W^1) (\text{Machine}_V(0) \cup \text{Machine}_W(\text{false})))) \]

Observe that for the middle action, we compute the set \( \text{count} \cup \text{toggle} \cup \text{count} . \text{tag} \), which is equal to \( \text{count} \cup \text{toggle} \cup \text{count} . \text{tag} \). This composition expression is strongly bisimilar to \( \text{Machine}_V(0, \text{false}) \) shown in Fig. 1. Note that the state space of \( \text{Machine}_V(0) \) has four states and transitions, and the state space of \( \text{Machine}_W(\text{false}) \) has two states and transitions, which are both smaller than the original state space. Their composition has the same size as the original state space and no further minimisation can be achieved (note that the state space of Fig. 1 is already minimal.

**4.2. Cleave correctness criteria**

It may be clear that not every decomposition which satisfies Definition 4.4 yields a valid decomposition (in the sense of Definition 3.1). For example, replacing the condition expression \( \text{true} \) in Example 4.5 of the summand in \( P_W \) by \( \text{false} \) would not result in a valid decomposition. Our aim in this section is to present the necessary and sufficient conditions to establish that the state space of the monolithic LPE is bisimilar to the state space of the composition expression resulting from Definition 4.4. We prove this in Theorem 4.12.
Consider a decomposition of an LPE $P$ according to Definition 4.4, induced by separation tuples $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$. We abbreviate the composition expression of Definition 4.4 by $\mathcal{C}[P_V(\vec{d}_V)||P_W(\vec{d}_W)]$. Recall that components $P_V$ and $P_W$ yield a valid decomposition of $P$ if there is a bisimulation relation between $P(\vec{d})$ and $\mathcal{C}[P_V(\vec{d}_V)||P_W(\vec{d}_W)]$. A bisimulation relation requires that related states can mimic each other’s steps. Since three LPEs are involved (the LPE and the two interacting components, induced by the separation tuples), we must consider situations that can emerge from any of these three LPEs executing a (multi-)action for states related by the bisimulation relation.

Two of the three relevant scenarios that must be considered are depicted in Fig. 3. Note that in all relevant scenarios, the initiative of the transition may be with either $P(\vec{d})$, or with the composition $\mathcal{C}[P_V(\vec{d}_V)||P_W(\vec{d}_W)]$.

Suppose that the monolithic LPE $P$ can take a step due to some summand $i \in I$, for which also $i \in K^V$. In that case—case (a) in Fig. 3—Definition 4.4 guarantees that the free variables of their condition, action and update expressions are taken from $d_i$; (multi-)action $a_i$ matches (multi-)action $a_i|\text{tag}$ after hiding tag. However, this is not sufficient to guarantee full independence of both components: what may happen is that the execution of a summand that is assumed to be independent still modifies the value of a process parameter of the other component, violating the idea of independence, and resulting in a target state in the composition that cannot be related to the target state of the monolithic LPE. In order to guarantee true independence, we must require that the $W$-projection on the update expression $\vec{g}_i$ of $P$ does not modify the corresponding parameters. Case (b) in Fig. 3 is dual. Formally, we require (IND): for all $r \in K^W$ we have $\vec{g}_{r|W} = \vec{d}_W$ and for all $r \in K^W$ we demand $\vec{g}_{r|W} = \vec{d}_W$. Note that in case $K^V$ and $K^W$ overlap, condition (IND) guarantees that the involved summands only induce self-loops since whenever $\vec{g}_{r|W} = \vec{d}_W$ and $\vec{g}_{r|W} = \vec{d}_W$ then $\vec{g}_i = \vec{d}$; thus no updates take place. Finally, observe that (IND) is also a sufficient condition for the monolithic LPE $P$ to match a (multi-)action $a_i$ in both cases (a) and (b) of Fig. 3, because the matching summand must occur in either $P_V$ or $P_W$.

The more complex scenario that must be considered is when $P_V$ and $P_W$ (must) synchronise to mimic the behaviour of $P$; see Fig. 4 due to the structure of $C$. Suppose again that the monolithic LPE $P$ can execute an $a_i$ action due to summand $i \in I$, but in this case, neither $i \in K^V$, nor $i \in K^W$. First, observe that the only option to match the behaviour of this summand is if a component covers at least all summands not already covered by the other component. We must therefore require at least the following condition (SYN): $J^V = I \setminus K^W$ and $J^W = I \setminus K^V$. It then follows from $K^V \subseteq J^V$ (and $K^W \subseteq J^W$) that $J^V \cap J^W$ are the summands that induce synchronisation and that these are disjoint from the independent summands; which are in $K^V \cup K^W$.

Second, observe that the enabledness of summand $i$ in $P$ depends on the enabling condition $c_i$. Consequently, if $c_i$ holds true, then the $i$-indexed conditions $c_i^V$ and $c_i^W$ must also hold true. Moreover, since we are dealing with dependent summands, the multi-action expression $a_i^V | a_i^W$ must reduce to $a_i$ under these conditions. Also the additional synchronisation vectors $\vec{h}^V$ and $\vec{h}^W$ must agree, for otherwise the sync actions of both components cannot participate in the synchronisation. Note that we do not need to explicitly require relating the update expressions of $P$ and the components $P_V$ and $P_W$ resulting from the execution of their $i$-indexed summands, since this property is already guaranteed by construction; see Definition 4.2. We collectively refer to the above requirements by condition (ORI).

Vice versa, whenever both components can simultaneously execute their $i$-indexed summand, we must ensure that also the monolithic LPE $P$ can execute its $i$-indexed summand. Condition (COM) ensures that this requirement is met. Note that $P_V$ and $P_W$ only synchronise on summands with equal indices due to the synchronisation on sync actions that is enforced. A technical complication in formalising requirement (COM), however, is that the sum variables of the individual components carry the same name in all three LPEs. In particular, from the fact that both individual components can successfully synchronise, we cannot deduce a unique value assigned to these homonymous sum-variables. We must therefore also ensure
that the update expressions of the components, resulting from executing the \( r \)-indexed summands, indeed is the same as could have resulted from executing the \( r \)-indexed summand in \( P \). Since this property is not guaranteed by the construction of Definition 4.2, there is a need to explicitly require it to hold.

A pair of separation tuples of \( P \) satisfying the above requirements is called a cleave of \( P \). Below, we formalise this notion, together with the requirements we informally introduced above. We defer a formal proof of correctness of these requirements to the end of this section.

**Definition 4.6.** Let \( P(\tilde{d} : \tilde{D}) = \sum_{\ell_i \in \ell} \sum_{\xi_i \in \xi} c_i \rightarrow \alpha_i \cdot P(\tilde{g}_i) \) be an LPE with the separation tuples \((V, K^V, J^V, c^V, \alpha^V, h^V)\) and \((W, K^W, J^W, c^W, \alpha^W, h^W)\) for \( P \) as defined in Definition 4.2. The two separation tuples are a cleave of \( P \) iff the following requirements hold.

**SYN.** \( J^V = I \setminus K^V \) and \( J^W = I \setminus K^V \).

**IND.** For all \( r \in K^V \), \( \tilde{g}_{r|W} = \tilde{d}_{|W} \), and for all \( r \in K^W \), \( \tilde{g}_{r|V} = \tilde{d}_{|V} \).

**ORI.** For all \( r \in (J^V \cap J^W) \) and substitutions \( \sigma \) satisfying \([\sigma(c_i)]\), also:

- \([\sigma(c_i)]\) and \([\sigma(c_i^W)]\), and
- \([\sigma(h_i)]\) = \([\sigma(h_i^W)]\), and
- \([\sigma(\alpha_i)]\) = \([\sigma(\alpha_i^W)]\).

**COM.** For all \( r \in (J^V \cap J^W) \) and substitutions \( \sigma \) and \( \sigma' \) satisfying \([\sigma(c_i^V)]\) and \([\sigma'(c_i^W)]\) and \([\sigma(h_i^V)]\) = \([\sigma'(h_i^W)]\), there is a substitution \( \rho \) such that \([\rho(d_{|V})] = [\sigma(d_{|V})]\) and \([\rho(d_{|W})] = [\sigma'(d_{|W})]\) and:

- \([\rho(\alpha_i)]\),
- \([\sigma(\alpha_i)]\) = \([\sigma'(\alpha_i)]\),
- \([\sigma(\tilde{g}_{r|V})]\) = \([\rho(\tilde{g}_{r|W})]\), and
- \([\rho(\tilde{g}_{r|W})]\) = \([\rho(\tilde{g}_{r|V})]\).

**Example 4.7.** We argue that the separation tuples inducing the decomposition obtained in Example 4.5 are a cleave indeed. First of all, the requirements SYN and IND can be checked quite easily. The requirements ORI and COM both have to be checked for the summand with index one. Consider the requirement ORI with a substitution \( \sigma \) assigning any value to \( n \) (and any value to other variables due to totality) such that \( [\sigma(n \approx 0)] \) holds. It follows directly that both \([\sigma(n \approx 0)]\) and \([\sigma(\text{true})]\) hold. Furthermore, \( (s) \) is the same synchronisation expression on both sides and \([\sigma(t \text{ toggle})]\) = \([\sigma(\text{toggle})]\) by definition. For the requirement COM consider any two substitutions \( \sigma \) and \( \sigma' \) such that both \([\sigma(n \approx 0)]\) and \([\sigma'(n \approx 0)]\) hold and \([\sigma'(s)]) = [\sigma'(s)].\) For substitution \( \rho \) we can choose \( n \) to be zero and \( s \) to be equal to \( \sigma(s) \). The most interesting observation is that then indeed \([\sigma(\text{it(\neg s, 3, 1)})]\) = \([\rho(\text{it(\neg s, 3, 1)})]\) and that \([\sigma'(\neg s)]\) = \([\rho(\neg s)]\). The other conditions are also satisfied and thus this is a cleave. We can also observe that leaving out the synchronisation of \( s \) does not yield a cleave since there is no substitution \( \rho \) meeting the conditions in \( \text{COM} \) when substitutions \( \sigma \) and \( \sigma' \) disagree on the value of \( s \).

Informally, we have already argued that the decomposition yields a state space that is bisimilar to the original monolithic LPE. We finish this section with a formal claim stating that a cleave induces a valid decomposition of a monolithic LPE. For this, we first introduce several auxiliary results and concepts. In particular, the proof of correctness of our claim relies on the notion of bisimulation up to. This notion appears in [25] and is repeated below for completeness.

**Definition 4.8.** Let \( \mathcal{L} = (S, \text{Act}, \rightarrow) \) be an LTS. A binary relation \( R \subseteq S \times S \) is a strong bisimulation up to \( \Rightarrow \) iff for all \( s R t \) it holds that:

- if \( s \overset{\alpha_i}{\rightarrow} s' \) then there is a state \( t' \in S \) such that \( t \overset{\alpha_i}{\rightarrow} t' \) and \( s' \Rightarrow t' \).
- if \( t \overset{\alpha_i}{\rightarrow} t' \) then there is a state \( s' \in S \) such that \( s \overset{\alpha_i}{\rightarrow} s' \) and \( t \Rightarrow s' \).

where the notation \( \Rightarrow \) denotes the relational composition, which is defined as \( \Rightarrow = \{(s, t) \in S \times S \mid \exists s' \in S, t' \in S : s \Rightarrow s' \land s' R t' \land t' \Rightarrow t\} \).

The following proposition is taken from [25].

**Proposition 4.9.** If \( R \) is a strong bisimulation up to \( \Rightarrow \) then \( R \subseteq \Rightarrow \)

This result establishes that if \( R \) is a strong bisimulation up to \( \Rightarrow \) then for any pair \((s, t) \in R \) we can conclude that \( s \Rightarrow t \).

We introduce two technical auxiliary lemmas to relate the transition induced by some expression \( P \in S \) to the transitions induced by applying the allow, hide and communication operators, in the same order as the composition expression defined in Definition 4.4, to \( P \). Their proofs are delegated to the appendix since these are trivial.
Lemma 4.10. Given expressions $P, Q \in S$, a set of multi-sets of action labels $A \subseteq 2^\Lambda \rightarrow \mathbb{N}$, two sets of action labels $H', H \subseteq \Lambda$, a set of communications $C \subseteq \text{Comm}$. If $P \overset{\omega}{\rightarrow} Q$ and $\theta_H(\gamma_C(\omega')) \in A'$ with $A' = A \cup \{\emptyset\}$ then:
$$
\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_H(\gamma_C(\omega'))} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))
$$

Lemma 4.11. Given expressions $P, Q' \in S$, a set of multi-sets of action labels $A \subseteq 2^\Lambda \rightarrow \mathbb{N}$, two sets of action labels $H', H \subseteq \Lambda$ and a set of communications $C \subseteq \text{Comm}$. If:
$$
\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \overset{\omega}{\rightarrow} Q'
$$
then there are $Q \in S$ and $\omega' \in \Omega$ such that $Q' = \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$, and $\omega = \theta_H(\gamma_C(\omega'))$, $P \overset{\omega'}{\rightarrow} Q$ and $\theta_H(\gamma_C(\omega')) \in A \cup \{\emptyset\}$.

We now formalise the correctness of the cleave in Theorem 4.12. The proof of this Theorem is a formalisation of the informal reasoning presented in this section of the theorem.

Theorem 4.12. Let $P(\tilde{d} : \tilde{D}) = \{\gamma\}_{i \in \Lambda} \sum_{\sigma_i \in E_i} e_i \rightarrow \alpha_i \cdot P(\tilde{g}_i)$ be an LPE with the separation tuples $(V, K^V, J^V, c^V, \alpha^V, \tilde{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \tilde{h}^W)$ for $P$ that are a cleave as defined in Definition 4.6. For every closed expression $\tilde{i} : \tilde{D}$ the composition expression defined in Definition 4.4 is strongly bisimilar to $P(\tilde{i})$ and, hence, a valid decomposition according to Definition 3.1.

Proof. Let $A = \{\alpha_i \mid i \in I\} \cup \{\alpha_i | \text{tag} \mid i \in (K_V \cup K_W)\}$, $H' = \{\text{tag}\}$, $H = \{\text{sync}^i \mid i \in I\}$ and $C = \{\text{sync}^V_i | \text{sync}^W_i \rightarrow \text{sync}^i \mid i \in I\}$. Let $R$ be the least relation such that $P(\tilde{i}') R \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\tilde{i}'_V) || P_W(\tilde{i}'_W))))))$ for closed expressions $\tilde{i}' : \tilde{D}$. We show that $R$ is a strong bisimulation relation up to $\equiv$.

We now formalise the correctness of the cleave in Theorem 4.12. The proof of this Theorem is a formalisation of the informal reasoning presented in this section of the theorem.

Theorem 4.12. Let $P(\tilde{d} : \tilde{D}) = \{\gamma\}_{i \in \Lambda} \sum_{\sigma_i \in E_i} e_i \rightarrow \alpha_i \cdot P(\tilde{g}_i)$ be an LPE with the separation tuples $(V, K^V, J^V, c^V, \alpha^V, \tilde{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \tilde{h}^W)$ for $P$ that are a cleave as defined in Definition 4.6. For every closed expression $\tilde{i} : \tilde{D}$ the composition expression defined in Definition 4.4 is strongly bisimilar to $P(\tilde{i})$ and, hence, a valid decomposition according to Definition 3.1.

Proof. Let $A = \{\alpha_i \mid i \in I\} \cup \{\alpha_i | \text{tag} \mid i \in (K_V \cup K_W)\}$, $H' = \{\text{tag}\}$, $H = \{\text{sync}^i \mid i \in I\}$ and $C = \{\text{sync}^V_i | \text{sync}^W_i \rightarrow \text{sync}^i \mid i \in I\}$. Let $R$ be the least relation such that $P(\tilde{i}') R \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\tilde{i}'_V) || P_W(\tilde{i}'_W))))))$ for closed expressions $\tilde{i}' : \tilde{D}$. We show that $R$ is a strong bisimulation relation up to $\equiv$.

Pick a closed expression $\tilde{i} : \tilde{D}$. Assume that $P(\tilde{i}) \overset{\omega}{\rightarrow} Q$. Then there is an index $r \in I$ and substitution $\sigma$ such that $\sigma(\tilde{d}) = \tilde{i}$ for which it holds that $[[\sigma(\epsilon_i^V)]$, $\omega = [[\sigma(\alpha_i)]$] and $Q' = P(\sigma(\tilde{g}_i))$. There are three cases to consider based on the index $r$.

- Case $r \in K_V$. This is essentially the case visualised in Fig. 3 (a), where the initiative comes from the monolithic process. We derive $P_V(\tilde{i}'_V) \xrightarrow{[\sigma(\alpha_i)|\text{tag}]} P_V(\sigma(\tilde{g}_i))$, because $[[\sigma(\epsilon_i^V)]$ holds. Rule ParL allows us to derive the transition $P_V(\tilde{i}'_V) || P_W(\tilde{i}'_W) \xrightarrow{[\sigma(\alpha_i)|\text{tag}]} P_V(\sigma(\tilde{g}_i)) || P_W(\tilde{i}'_W)$. By definition, $\theta_H(\gamma_C(\sigma(\epsilon_i^V)|\text{tag})) = \sigma(\alpha_i)$ and $\sigma(\epsilon_i^V) \in A \cup \{\emptyset\}$.

- Case $r \in K_W$. Follows the same line of reasoning as for case $r \in K_V$.

- Case $r \in I \setminus (K_V \cup K_W)$. This is the more complex situation sketched in Fig. 4. From requirement SYN we obtain $r \in J_V \cap J_W$. We derive the following transitions using requirement ORI. First, from $[[\sigma(\epsilon_i^V)]$ and $[[\sigma(\epsilon_i^W)]$ it follows that both:

$$
P_V(\tilde{i}'_V) \xrightarrow{[\sigma(\alpha_i)|\text{sync}^V(\tilde{h}_i^V)]} P_V(\sigma(\tilde{g}_i))
$$
and
$$
P_W(\tilde{i}'_W) \xrightarrow{[\sigma(\alpha_i)|\text{sync}^W(\tilde{h}_i^W)]} P_W(\sigma(\tilde{g}_i))
$$

Furthermore, $[[\sigma(\alpha_i)] = [[\sigma(\alpha_i)|\alpha_i]]$ and by rule Par we then derive:

$$
P_V(\tilde{i}'_V) || P_W(\tilde{i}'_W) \xrightarrow{[\sigma(\alpha_i)|\text{sync}^V(\sigma(\tilde{h}_i^V))|\text{sync}^W(\sigma(\tilde{h}_i^W))]} P_V(\sigma(\tilde{g}_i)) || P_W(\sigma(\tilde{g}_i))
$$

Note that from $[[\sigma(\tilde{h}_i^V)] = [[\sigma(\tilde{h}_i^W)]$ it follows that:

$$
\theta_H(\gamma_C(\sigma(\epsilon_i^V)|\text{sync}^V(\sigma(\tilde{h}_i^V))|\text{sync}^W(\sigma(\tilde{h}_i^W)))) = [[\sigma(\alpha_i)]
$$

Since $[[\sigma(\alpha_i)] \in A \cup \{\emptyset\}$ follows from $\sigma(\epsilon_i^V) \in A \cup \{\emptyset\}$. Lemma 4.10 allows us to derive that:

$$
\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\tilde{i}'_V) || P_W(\tilde{i}'_W)))))) \xrightarrow{[\sigma(\alpha_i)]} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\tilde{g}_i)) || P_W(\sigma(\tilde{g}_i))))))
$$

Finally, $P(\sigma(\tilde{g}_i)) R \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\tilde{g}_i)) || P_W(\sigma(\til{g}_i))))))$. 

13
• Case \( \tau_H(\nu_A(\tau_H(\Gamma_C(P_V(\bar{e}_V) \parallel P_W(\bar{e}_W)))))) \stackrel{\alpha}{\to} Q' \). By Lemma 4.11 there is an expression \( Q' = \tau_H(\nu_A(\tau_H(\Gamma_C(Q)))) \), and a multi-action \( \omega' \in \Omega \) such that \( \omega = \theta_H(\nu_H(\gamma(C(\omega')))) \). \( P_V(\bar{e}_V) \parallel P_W(\bar{e}_W) \stackrel{\omega}{\to} Q \) and \( \theta_H(\gamma(C(\omega'))) \in A \cup \{\bar{e}\} \). There are three cases where a parallel composition results in a transition. Suppose that \( P_V(\bar{e}_V) \parallel P_W(\bar{e}_W) \stackrel{\omega}{\to} Q \) is due to:

- Rule PARL and \( P_V(\bar{e}_V) \stackrel{\omega}{\to} P' \), allowing us to derive transition \( P_V(\bar{e}_V) \parallel P_W(\bar{e}_W) \stackrel{\omega}{\to} P' \parallel P_W(\bar{e}_W) \). This is the scenario depicted in Fig. 3 (a), where the initiative to execute an action comes from the composition expression. Pick an arbitrary index \( r \in J_V \). Assume that \( r \in J_V \setminus K_V \). Then the action expression contains an action labelled \( \text{sync} \), which means that \( \theta_H(\gamma(C(\omega'))) \notin A \cup \{\bar{e}\} \). Contradiction.

Hence, we conclude that \( r \in K_V \). From \( P_V(\bar{e}_V) \stackrel{\omega}{\to} P' \), we conclude that there must be a substitution \( \sigma \) such that \( \sigma(\bar{e}_V) = \bar{e}_V \) for which \( [\sigma(c_e)] \), \( \omega' = \sigma(\alpha_e) \) and \( P_V = P_V(\sigma(\bar{e}_V)) \). Let \( \sigma \) be that substitution. Since requirement \( \text{IND} \) holds, we know that \( \bar{g}_{1|W} = \bar{d}_{1|W} \). We may therefore conclude that \( P(\bar{d} \stackrel{\omega'}{\to} P(\sigma(\bar{e}_V)) \). Finally, \( P(\sigma(\bar{e}_V)) \) and \( P_W(\sigma(\bar{e}_V))) \).

- Rule PARL, and \( P_W(\bar{e}_W) \stackrel{\alpha \omega}{\to} P' \). Follows the same line of reasoning as for rule PARL.

From the observation it follows that \( \bar{g}_{1|W} = \bar{d}_{1|W} \). We then observe that \( \omega' = \sigma(\alpha_e) \) for some index \( r \in I \), because only a single tag is allowed with original action labels. Therefore, it also follows that \( r \in I \setminus (K_V \cup K_W) \) and \( [\sigma(\bar{h}_V)] = [\sigma(\bar{h}_W)] \).

From requirement \( \text{COM} \) it follows that there is a substitution \( \rho \) such that \( [\rho(C)] \) and \( [\sigma(\alpha_e)] = [\rho(\alpha_e)] \). We conclude that \( P(\bar{d} \stackrel{\omega}{\to} P(\rho(\bar{g}_{1|W})) \). Furthermore, \( [\sigma(\bar{g}_{1|W})] = [\rho(\bar{g}_{1|W})] \), and therefore by Lemma 2.7 it follows that both:

\[
\begin{align*}
\tau_H(\nu_A(\tau_H(\Gamma_C(P_V(\sigma(\bar{g}_{1|W})))), P_W(\sigma(\bar{g}_{1|W})))) & \Rightarrow \tau_H(\nu_A(\tau_H(\Gamma_C(P_V(\rho(\bar{g}_{1|W})))), P_W(\rho(\bar{g}_{1|W})))) \\
\end{align*}
\]

5. State invariants

The separation tuples inducing the decomposition obtained in Example 4.5 are indeed a cleave as shown in Example 4.7, but this is by no means the only cleave for process Machine. For instance, also the decomposition we obtained in Example 4.1 can be achieved by means of a cleave. The infinite branching of Machine (false) in that example is, however, problematic for the purpose of compositional minimisation. While in this case, as shown by Example 4.5, we could avoid the infinite branching of Machine (false) by reducing the amount of synchronisation, this might not always be possible.

Another way to restrict the behaviour of the components is to strengthen the condition expressions of each summand, thus limiting the number of outgoing transitions. We show that so-called (inductive) state invariants [26] can be used for this purpose. These state invariants are typically formulated by the user based on the understanding of the modelled behaviour.

**Definition 5.1.** Given an LPE \( P(\bar{d} : \bar{D}) = \sum_{i \in I} \sum_{c_i : E_i} a_i \cdot P(\bar{e}_i) \). A Boolean expression \( \psi \) such that \( FV(\psi) \subseteq Vars(\bar{d}) \) is called a state invariant iff the following holds: for all \( i \in I \) and substitutions \( \sigma \) satisfying \( [\sigma(c_i \land \psi)] \), also \( [\sigma(\bar{d} \leftarrow \sigma(\bar{e}_i))(\psi)] \) should hold.

The essential property of a state invariant is that whenever it holds for some given state it is guaranteed to hold for all states reachable from that state. This follows relatively straightforward from its definition. Next, we define a restricted LPE where (some of) the condition expressions are strengthened with a Boolean expression.

\[
\begin{align*}
\]
Definition 5.2. Given an LPE $P(\bar{d} : \bar{D}) = \sum_{i \in I} \sum_{c : E_i} c \rightarrow \alpha_i . P(\bar{g}_i)$, a Boolean expression $\psi$ such that $\text{FV}(\psi) \subseteq \text{Vars}(\bar{d})$ and a set of indices $J \subseteq I$. We define the restricted LPE, denoted by $P^{\psi,J}$, as follows:

$$P^{\psi,J}(\bar{d} : \bar{D}) = \sum_{i \in J} \sum_{c : E_i} c \rightarrow \alpha_i . P^{\psi,J}(\bar{g}_i)$$

Note that if the Boolean expression $\psi$ in Definition 5.2 is a state invariant for the given LPE then for all closed expressions $\tilde{\bar{d}} : \tilde{\bar{D}}$ and substitutions $\sigma$ for which $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds, it holds that $P(\tilde{\bar{d}}) \models P^{\psi,J}(\tilde{\bar{g}})$, for any $J \subseteq I$. Therefore, we can use a state invariant of an LPE to strengthen all of its condition expressions.

Moreover, a state invariant of the original LPE can also be used to restrict the behaviour of the components obtained from a cleave, as formalised in the following theorem. Note that the set of indices is used to only strengthen the condition expressions of summands that introduce synchronisation, because the condition expressions of independent summands cannot contain the other parameters as free variables. Furthermore, the restriction can be applied to independent summands before the decomposition. The theorem below states that the validity of the decomposition does not change by strengthening the components (induced by separation tuples) using state invariants.

Theorem 5.3. Let $P(\bar{d} : \bar{D}) = \sum_{i \in I} \sum_{c : E_i} c \rightarrow \alpha_i . P(\bar{g}_i)$ be an LPE with the separation tuples $(V, K^V, J^V, C^V, \alpha^V, \bar{h}^V)$ and $(W, K^W, J^W, C^W, \alpha^W, \bar{h}^W)$ for $P$. Let $\psi$ be a state invariant of $P$. For every closed expression $\tilde{\bar{d}} : \tilde{\bar{D}}$ and substitution $\sigma$ for which $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds, the following expression, where $C = J^V \cap J^W$, is a valid decomposition:

$$\tau(\mu \tau(\nu(\mu \tau(\tilde{\bar{d}}))))$$

Proof. Let $R$ be the relation that satisfies $P(\tilde{\bar{d}}) R \tau(\nu(\mu \tau(\tilde{\bar{d}})))$ for closed expression $\tilde{\bar{d}}$ exactly when $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds for some substitution $\sigma$. We show that $R$ is a strong bisimulation relation up to $\models$.

Pick a pair $P(\bar{d}) R \tau(\nu(\mu \tau(\tilde{\bar{d}}))))$.

- Case $P(\tilde{\bar{d}}) \models Q^C$. Then there is an index $r$ such that $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds for some substitution $\sigma$. We show that $Q^C$.

- Case $P(\tilde{\bar{d}}) \models Q^C$. Similarly to the proof of Theorem 4.12 we have three cases to consider.

Consider the second case where $P^{\psi,C}(\tilde{\bar{d}}) \models Q^C$ and rule ParL is used to derive $P^{\psi,C}(\tilde{\bar{d}}) \models Q^C$. We observe that if $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds then $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds as well, and similarly if $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds then $[\sigma[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds. The remainder of the proof proceeds along the lines of the proof of Theorem 4.12. Since for the substitution $\rho$ it holds that $[\rho[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds, it also follows that $[\rho[\bar{d} \leftarrow \tilde{\bar{d}}](\psi)]$ holds. Therefore, we can conclude that $P(\rho(\bar{g}_i)) R \tau(\nu(\mu \tau(\tilde{\bar{d}}))))$.

Observe that the predicate $n \leq 3$ is a state invariant of the LPE Machine in Example 2.8. Therefore, we can consider the process Machine$^{\psi,J}$ in Example 4.1 for the composition expression, which is finite. This would yield two finite components. However, the state space of Machine$^{\psi,J}$ is still larger than that of $P_W$ in Example 4.5.

Finally, we remark that the restricted state space contains deadlock states whenever the invariant does not hold. These deadlocks can be avoided by applying the invariant to the update expression of each parameter instead of the parameter itself without affecting the correctness.
6. Implementation

While Theorem 4.12 and Definition 4.6 together provide the conditions that guarantee that a cleave yields a valid decomposition, requirements (ORI) and (COM) of Definition 4.6 are difficult to ensure (and verify) due to the semantic nature of these requirements. In this section, we show how, in practice, one can cheaply approximate these correctness requirements by means of a static analysis that relies only on the expressions that occur in an LPE.

6.1. Computing a cleave

For the remainder of this section let \( P(\bar{d} : \bar{D}) = \frac{1}{|E|} \sum_{c_i \in E} c_i \rightarrow \alpha_i \cdot P(\bar{g}) \) be the LPE that we analyse. We assume that the indices for the cleave parameters \( V \) and \( W \), such that \( V \cup W = \{0, \ldots, n\} \), are given by the user. Computing a promising parameter partitioning automatically is left as future work.

The cleave procedure that is defined in Algorithm 1 yields two separation tuples for \( P \) such that these form a cleave. Apart from the user-supplied sets of indices \( V \) and \( W \), this algorithm takes an additional input \( M \) which we will explain later. The algorithm loops over all summands of the given LPE \( P \) and for each summand decides whether the summand is independent by checking the conditions for independent summands. If a summand is independent (i.e., meets requirement (IND), which can be checked cheaply), and its condition is independent of the parameters of \( W \) (which we approximate by checking whether the relevant expressions of the summand do not contain parameters from \( W \)), it is added to \( K^V \); if it is independent and its condition is independent of the parameters of \( V \), it is added to \( K^W \). Otherwise, the summand is not independent and we continue to compute the condition, action and synchronisation expressions for this dependent summand in both separation tuples. In order to construct the synchronisation expressions, we compute a set of variables \( S \), which we call synchronisation variables, that mirrors the set of parameters of the other component that may occur within relevant expressions in the summand.

Algorithm 1 Given an LPE \( P(\bar{d} : \bar{D}) = \frac{1}{|E|} \sum_{c_i \in E} c_i \rightarrow \alpha_i \cdot P(\bar{g}) \), two sets of indices \( V, W \subseteq \mathbb{N} \) and a function specifying user-defined choices \( M : I \rightarrow 2^\mathbb{N} \) returns two separation tuples that are a cleave as defined in Definition 4.6.

```
1: procedure CLEAVE(P, V, W, M)
2:   d_V ← Vars(\bar{d}_V), d_W ← Vars(\bar{d}_W)
3:   K^V, K^W ← \emptyset, \emptyset
4:   c^V, c^W, \alpha^V, \alpha^W, h^V, h^W ← \emptyset, \emptyset, \emptyset, \emptyset
5:   for i \in I do
6:       if FV(c^V) \cup FV(\alpha^V) \cup FV(\bar{g}) \subseteq d_V \cup \{c_i\} \wedge \bar{g}_{i|W} = \bar{d}_W then
7:         K^V ← K^V \cup \{i\}
8:       else if FV(c^V) \cup FV(\alpha^V) \cup FV(\bar{g}) \subseteq d_W \cup \{c_i\} \wedge \bar{g}_{i|V} = \bar{d}_V then
9:         K^W ← K^W \cup \{i\}
10:      else
11:          c^V ← c^V \cup \{l : c_i\}, c^W ← c^W \cup \{l : c_i\}
12:          \alpha^V ← \alpha^V \cup \{l : \alpha_i\}, \alpha^W ← \alpha^W \cup \{l : \alpha_i\}
13:          h^V ← h^V \cup \{l : h_{i\alpha} \in S\}, h^W ← h^W \cup \{l : h_{i\alpha} \in S\}
14:          \bar{g} ← \bar{g} \cup \{l : \bar{g}_{i|W} \cup \{l : \bar{g}_{i|V} \}
15:       end for
16:   end for
17:   \bar{g} ← \bar{g} \cup \{l : \bar{g}_{i|W} \cup \{l : \bar{g}_{i|V} \}
18:   \alpha^V ← \alpha^V \cup \{l : \alpha_i\}, \alpha^W ← \alpha^W \cup \{l : \alpha_i\}
19:   h^V ← h^V \cup \{l : h_{i\alpha} \in S\}, h^W ← h^W \cup \{l : h_{i\alpha} \in S\}
20:   \bar{g} ← \bar{g} \cup \{l : \bar{g}_{i|W} \cup \{l : \bar{g}_{i|V} \}
21:   return (V, K^V, \bar{g}, \alpha^V, \bar{g}^V), (W, K^W, \bar{g}, \alpha^W, \bar{g}^W)
```

Algorithm 1 relies on two subroutines: one routine to split the \{multi-\}action of a dependent summand over the two components, and one routine to split its condition expression. The routine for splitting a \{multi-\}action is detailed in Algorithm 2. This algorithm first checks whether the given \{multi-\}action only depends on the parameters of one component (by checking whether only parameters from \( d_V \) or \( d_W \) occur) and possibly on sum variables (in the set \( E \)). If so, then it makes sense to put that action expression in that component. In that case the action expression does not induce any \{additional\} synchronisation of parameters. There may be cases in which a \{multi-\}action relies on parameters of both \( V \) and \( W \); since in that case it is impossible to make a decent choice, we rely on input \( M \), supplied by the user, to resolve the distribution.

The routine for computing the condition expression of each component is described by Algorithm 3. We assume that the condition expression is of the shape \( \wedge_{c \in C} c \) where each element \( c \) is a clause to simplify the analysis. Note that this can always be achieved by preprocessing the expression. Ideally, the clauses are as small as possible. We provide special treatment for clauses that are equality conditions, i.e., expressions of the form \( h \approx h' \). For these type of conditions it is possible to use the synchronisation vector to ensure that \( h \) is equal to \( h' \) whenever synchronisation takes place, which can be advantageous over synchronising the dependencies of \( h \) or \( h' \) if \( h \) is closely related to one component and \( h' \) to the other
Algorithm 2 Given a multi-action $\alpha = a^0(f^0)|...|a^k(f^k)$ and sets of variables $d_V$, $d_W$ and $E$ and a set of indices $M \subseteq \mathbb{N}$, \texttt{SplitAction} computes the resulting pair of multi-actions that satisfy the property in Lemma 6.1.

1: procedure \texttt{SplitAction}(\alpha, d_V, d_W, E, M)
2: \begin{align*}
&\alpha_V, \alpha_W \leftarrow \tau, \tau \\
&\text{for } 0 \leq i \leq k \text{ do}
\end{align*}
3: \begin{align*}
&\text{if } \text{FV}(f^i) \subseteq (d_V \cup E) \land \text{FV}(f^i) \nsubseteq (d_W \cup E) \text{ then} \\
&\quad \alpha_V \leftarrow \alpha_V \cup a_i^0(f^i)
\end{align*}
4: \begin{align*}
&\text{else if } \text{FV}(f^i) \subseteq (d_W \cup E) \land \text{FV}(f^i) \nsubseteq (d_V \cup E) \text{ then} \\
&\quad \alpha_W \leftarrow \alpha_W \cup a_i^0(f^i)
\end{align*}
5: \begin{align*}
&\quad \text{else} \\
&\quad \text{if } i \in M \text{ then} \\
&\quad \quad \alpha_V \leftarrow \alpha_V \cup a_i^0(f^i)
\end{align*}
6: \begin{align*}
&\quad \text{else} \\
&\quad \quad \alpha_W \leftarrow \alpha_W \cup a_i^0(f^i)
\end{align*}
7: \begin{align*}
&\text{return } (\alpha_V, \alpha_W)
\end{align*}

The set of clauses that cannot be dealt with using synchronisation (the set $C'$ in the algorithm) must be distributed over both components. For correctness it would be sufficient to return the set of clauses $C'$, but we can improve on this by weakening the conditions using the routine \texttt{ComputeCondition}, see Algorithm 4. Consider the call to \texttt{ComputeCondition} on line 14 with the clauses $C'$ and a set of parameters $d_V$, the sum variables $E$ and the synchronisation variables $S$. The idea of this procedure is to select all clauses in $C'$ that contain (some of the) variables in $d_V \cup U \cup S$. This is useful since for local parameters in $d_V$ we need to keep the conditions anyway and for variables in $S \setminus d_V$ we want to keep conditions that can restrict their possible values as these variables will be introduced as sum variables (which is a non-deterministic choice). However, every selected clause might depend on variables that are not yet in $S'$ (which is initially equal to $S$), which is why we iterate until we reach a fixed point.

Note that there are cases for which adding all clauses that depend on variables $S'$ in this manner is not necessarily optimal. For instance, if we have an expression $x < y$, for natural numbers $x, y$, and only $y$ is included in $S'$ then adding $x$ to $S'$ introduces synchronisation for both $x$ and $y$, whereas with a more careful analysis the synchronisation of $x$ could be avoided. It is unlikely that this issue can be avoided in general, but we can imagine that specific instances can be avoided by analysing the structure of each clause.

6.2. Proof of correctness

We conclude with a proof of correctness for the presented static analysis.

Lemma 6.1. Let $\alpha$ be a multi-action, and $d_V$, $d_W$ and $E$ three sets of variables and $M \subseteq \mathbb{N}$ a set of indices. Let $(\alpha_V, \alpha_W)$ be the pair of (multi-)actions returned by \texttt{SplitAction}(\alpha, d_V, d_W, E, M). For all substitutions $\sigma$ it holds that $\llbracket \sigma(\alpha_V | \alpha_W) \rrbracket = \llbracket \sigma(\alpha) \rrbracket$.

Proof. Observe that procedure \texttt{SplitAction} terminates. The statement follows directly from the fact that every action expression is syntactically put into either $\alpha_V$ or $\alpha_W$. \qed
Algorithm 4 Given a set of Boolean expressions \( C \) and a set of synchronised variables \( S \) computes, \textsc{ComputeCondition} computes a set subset of conditions that satisfy the property in Lemma 6.2.

1: procedure \textsc{ComputeCondition}(\( C, S \))
2: \( C' \leftarrow \{ c \in C \mid \text{FV}(c) = \emptyset \} \)
3: \( S' \leftarrow S \)
4: do
5: \( S'' \leftarrow S' \)
6: for \( c \in C \) do
7: if \( \text{FV}(c) \cap S'' \neq \emptyset \) then
8: \( C' \leftarrow C' \cup \{ c \} \)
9: \( S' \leftarrow S' \cup \text{FV}(c) \)
10: while \( S'' \neq S' \)
11: return \( C' \)

Lemma 6.2. Let \( C \) be a set of clauses and \( S \) a set of variables. Then \textsc{ComputeCondition}(\( C, S \)) terminates and the set of clauses \( C' \) it returns satisfies \( C' \subseteq C \) and for all variables \( x \in S \) if there is a clause \( c' \in C \) for which \( x \in \text{FV}(c') \) then \( c' \in C' \).

Proof. First of all, observe that this procedure terminates since for the set of variables \( S', S'' \) on line 10 it holds that \( S' \) and \( S'' \) can only grow with elements in \( \bigcup_{c \in C} \text{FV}(c) \) this mean that they can only grow by a finite amount. For every clause \( c \) added to \( C' \) on lines 2 and 8 it holds that \( c \in C \), so \( C' \subseteq C \).

We need to show that for all variables \( x \in S \) if there is a clause \( c' \in C \) for which \( x \in \text{FV}(c') \) then \( c' \in C' \) holds. Finally, if \( S \) is empty then this statement holds. Therefore, assume that \( S \) is not empty and we show that the statement is a loop invariant of the do-while loop starting on line 4. Upon entry of the loop the statement does not hold, but since it is a do-while loop the body will be executed at least once. Pick an arbitrary variable \( x \in S' \) at line 5 then also \( x \in S'' \). For all \( c \in C \) if \( x \in \text{FV}(c) \) then \( c \in C' \) on the line 10 since \( \text{FV}(c) \cap S'' \neq \emptyset \) and the fact that clauses are never removed from \( C' \). Therefore, the statement holds after the first iteration of the while loop. Next, we observe that the statement is maintained in every iteration of the while loop since no elements are removed from \( C' \).

Lemma 6.3. Let \( \bigwedge_{c \in C} c \) be a condition and \( d_V, d_W, E \) and \( S \) four sets of variables such that \( \text{FV}(\bigwedge_{c \in C} c) \subseteq (d_V \cup d_W \cup E \cup S) \). Let \( (\bigwedge_{c \in C} c, \bigwedge_{c \in E} c, h_V, h_W) \) be the tuple returned by \textsc{SplitCondition}(\( \bigwedge_{c \in C} c, d_V, d_W, E, S \)). For all substitutions \( \sigma \) it holds that \( [\sigma(\bigwedge_{c \in C} c)] \iff [\sigma(\bigwedge_{c \in E} c)] \) and \( [\sigma(h_V)] \) and \( [\sigma(h_W)] = [\sigma(h_V)] \).

Proof. Pick an arbitrary substitution \( \sigma \).

\[
\Rightarrow ) \text{ Assume that } [\sigma(\bigwedge_{c \in C} c)] \text{ holds. Since } C_V \subseteq C \text{ by Lemma 6.2 it follows that } [\sigma(\bigwedge_{c \in C_V} c)] \text{ holds as well. Furthermore, consider any expression of the form } h \approx h' \in C \text{ for which both } \text{FV}(h) \neq \emptyset \text{ and } \text{FV}(h') \neq \emptyset \text{ and also } \text{FV}(h) \subseteq d_V \text{ and } \text{FV}(h') \subseteq d_V. \text{ Then from } [\sigma(\bigwedge_{c \in C} c)] \text{ it follows that } [\sigma(h)] = [\sigma(h')]. \text{ Similarly, whenever } \text{FV}(h) \subseteq d_W \text{ and } \text{FV}(h') \subseteq d_W \text{ it holds that } [\sigma(h)] = [\sigma(h')]. \text{ Therefore, it follows that } [\sigma(h_V)] = [\sigma(h_W)].
\]

\[
\Leftarrow ) \text{ Assume that } [\sigma(\bigwedge_{c \in C} c)] \text{ and } [\sigma(\bigwedge_{c \in E} c)] \text{ and } [\sigma(h_V)] \text{ and } [\sigma(h_W)] \text{ hold. For all clauses of the form } h \approx h' \in C \text{ for which } h \approx h' \notin (C_V \cup C_W) \text{ we know that } [\sigma(h)] = [\sigma(h')] \text{ because } [\sigma(h_V)] = [\sigma(h_W)] \text{ and by the construction of these vectors. Furthermore, by Lemma 6.2 we know that for all variables } x \in (d_V \cup d_W \cup E \cup S) \text{ if there is a clause } c \in C \text{ for which } x \in \text{FV}(c) \text{ then } c \in C_V. \text{ Similarly, for variables } x \in (d_V \cup d_W \cup E \cup S) \text{ if there is a clause } c \in C \text{ for which } x \in \text{FV}(c) \text{ then } c \in C_W. \text{ Finally, for all } c \in C \text{ for which } \text{FV}(c) = \emptyset \text{ it holds that } c \in (C_V \cap C_W). \text{ Since } \text{FV}(\bigwedge_{c \in C} c) \subseteq (d_V \cup d_W \cup E \cup S) \text{ we can conclude that } C_V \cup C_W = C'. \text{ Therefore, from } [\sigma(\bigwedge_{c \in C_V} c)] \text{ and } [\sigma(\bigwedge_{c \in C_W} c)] \text{ it follows that } [\sigma(\bigwedge_{c \in C} c)] \text{ holds.}
\]

Lemma 6.4. Let \( P(\overline{a}; \overline{d}) = \sum_{x \in \mathbb{N}} \sum_{y \in \mathbb{N}} A(x, y) \cdot P(x) \) be an LPE with two sets of indices \( V, W \subseteq \mathbb{N} \) and an indexed set of index sets \( M \). Let separation tuples \( (V, K^V, J^V, C^V, \alpha^V, h_V^V) \) and \( (W, K^W, J^W, C^W, \alpha^W, h_W^W) \) be the result of \textsc{Cleave}(\( P, V, W, M \)); as defined in Algorithm 1. Then these tuples are separation tuples for \( P \) as defined in Definition 4.2.

Proof. First of all, we will argue that the procedure \textsc{Cleave} (Algorithm 1) terminates. This follows almost directly from the finiteness of \( I \) and the finiteness of all expressions in the LPE. The procedures \textsc{SplitAction} and \textsc{SplitCondition} terminate by Lemmas 6.1 and 6.3.

Next, we show that \( (V, K^V, J^V, C^V, \alpha^V, h_V^V) \) and \( (W, K^W, J^W, C^W, \alpha^W, h_W^W) \) are indeed separation tuples according to Definition 4.2. Due to line 21 it follows that \( J^V \subseteq J^V \) and \( C^V, \alpha^V \) and \( h_V^V \) are all indexed sets over \( J^V \setminus K^V \). For all indices \( i \in (J^V \setminus K^V) \) we observe that \textsc{SplitAction} (Algorithm 2) indeed yields two action expressions for which \( \text{FV}(\alpha^V) \subseteq \text{FV}(\alpha_i) \). Also \textsc{SplitCondition} (Algorithm 3) returns two Boolean conditions for which \( \text{FV}(c_i^V) \subseteq \text{FV}(c_i) \) due to Lemma 6.3 and two synchronisation expressions for which \( \text{FV}(h_i^V) \subseteq \text{FV}(h_i) \). Finally, it holds that \( S \subseteq (\text{Vars}(\overline{d}) \cup \{ e_1 \}) \) after line 12 since \( P \) is an LPE and after line 14 due to the observation that \( \text{FV}(c_i^V) \subseteq \text{FV}(c_i) \) and \( \text{FV}(h_i^V) \subseteq \text{FV}(h_i) \) (and similarly for \( c_i^W \) and \( h_i^W \)).
Therefore, for all \(i \in (J^V \setminus K^V)\) it holds that \(\text{FV}(c_i^V) \cup \text{FV}(\alpha_i^V) \cup \text{FV}(\tilde{h}_i^V) \subseteq \text{Vars}(\tilde{d}) \cup \{e_i\}\) on line 20. Finally, for all \(i \in K^V\) it holds that \(\text{FV}(c_i) \cup \text{FV}(\alpha_i) \cup \text{FV}(\tilde{g}_i) \subseteq \text{Vars}(\tilde{d}) \cup \{e_i\}\) due to line 6. Thus \((V, K^V, J^V, c_i^V, \alpha_i^V, \tilde{h}_i^V)\) is a separation tuple. The same arguments can be used to show that the tuple \((W, K^W, J^W, c_i^W, \alpha_i^W, \tilde{h}_i^W)\) is a separation tuple. □

**Theorem 6.5.** Let \(P(\tilde{d} : \tilde{D}) = \sum_{i \in I} \tilde{d}_i : \tilde{E}_i \rightarrow \alpha_i . P(\tilde{g}_i)\) be an LPE with two sets of indices \(V, W \subseteq \mathbb{N}\) and an indexed set of index sets \(M\). Let \((V, K^V, J^V, c_i^V, \alpha_i^V, \tilde{h}_i^V)\) and \((W, K^W, J^W, c_i^W, \alpha_i^W, \tilde{h}_i^W)\) be the separation tuples returned by \(\text{CLEAN}(P, V, W, M)\); as defined in Algorithm 1. Then these separation tuples are a cleave as defined in Definition 4.6.

**Proof.** We verify the requirements of Definition 4.6. First of all, requirement SYN holds trivially and requirement IND also holds due to line 6 and 8. Next, we show that requirements ORI and COM are satisfied. Pick an arbitrary index \(r \in (J^V \cap J^W)\).

- Case requirement ORI. Pick a substitution \(\sigma\) satisfying \([\sigma(c_i)]\). Then \([\sigma(c_i^V)]\) and \([\sigma(c_i^W)]\) hold, and also \([\sigma(h_i^V)] = [\sigma(h_i^W)]\) by Lemma 6.3 and the construction of the separation tuples. Furthermore, \([\sigma(\alpha_i^V | c_i^W)]\) is equal to \([\sigma(\alpha_i)]\) by Lemma 6.1.

- Case requirement COM. Consider two substitutions \(\sigma\) and \(\sigma'\) satisfying \([\sigma(c_i^V)]\) and \([\sigma'(c_i^V)]\) and \([\sigma(h_i^V)] = [\sigma'(h_i^W)]\). Furthermore, let \(\rho\) be a substitution such that \([\rho(\tilde{d}_i^V)] = [\sigma(\tilde{d}_i^V)]\) and \([\rho(\tilde{g}_i)] = [\sigma'(\tilde{g}_i)]\).

First, we show that the synchronisation of variables in \(S\) on line 16 of Algorithm 1 put the necessary restrictions on \(\sigma\) and \(\sigma'\). For every variable \(x \in S\) we know from \([\sigma(h_i^V)] = [\sigma(h_i^W)]\) and the construction on line 16 that \([\sigma(x)] = [\sigma'(x)]\). Note that the order of insertion into \(\tilde{S}\) does not matter since the variables are added to both \(h_i^W\) and \(h_i^W\) in the same order. This means that \([\rho(x)] = [\rho'(x)]\) (and thus equal to \([\sigma(x)]\)).

From \((\text{FV}(c_i^V) \setminus d_i^V) \subseteq S\) it follows that \([\rho(c_i^V)] = [\rho(c_i^V)]\). The same argument applies to \(c_i^W\) for which \([\rho(c_i^W)] = [\rho(c_i^W)]\). Furthermore, since \(\text{FV}(h_i^V) \subseteq d_i^V\) and \(\text{FV}(\tilde{h}_i^W) \subseteq d_i^W\) thus \([\sigma(h_i^V)] = [\rho(h_i^V)]\) and \([\sigma(h_i^W)] = [\rho(h_i^W)]\). Therefore, we can also conclude that \([\rho(c_i)]\) holds since all equality conditions are satisfied by the synchronisation.

Finally, from \((\text{FV}(\alpha_i^V) \setminus d_i^V) \subseteq S\) and \((\text{FV}(\alpha_i^W) \setminus d_i^W) \subseteq S\) it also follows that \([\rho(\alpha_i^V)] = [\rho(\alpha_i^W)]\), \([\rho(\alpha_i^V)] = [\rho(\alpha_i^W)]\), which by Lemma 6.1 is equal to \([\rho(\alpha_i)]\). Similarly, by \((\text{FV}(\tilde{g}_i) \setminus d_i^V) \subseteq S\) it follows that \([\rho(\tilde{g}_i)] = [\rho(\tilde{g}_i)]\) and the same for \([\rho(\tilde{g}_i)]\).

7. Case studies

We have implemented our prototype based on the previously described algorithms to carry out several experiments using specifications written in the high-level language mCRL2 [1], a process algebra generalising the one of Section 2.3. To apply the decomposition technique we use the LPEs that the mCRL2 toolset [2] generates as part of the pre-processing step that the toolset performs before further analyses of the specifications are conducted. We compare the results of the monolithic exploration and the exploration based on the decomposition technique. The sources for these experiments can be obtained from the downloadable artifact [27].

Since the decomposition technique is not fully automated yet and several aspects are left as future work we use the practical examples to demonstrate the effectiveness of the technique and point out interesting observations that could be used to improve the cleave algorithm and eventually lead to a completely automated approach. In these case studies we did not manually choose the action label splitting; which means that input \(M\) in Algorithm 1 is always the empty function. Therefore, all actions are generated in the \(W\)-component when the best choice cannot be made. Furthermore, we only specified an invariant when it is explicitly stated. All experiments are performed on a laptop with an Intel Core i7-7700HQ CPU and 32GB main memory. The only exception is the connect four experiment, which has been performed on a machine with an Intel Xeon Gold 6136 CPU and 15TB main memory.

7.1. Alternating bit protocol

The alternating bit protocol (ABP) is a communication protocol that uses a single control bit, which is sent along with the message, to implement a reliable communication channel over two unreliable channels [1]. The specification contains four processes: one for the sender, one for the receiver and two for the unreliable communication channels.

First, we choose the partitioning of the parameters such that one component (ABPV) contains the parameters of the sender and one communication channel, and the other component (ABPW) contains the parameters of the receiver and the other communication channel. See Table 1 for details concerning their state spaces. We observe that component ABPV is already larger than the original state space and that it cannot be minimised further, illustrating that traditional compositional minimisation is, in this case, not particularly useful. The composition of the minimised components ABPV | ABPW is shown under ABPV | ABPW. This shows that it is possible to derive a (slightly) smaller state space.

The main reason for this disappointing result is because the behaviour of each process heavily depends on the state of the other processes, resulting in large components, as this information is lost in the decomposition. We can encode such global information as a state invariant based on the control flow parameters (see [3] for a formal definition of the notion of
a control flow parameter). The second cleave \((\text{ABP}_V^W \parallel \text{ABP}_W^V)\) for the same parameter partitioning is obtained by restricting the components using this invariant. This does yield a useful decomposition as the state spaces of these components are both smaller than the original state space, even though their composition has again a state space that is only fractionally smaller than that of the monolithic LPE. Finally, we have obtained a cleave into components \(\text{ABP}_V^W\) and \(\text{ABP}_W^V\), where the partitioning is not based on the original processes. Here we have a process \(\text{ABP}_V^W\) for the data parameter that contains the message that is being sent and all other parameters are in component \(\text{ABP}_W^V\). This yields a very effective cleave as shown in Table 1. However, this example is so small that there is hardly any difference in execution times, which are less than a second.

### 7.2. Decomposing monolithic processes

The Chatbox specification [28] describes a chat room facility in which four users can join, leave and send messages. This specification is interesting because it is described as a monolithic process, which means that compositional minimisation is not applicable in the first place. There are (at most) four users in the chatbox, so we can perform a cleave where the behaviour of one user is separated from the behaviour of the other users.

In Table 2 we use Chatbox\(_0\) to indicate the component for user 0 and Chatbox\(_{123}\) for the component containing users 1, 2 and 3. The size of the components (Chatbox\(_0\) and Chatbox\(_{123}\)) before and after minimisation modulo strong bisimulation are presented to show that these are small and can be further reduced. Furthermore, their composition (Chatbox\(_{0123}\)) shows that indeed the decomposition technique can be used quite successfully, because the result under exploration is much smaller than the original state space. Since component Chatbox\(_{123}\) contains (at most) three users we can also apply another cleave to obtain the components Chatbox\(_{1}^0\) and Chatbox\(_{23}\). In the implementation we only need to add prefixes to the synchronisation and tag action labels since these may not already occur in the LPE. After this computing the composition Chatbox\(_{123}\) shows that this is even more efficient than computing Chatbox\(_{123}\) directly.

However, we can improve upon these results even further by adapting the specification. Inspecting the Chatbox specification more carefully reveals that many summands of the LPE have disjunctive enabling conditions. This lead to additional synchronisation since the condition cannot be split easily into the two components. We have manually adapted the Chatbox to a strongly bisimilar variant Chatbox\(_{\ast}\) which avoids these disjunctive enabling conditions. The resulting decomposition shown in Table 2 shows that the amount of synchronisation transitions can be greatly reduced this way.

Another example of a monolithic process is the ConnectFour specification, which models the behaviour of a game played by two players on a board with seven columns and four rows. Using the decomposition procedure we first obtain a component for the left-most column and a component for the six remaining columns. Next, we apply the decomposition to the process for the six remaining columns recursively until we have one component for every column. In Table 2 we can see the state space of the process for only column seven. We have left out the state spaces of the other components for a single column since these are all similar in size. Then we compose columns six and seven, which is the state space listed under ConnectFour\(_{07}\) and the composition of column five, six and seven is listed under Columns ConnectFour\(_{5-7}\), etcetera. Repeating this process until we have composed all the columns shows that we can obtain a state space that is roughly half in size compared to the original state space in the number of states and transitions.

### 7.3. Practical specifications

The Register specification [29] describes a wait-free handshake register and the WMS specification is a workload management system [30], used at CERN. For these two experiments we found that partitioning the parameters into a set of control flow parameters and remaining parameters yields good results. For WMS we observed that one control flow parameter was only used in the initialisation part of the process. This meant that splitting it off from the data parameters caused this initialisation to become possible in every state, leading to many unnecessary synchronisation transitions. Therefore, we have also performed an alternative cleave into components WMS\(_V^W\) and WMS\(_W\), whose components are (much) smaller.
Table 2
State space metrics for the chatbox [28] and a connect four specification.

| Model        | exploration | minimised |
|--------------|-------------|-----------|
|              | #states | #transitions | #states | #transitions |
| Chatbox0      | 128    | 4352        | 128    | 3456        |
| Chatbox123    | 512    | 37888       | 8      | 440         |
| Chatbox0123   | 1024   | 22528       | 16     | 144         |
| Chatbox1      | 32     | 1344        | 16     | 1120        |
| Chatbox23     | 16     | 1280        | 4      | 276         |
| Chatbox0123   | 128    | 8192        | 16     | 144         |
| Chatbox*      | 65536  | 720896      | 16     | 144         |
| Chatbox*0     | 128    | 768         | 128    | 768         |
| Chatbox*1     | 32     | 224         | 32     | 224         |
| Chatbox*23    | 16     | 216         | 4      | 52          |
| Chatbox*0123  | 128    | 1920        | 8      | 104         |
| Chatbox*0123  | 1024   | 11776       | 16     | 144         |
| ConnectFour   | 4571392011 | 1881446993  | 418390653 | 2079589075 |
| ConnectFourV  | 7      | 1664        | 7      | 1664        |
| ConnectFourW  | 961    | 40992       | 961    | 40992       |
| ConnectFourV.7| 29791  | 908876      | 23327  | 713264      |
| ConnectFourW.7| 723137 | 13059584    | 503723 | 9158684     |
| ConnectFourV.7| 15615413 | 204751466  | 13560351 | 180369650   |
| ConnectFourW.7| 420370881 | 4541332512 | 326297880 | 3628882674  |
| ConnectFourV.7| 2388678550 | 10967818533 | 418390653 | 2079589075  |

Table 3
State space metrics for Hesselink’s wait-free handshake register [29] and a workload management system used at CERN [30].

| Model        | exploration | minimised |
|--------------|-------------|-----------|
|              | #states | #transitions | #states | #transitions |
| Register     | 914048  | 1885824     | 1740   | 3572        |
| RegisterV    | 464     | 10672       | 464    | 10672       |
| RegisterW    | 97280   | 273408      | 5760   | 16832       |
| RegisterVW   | 76416   | 157952      | 1740   | 3572        |
| WMS          | 155034776 | 2492918760 | 44526316 | 698524456  |
| WMSV         | 212992  | 5144576     | 212992 | 2801664     |
| WMSW         | 1903715 | 121945196   | 414540 | 26429911    |
| WMSVW        | 64635040 | 1031080812  | 44526316 | 698524456  |
| WMSV          | 311296  | 7159808     | 294912 | 6815744     |
| WMSW          | 345527  | 26084118    | 75121  | 5665871     |
| WMSVW         | 64635040 | 1049716700  | 44526316 | 698524456  |

Table 4
Execution times and maximum memory usage measurements.

| Model        | monolithic | decomposition |
|--------------|------------|---------------|
|              | time | memory | time | memory |
| Chatbox      | 4.76s | 21.9MB | 0.2s | 15.7MB |
| Register     | 7.94s | 99.7MB | 1.56s | 47.7MB |
| WMS          | 2.4h  | 15.1GB | 0.6h  | 11.8GB |
| WMS′         | 2.4h  | 15.1GB | 0.6h  | 3.0GB  |
| Connect Four | 25h   | 437GB | 20h   | 543GB  |

7.4. Execution times

We also consider the total execution time and maximum amount of memory required to obtain the original state space using exploration and the state space obtained using the decomposition technique. Table 4 shows the execution times in seconds or hours and memory required to obtain the state space under ‘exploration’ in Tables 2 and 3, excluding the final minimisation step of the original or composition state space which are only shown for reference. The cost of the static analysis of the cleave itself was in the range of several milliseconds.
Although in most cases the decomposition improves both runtime and peak memory usage this does not hold for the Connect Four specification. Here, we observe that the peak memory usage is much higher. This memory peak is caused by the strong bisimulation minimisation step for the process for columns two to seven. Although this seems worse at first we must note that the final minimisation step requires far more memory and time, whereas with the decomposition we obtain a state space that is almost half in size of monolithic exploration directly.

8. Conclusion

We have presented a decomposition technique, referred to as cleave, that can be applied to any monolithic process with the structure of an LPE and have shown that the result is always a valid decomposition. Furthermore, we have shown that state invariants can be used to improve the effectiveness of the decomposition. We consider defining a static analysis to automatically derive the parameter partitioning for the practical application of this technique as future work. Furthermore, the cleave is currently not well-suited for applying the typically more useful abstraction based on (divergence-preserving) branching bisimulation minimisation [31]. The reason for this is that τ-actions are ‘decorated’ with synchronisation actions and tags. As a result these actions become visible, and therefore effectively branching bisimilarity yields the same reduction as strong bisimilarity.

It seems that the way communication is formalised in the process algebra that we consider is essential to achieve a valid (i.e., strong bisimulation preserving) decomposition based on the data parameters. In any process algebra in which the synchronised parallel composition between processes (similar to our rule Par) is renamed into a single invisible action, often also denoted by τ, such as CCS [18] and CSP [32] it is impossible for a parallel composition of two components to mimic a visible transition of the monolithic process that requires synchronisation of data since we cannot distinguish which transition of the monolithic process it belongs to, and therefore preserve strong bisimulation. If communication does result in a visible transition then an expressive relabelling operator could be used to rename the resulting communication. Here, expressive means that it must for example be able to ignore the action data parameters that are used purely to synchronise data, but also rename actions to τ. However, such a relabelling operator is often not present in process algebras, because it does not behave nicely with respect to congruence. Finally, in the presence of global variables the synchronisation between components could easily be achieved, but then the question is how the state space of individual components can be derived. However, finding the exact minimal set of features required for a process algebra to achieve a cleave could be an interesting future direction.

On a more practical note, the main disadvantage of the current implementation is that the parameters must be provided by the user, and this requires quite some insight into the specification. As proposed earlier possible future work would be to employ the techniques in [3] to analyse the dependencies in order to automatically find a decent partitioning. Furthermore, we could use information about the original process specification from which the monolithic process is derived to guide the partitioning. In its current form the technique is difficult to apply to systems with infinite data sorts since manual analysis is required to preserve the finiteness of the components. The optimal parameter partitioning is also an algorithmic optimisation problem since the choice for the partition also influences the sizes of the intermediate state spaces.

Another disadvantage of the current implementation is that only specific summands of the components can synchronise with each other due to the unique indices assigned to the synchronisation actions. However, some early experiments suggest that it might be more efficient to allow different summands of the components to synchronise with each other whenever different summands of the monolithic LPE deal with the same action labels. Finally, the technique could be extended to allow splitting into more than two components, yielding synchronisation over multiple components simultaneously. This is also related to the problem of nested applications of the cleave procedures.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The artifact has been made available on Zenodo and used toolset is open source.

Acknowledgement

We would like to thank the anonymous reviewers of both JLAMP and ICE for their effort and constructive feedback.

Appendix A. Proof of Lemma 4.10

Lemma 4.10. Given expressions \( P, Q \in S \), a set of multi-sets of action labels \( A \subseteq 2^{\omega \rightarrow N} \), two sets of action labels \( H', H \subseteq \Lambda \), a set of communications \( C \subseteq \text{Comm} \). If \( P \xrightarrow{\omega'} Q \) and \( \theta_H(\gamma_C(\omega')) \in A' \) with \( A' = A \cup \{ \} \) then:
\[ \tau_H'(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_H'(\gamma_C(\omega'))} \tau_H'(\nabla_A(\tau_H(\Gamma_C(Q)))) \]

**Proof.** We can derive the following:

\[
\begin{array}{c}
\text{Com} \quad P \xrightarrow{\alpha'} Q \\
\text{HIDE} \quad \Gamma_C(P) \xrightarrow{\gamma_C(\omega')} \Gamma_C(Q) \\
\text{ALLOW} \quad \tau_H(\Gamma_C(P)) \xrightarrow{\theta_H(\gamma_C(\omega'))} \tau_H(\Gamma_C(Q)) \\
\text{HIDE} \quad \nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\theta_H(\gamma_C(\omega'))} \nabla_A(\tau_H(\Gamma_C(Q))) \\
\text{HIDE} \quad \tau'_H(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_H'(\gamma_C(\omega'))} \tau'_H(\nabla_A(\tau_H(\Gamma_C(Q))))
\end{array}
\]

\[ \Box \]

**Appendix B. Proof of Lemma 4.11**

**Lemma 4.11.** Given expressions \( P, Q' \in S \), a set of multi-sets of action labels \( A \subseteq 2^{A \rightarrow \mathbb{N}} \), two sets of action labels \( H', H \subseteq A \) and a set of communications \( C \subseteq \text{Comm} \). If:

\[ \tau_H'(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q' \]

then there exist \( Q \in S \) and \( \omega' \in \Omega \) such that \( Q' = \tau_H'(\nabla_A(\tau_H(\Gamma_C(Q)))) \), and \( \omega = \theta_H(\gamma_C(\omega')) \), \( P \xrightarrow{\omega'} Q \) and \( \theta_H(\gamma_C(\omega')) \in A \cup \{ \varnothing \} \).

**Proof.** Pick arbitrary expressions \( P, Q' \in S \), a set of multi-sets of action labels \( A \subseteq 2^{A \rightarrow \mathbb{N}} \), two sets of action labels \( H', H \subseteq A \) and a set of communications \( C \subseteq \text{Comm} \). Assume that \( \tau_H'(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega'} Q' \). By the syntactic structure of \( \tau_H'(\nabla_A(\tau_H(\Gamma_C(P)))) \), we must conclude that transition \( \tau_H'(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q' \) must be due to rule HIDE and \( Q' \) must be of the shape \( \tau_H'(Q_1) \) for some expression \( Q_1 \) and \( \omega = \theta_H(\omega_1) \) for some \( \omega_1 \):

\[
\text{HIDE} \quad \nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\omega_1} Q_1 \\
\tau'_H(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_H(\omega_1)} \tau'_H(\nabla_A(\Gamma_C(Q_1)))
\]

Now, we find that only the ALLOW rule has a conclusion that permits us to derive \( \nabla_A(\tau_H(\Gamma_C(Q))) \xrightarrow{\omega_1} Q_1 \), in which case \( Q_1 \) must be of the shape \( \nabla_A(Q_2) \) for some expression \( Q_2 \), and \( \omega_1 \in A \cup \{ \varnothing \} \):

\[
\text{ALLOW} \quad \tau_H(\Gamma_C(P)) \xrightarrow{\omega_1} Q_2 \\
\nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\omega_1} \nabla_A(Q_2) \xrightarrow{\omega_1} \nabla_A(Q_2) \xrightarrow{\omega_1} A \cup \{ \varnothing \}
\]

Observe that then also \( Q' \) is of the shape \( \tau_H'(\nabla_A(Q_2)) \). Again, a transition \( \tau_H(\Gamma_C(P)) \xrightarrow{\omega_1} Q_2 \) can only be derived when \( Q_2 \) is of the shape \( \tau_H(Q_3) \), for some expression \( Q_3 \), and \( \omega_1 = \theta_H(\omega_2) \) for some \( \omega_2 \):

\[
\text{HIDE} \quad \Gamma_C(P) \xrightarrow{\omega_2} Q_3 \\
\tau_H(\Gamma_C(P)) \xrightarrow{\theta_H(\omega_2)} \tau_H(Q_3)
\]

Note that then \( Q' \) is of the shape \( \tau_H'(\nabla_A(\tau_H(Q_3))) \). Finally, only the Com rule allows us to derive \( \Gamma_C(P) \xrightarrow{\omega_3} Q_3 \), in which case \( Q_3 \) is of the form \( \Gamma_C(Q_4) \), for some \( Q_4 \), and \( \omega_2 = \gamma_C(\omega_3) \) for some \( \omega_3 \).

\[
\text{Com} \quad P \xrightarrow{\omega} Q_4 \\
\Gamma_C(P) \xrightarrow{\gamma_C(\omega_3)} \Gamma_C(Q_4)
\]

For this derivation to exist it must be true that for some expression \( Q_4 \), transition \( P \xrightarrow{\omega_3} Q_4 \) exists and that \( Q' \) is of the form \( \tau_H'(\nabla_A(\tau_H(Q_4))) \), and \( \omega = \theta_H(\gamma_C(\omega_3)) \) for some \( \omega_3 \). Furthermore, it must hold that \( \omega_1 = \theta_H(\gamma_C(\omega_3)) \in A \cup \{ \varnothing \} \). \( \Box \)
