Decomposing monolithic processes in a process algebra with multi-actions

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](#)

Download date: 17. Nov. 2023

Decomposing Monolithic Processes in a Process Algebra with Multi-actions

Maurice Laveaux   Tim A.C. Willemse

Eindhoven University of Technology,
Eindhoven, The Netherlands

m.laveaux@tue.nl   t.a.c.willemse@tue.nl

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.

1 Introduction

The mCRL2 language [11] 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 *multi-action*, which allows for specifying that atomic actions can happen simultaneously.

Specifications written in mCRL2 can be analysed using the mCRL2 toolset [4]. The corresponding mCRL2 toolset [4] 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. 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. One example is a static analysis to detect live variables as presented in [17]. 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 Figure 1. Assume that this machine is described by a single recursive mCRL2 process with two parameters: a natural number representing the counter 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 Figure 2. Observe that indeed the states of a component rely on only one of the two parameters. Furthermore, note

$$\text{Machine}(3,\text{true}) \xrightarrow{\wr\text{count}\wr} \text{Machine}(2,\text{true}) \xrightarrow{\wr\text{count}\wr} \text{Machine}(1,\text{true}) \xrightarrow{\wr\text{count}\wr} \text{Machine}(0,\text{true})$$
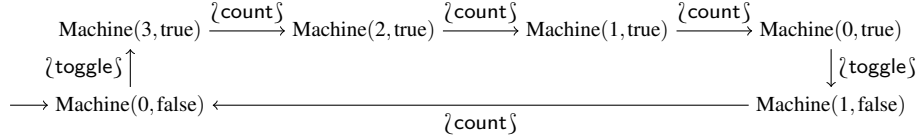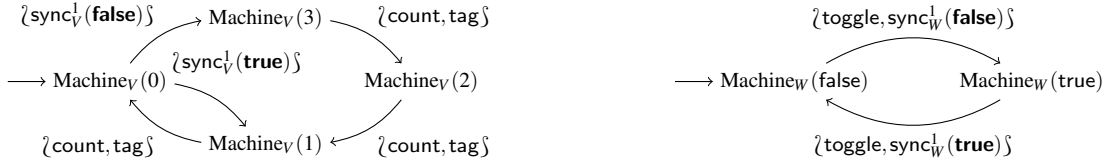
Figure 1: Behaviour of a machine.

Figure 2: Behaviour of the decomposition processes.

that the transition systems of both components include sync 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. Summarising, the contributions of our work are as follows:

- we formalise the notion of a decomposition 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,

- we show that *state invariants* [10] can be used to obtain even smaller state spaces by restricting the interfaces of the components resulting from the decomposition,

- we confirm the practical applicability of our techniques on several cases.

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 [2] also aims to speed up state space exploration by means of decomposition. The work on functional decomposition [3] describes a technique to decompose

a specification based on a partitioning of the action labels instead of a partitioning of the data parameters. In [13] it was shown how this type of decomposition can be achieved for mCRL2 processes. Furthermore, a decomposition technique was used in [10] to improve the efficiency of equivalence checking. However, that work considers processes that are already in a 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 [21, 20]. 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 exceed the size of the original state space [7]. One way to (partly) avoid this is by specifying *interface constraints* (also known as *environmental constraints* or *context constraints*), see [9, 5]. Note that 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 [6]; 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.

Outline. In Section 2 the syntax and semantics of the considered process algebra are defined. The decomposition problem is defined in Section 3 and the cleave technique is presented in Section 4. In Section 5 the cleave technique is improved with state invariants. In Section 6 the implementation is described shortly and a case study is presented in Section 7 to illustrate the effectiveness of the decomposition technique in practice. Finally, a conclusion and future work is presented in Section 8.

2   Preliminaries

We assume the existence of an abstract data theory that describes data sorts. Each sort D has an associated non-empty semantic domain denoted by $\mathbb{D}$. The existence of sorts *Bool* and *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 variable) of sort D. The set of free variables of an expression e is denoted $\mathsf{FV}(e)$, and a variable that is not free is called *bound*. An expression e is *closed* iff $\mathsf{FV}(e) = \emptyset$. A substitution σ is a total function from variables to closed data expressions of their corresponding sort. We use $\sigma(e)$ to denote the syntactic replacement of variables in expression e by their substituted expression.

An *interpretation* function, denoted by $[\![\ldots]\!]$, 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 $e \approx f$, which is true iff $[\![e]\!] = [\![f]\!]$; for other operators we use the same symbol

in both syntactic and semantic domains. We adopt the usual principle of substitutivity; *i.e.*, for all variables x, expressions e and closed expressions g and h it holds that if $g \approx h$ then $[x \leftarrow g](e) \approx [x \leftarrow h](e)$.

We denote a *vector* of length $n+1$ by $\vec{d} = \langle d_0, \ldots, d_n \rangle$. Two vectors are equivalent, denoted by $\langle d_0, \ldots, d_n \rangle \approx \langle e_0, \ldots, e_n \rangle$, iff their elements are *pairwise equivalent*, *i.e.*, $d_i \approx e_i$ for all $0 \le i \le n$. Given a vector $\langle d_0, \ldots, d_n \rangle$ and a subset $I \subseteq \mathbb{N}$, we define the *projection*, denoted by $\langle d_0, \ldots, d_n \rangle_{|I}$, as the vector $\langle d_{i_0}, \ldots, d_{i_l} \rangle$ for the largest $l \in \mathbb{N}$ such that $i_0 < i_1 < \ldots < i_l \le n$ and $i_k \in I$ for $0 \le k \le l$. We write $\vec{d} : \vec{D}$ for a vector of $n+1$ variables $d_0 : D_0, \ldots, d_n : D_n$ and denote the projection for a subset of indices $I \subseteq \mathbb{N}$ by $\vec{d}_{|I} : \vec{D}_{|I}$. Finally, we define $\mathsf{Vars}(\vec{d}) = \{d_0, \ldots, d_n\}$.

A *multi-set* over a set A is a total function $m : A \to \mathbb{N}$; we refer to $m(a)$ as the *multiplicity* of a and we write $\langle \ldots \langle$ for a multi-set where the multiplicity of each element is either written next to it or omitted when it is one. For instance, $\langle a : 2, b \langle$ has elements a and b with multiplicity two and one respectively, and all other elements have multiplicity zero. For multi-sets $m, m' : A \to \mathbb{N}$, we write $m \subseteq m'$ iff $m(a) \le m'(a)$ 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 Λ be the set of (sorted) action *labels*. We use D_a to indicate the sort of action label $a \in \Lambda$. The set of all multi-sets over $\{a(\mathbf{e}) \mid a \in \Lambda, \mathbf{e} \in \mathbb{D}_a\}$ is denoted Ω. Note that $\mathbb{D}_a$ is the semantic domain of D_a. In examples we typically omit the expression and parentheses whenever D_a consists of a single element.

Definition 2.1. A labelled transition system with multi-actions, abbreviated LTS, is a tuple $\mathscr{L} = (S, s_0, Act, \rightarrow)$ where S is a set of states; $s_0 \in S$ is an initial state; $Act \subseteq \Omega$ and $\rightarrow \subseteq S \times Act \times S$ is a labelled transition relation.

We typically use ω to denote an element of *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, and a dangling arrow indicates the initial state. The left graph of Figure 2 depicts an LTS with four states and five transitions, which are labelled with multi-actions $\langle \mathsf{count}, \mathsf{tag} \langle, \langle \mathsf{sync}_V^1(\mathbf{true}) \langle$ and $\langle \mathsf{sync}_V^1(\mathbf{false}) \langle$.

We recall the well-known strong bisimulation equivalence relation on LTSs [16].

Definition 2.2. Let $\mathscr{L}_i = (S_i, s_i, Act_i, \rightarrow_i)$ for $i \in \{1, 2\}$ be two LTSs. A binary relation $R \subseteq S_1 \times S_2$ is a *(strong) bisimulation relation* iff for all $s\,R\,t$:

- if $s \xrightarrow{\omega}_1 s'$ then there is a state $t' \in S_2$ such that $t \xrightarrow{\omega}_2 t'$ and $s'\,R\,t'$, and

- if $t \xrightarrow{\omega}_2 t'$ then there is a state $s' \in S_1$ such that $s \xrightarrow{\omega}_1 s'$ and $s'\,R\,t'$.

States s and t are *bisimilar*, denoted $s \leftrightarrow t$, iff $s\,R\,t$ for a bisimulation relation R. We write $\mathscr{L}_1 \leftrightarrow \mathscr{L}_2$ iff $s_1 \leftrightarrow s_2$ and say $\mathscr{L}_1$ and $\mathscr{L}_2$ are bisimilar.

2.2   Linear Process Equations

We draw inspiration from the process algebra mCRL2 [11], which contains multi-actions, to describe the elements of an LTS; similar concepts and constructs may appear in other shapes elsewhere.

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

$$\alpha \ ::= \ \tau \ \mid \ a(e) \ \mid \ \alpha | \alpha$$

Constant τ 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 α for any substitution σ, denoted by $[\![\alpha]\!]_\sigma$, is an element of Ω and defined inductively as follows: $[\![\tau]\!]_\sigma = \wr \jmath$, $[\![a(e)]\!]_\sigma = \wr a([\![\sigma(e)]\!]) \jmath$ and $[\![\alpha|\beta]\!]_\sigma = [\![\alpha]\!]_\sigma + [\![\beta]\!]_\sigma$. If α is a closed expression then the substitution can be omitted.

The states and transitions of an LTS are described by means of a monolithic process called a *linear process equation*, which consists 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 process *names*.

Definition 2.4. A *linear process equation* (LPE) is an equation of the form:

$$P(d : D) = \sum_{e_0:E_0} c_0 \to \alpha_0 . P(g_0) + \ \ldots \ + \sum_{e_n:E_n} c_n \to \alpha_n . P(g_n)$$

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 $\mathsf{FV}(c_i) \subseteq \{d, e_i\}$,
- α_i is a multi-action τ or $a_i^1(f_i^1)|\ldots|a_i^{n_i}(f_i^{n_i})$ such that each $a_i^k \in \Lambda$ and f_i^k is an expression of sort $D_{a_i^k}$ such that $\mathsf{FV}(f_i^k) \subseteq \{d, e_i\}$,
- g_i is an *update* expression of sort D, satisfying $\mathsf{FV}(g_i) \subseteq \{d, e_i\}$.

The $+$-operator denotes a non-deterministic choice among the summands of the 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.

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 each 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 $\sum_{e_0:E_0,\ldots,e_l:E_l}$, respectively.

The *operational semantics* of an LPE are defined by a mapping to an LTS. Let P be the set of symbols $P(\iota)$ such that $P(d : D) = \phi_P$, for any $P \in PN$, is an LPE and ι is a closed expression of sort D.

Definition 2.5. Let $P(d : D) = +_{i \in I} \sum_{e_i:E_i} c_i \to \alpha_i . P(g_i)$ be an LPE and let $\iota : D$ be a closed expression. The semantics of $P(\iota)$, denoted by $[\![P(\iota)]\!]$, is the LTS $(\mathsf{P}, P(\iota), \Omega, \to)$ where $\to$ is defined as follows: for all indices $i \in I$, closed expressions $\iota' : D$ and substitutions σ such that $\sigma(d) = \iota'$ there is a transition $P(\iota') \xrightarrow{[\![\sigma(\alpha_i)]\!]} P(\sigma(g_i))$ iff $[\![\sigma(c_i)]\!] = \mathbf{true}$.

For a given LPE, we refer to the reachable part of the LTS, induced by the LPE, as the *state space*. Note that 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 which correspond to the same semantic object, *e.g.*, $1 + 1$ and 2 for our assumed sort *Nat*, result in different states. As stated by the lemma below, such states are always bisimilar.

Lemma 2.6. Given an LPE $P(d : D) = \phi_P$. For all closed expressions $e, e' : D$ such that $[\![e \approx e']\!] = \mathbf{true}$ we have $[\![P(e)]\!] \Leftrightarrow [\![P(e')]\!]$.

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 2.7. 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 toggle; switching between modes happens after a number of clock cycles and is dependent on the mode 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 a parameter n which keeps track of the number of cycles left before switching modes.

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

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 the machine that is initially off, defined by $[\![\text{Machine}(0, \text{false})]\!]$, is shown in Figure 1.

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 [11] and similar-styled process algebras. Let Comm be the set of *communication* expressions of the form $a_0|\dots|a_n \rightarrow c$ where $a_0, \dots, a_n, c \in \Lambda$ are action labels.

Definition 2.8. The process algebra is defined as follows:

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

Here, $A \subseteq 2^{\Lambda \rightarrow \mathbb{N}}$ 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 $C \subseteq$ Comm is a finite set of *communications*. Finally, we have $P(\iota) \in \mathsf{P}$.

The set S contains all expressions of the process algebra. Operator Γ_C describes *communication*, ∇_A *action allowing*, τ_H *action hiding* and $\parallel$ *parallel composition*; the elementary objects are the processes, defined as LPEs.

The operational semantics of expressions in S are defined in Definition 2.11. We first introduce three auxiliary functions on Ω that are used in the semantics.

Definition 2.9. Given $\omega \in \Omega$ we define γ_C, where $C \subseteq$ Comm, as follows:

$$\gamma_\emptyset(\omega) = \omega$$
$$\gamma_C(\omega) = \gamma_{C \setminus C_1}(\gamma_{C_1}(\omega)) \text{ for } C_1 \subset C$$
$$\gamma_{\{a_0|\dots|a_n \rightarrow c\}}(\omega) = \begin{cases} \wr c(\mathbf{d}) \wr + \gamma_{\{a_0|\dots|a_n \rightarrow c\}}(\omega - \wr a_0(\mathbf{d}), \dots, a_n(\mathbf{d}) \wr) \\ \quad \text{if } \wr a_0(\mathbf{d}), \dots, a_n(\mathbf{d}) \wr \subseteq \omega \\ \omega \quad \text{otherwise} \end{cases}$$

For γ_C to be well-defined we require that the left-hand sides of the communications do not share labels. Furthermore, the action label on the right-hand side must not occur in any *other* left-hand side. For example $\gamma_{\{a|b \rightarrow c\}}(a|d|b) = c|d$, but $\gamma_{\{a|b \rightarrow c, a|d \rightarrow c\}}(a|d|b)$ and $\gamma_{\{a|b \rightarrow c, c \rightarrow d\}}(a|d|b)$ are not allowed.

Definition 2.10. Let $\omega \in \Omega$, $H \subseteq \Lambda$ and $\omega \in \Omega$. We define $\theta_H(\omega)$ as the multi-set ω' defined as:

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

Finally, given a multi-action α we define $\underline{\alpha}$ to obtain the multi-set of action labels, *e.g.*, $\underline{a(3)|b(5)} = \{a,b\}$. Formally, $\underline{a(e)} = \{a\}$, $\underline{\tau} = \{\}$ and $\underline{\alpha|\beta} = \underline{\alpha} + \underline{\beta}$. We define $\underline{\omega}$ for $\omega \in \Omega$ in a similar way.

Definition 2.11. The operational semantics of an expression Q of S, denoted $[\![Q]\!]$, are defined by the corresponding LTS $(\mathsf{S},Q,\Omega,\rightarrow)$ with its transition relation defined by the rules below and the transition relation given in Definition 2.5 for each expression in P. For any $\omega, \omega' \in \Omega$, expressions P,P',Q,Q' of S and sets $C \subseteq \mathsf{Comm}$, $A \subseteq 2^{\Lambda \rightarrow \mathbb{N}}$ and $H \subseteq \Lambda$:

$$\text{COM } \frac{P \xrightarrow{\omega} P'}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega)} \Gamma_C(P')} \qquad \text{ALLOW } \frac{P \xrightarrow{\omega} P'}{\nabla_A(P) \xrightarrow{\omega} \nabla_A(P')} \; \underline{\omega} \in A$$

$$\text{HIDE } \frac{P \xrightarrow{\omega} P'}{\tau_H(P) \xrightarrow{\theta_H(\omega)} \tau_H(P')} \qquad \text{PAR } \frac{P \xrightarrow{\omega} P' \quad Q \xrightarrow{\omega'} Q'}{P \parallel Q \xrightarrow{\omega + \omega'} P' \parallel Q'}$$

$$\text{PARR } \frac{Q \xrightarrow{\omega} Q'}{P \parallel Q \xrightarrow{\omega} P \parallel Q'} \qquad \text{PARL } \frac{P \xrightarrow{\omega} P'}{P \parallel Q \xrightarrow{\omega} P' \parallel Q}$$

Note that for ALLOW the condition $\underline{\omega} \in A$ must be satisfied in order for the rule to be applicable.

Example 2.12. Consider the following LPE that models a drill component in which each toggle action leads to a drill action.

$$\text{Drill}(t : Bool) = (\neg t) \rightarrow \text{toggle} \, . \, \text{Drill}(\text{true})$$
$$+ \quad (t) \rightarrow \text{drill} \, . \, \text{Drill}(\text{false})$$

Suppose that we wish to study the interaction of LPEs Machine of Example 2.7 and Drill, assuming that their toggle actions must synchronise, resulting in a $\overline{\text{toggle}}$ action. Let $C = \{\text{toggle}|\text{toggle} \rightarrow \overline{\text{toggle}}\}$ be the communication function that specifies this synchronisation, and let $A = \{\{\overline{\text{toggle}}\}, \{\text{drill}\}, \{\text{count}\}\}$ be the set of multi-action labels we allow. The interaction between LPEs Machine and Drill can be specified by the expression $\nabla_A(\Gamma_C(\text{Machine}(0,\text{false}) \parallel \text{Drill}(\text{false})))$ in the algebra. An example derivation is depicted below:

$$\text{PAR } \frac{\text{Machine}(0,\text{false}) \xrightarrow{\{\text{toggle}\}} \text{Machine}(3,\text{true}) \qquad \text{Drill}(\text{false}) \xrightarrow{\{\text{toggle}\}} \text{Drill}(\text{true})}{\text{Machine}(0,\text{false}) \parallel \text{Drill}(\text{false}) \xrightarrow{\{\text{toggle:2}\}} \text{Machine}(3,\text{true}) \parallel \text{Drill}(\text{true})}$$

$$\text{COM } \frac{}{\Gamma_C(\text{Machine}(0,\text{false}) \parallel \text{Drill}(\text{false})) \xrightarrow{\{\overline{\text{toggle}}\}} \Gamma_C(\text{Machine}(3,\text{true}) \parallel \text{Drill}(\text{true}))}$$

$$\text{ALLOW } \frac{}{\nabla_A(\Gamma_C(\text{Machine}(0,\text{false}) \parallel \text{Drill}(\text{false}))) \xrightarrow{\{\overline{\text{toggle}}\}} \nabla_A(\Gamma_C(\text{Machine}(3,\text{true}) \parallel \text{Drill}(\text{true})))} \; \{\overline{\text{toggle}}\} \in A$$

Note that we cannot derive a $\{\text{toggle}\}$ transition for $\nabla_A(\Gamma_C(\text{Machine}(0,\text{false}) \parallel \text{Drill}(\text{false})))$, even though we can derive, *e.g.*, $\text{Machine}(0,\text{false}) \parallel \text{Drill}(\text{false}) \xrightarrow{\{\text{toggle}\}} \text{Machine}(3,\text{true}) \parallel \text{Drill}(\text{false})$ by rule PARL. The reason for this is that $\{\text{toggle}\} \notin A$.

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') which formalises how to combine the components. We formalise this problem as follows.

Definition 3.1. Let $P(\vec{d} : \vec{D}) = \phi$ be an LPE and $\vec{\iota} : \vec{D}$ a closed expression. The LPEs $P_0(\vec{d}_{|I_0} : \vec{D}_{|I_0}) = \phi_0$ to $P_n(\vec{d}_{|I_n} : \vec{D}_{|I_n}) = \phi_n$, for indices $I_0, \ldots, I_n \subseteq \mathbb{N}$, are a valid *decomposition* of P and $\vec{\iota}$ iff there is a context C such that:

$$[\![P(\vec{\iota})]\!] \Leftrightarrow [\![\mathsf{C}[P_0(\vec{\iota}_{|I_0}) \parallel \ldots \parallel P_n(\vec{\iota}_{|I_n})]]\!]$$

Where $\mathsf{C}[P_0(\vec{\iota}_{|I_0}) \parallel \ldots \parallel P_n(\vec{\iota}_{|I_n})]$ is an expression in S. We refer to the expression $\mathsf{C}[P_0(\vec{\iota}_{|I_0}) \parallel \ldots \parallel P_n(\vec{\iota}_{|I_n})]$ 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 the smaller components. The primary benefit of a valid decomposition is that a state space that is equivalent to the original state space can be obtained as follows. 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. The composition resulting from these minimised components can be considerably smaller than the original state space, because also the original state space can often be reduced considerably modulo strong bisimilarity after generation. This process is referred to as *compositional minimisation*.

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 α, provided that c does not occur in α. When the multi-action α is an event that is possible in a monolithic LPE and the label c is the result of a communication of 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 2.7.

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

$$\mathrm{Machine}_V(n : Nat) = \sum_{s:Bool} (n > 0) \rightarrow \mathsf{count}|\mathsf{sync}_V^0(n, s) \,.\, \mathrm{Machine}_V(n - 1)$$

$$+ \sum_{s:Bool} (n \approx 0) \rightarrow \mathsf{sync}_V^1(n, s) \,.\, \mathrm{Machine}_V(\mathsf{if}(\neg s, 3, 1))$$

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

$$+ \sum_{n:Nat} (n \approx 0) \rightarrow \mathsf{toggle}|\mathsf{sync}_W^1(n, s) \,.\, \mathrm{Machine}_W(\neg s)$$

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 $\mathrm{Machine}_V(0)$ is shown

below. The synchronisation actions sync expose the non-deterministically chosen values of the unknown parameters.
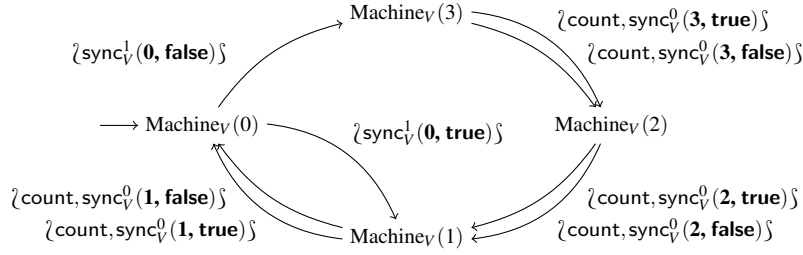
Enforcing synchronisation of the sync actions, the context C can be chosen as follows to achieve a valid decomposition:

$$\nabla_{\{\langle toggle\rangle,\langle count\rangle\}}\left(\tau_{\{sync^0,sync^1\}}\left(\Gamma_{\{sync_V^0|sync_W^0\to sync^0,\,sync_V^1|sync_W^1\to sync^1\}}\left(\mathrm{Machine}_V(0)\parallel \mathrm{Machine}_W(\mathsf{false})\right)\right)\right)$$

Unfortunately the state space of $\mathrm{Machine}_W(\mathsf{false})$ 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 $\mathrm{Machine}_W$ 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 2.7, 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. 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 $\mathrm{Machine}_V$ to independently execute (multi-)action count without synchronising the values of *s* and *n* with $\mathrm{Machine}_W$.

A second observation that we exploit is that if there are independent summands, then not every summand needs to be present in both components. However, we must ensure that each summand of the monolithic LPE is covered by at least one of the two components that we extract from the LPE. 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 the set *K*.

A third observation that can be utilised is that for the dependent summands, there is some degree of flexibility for deciding which part of the summand of the monolithic LPE will be contributed by which component. 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 (*i.e.*, information about 'missing' parameters, given by a synchronisation expression *h*) that needs to be exchanged between these two components when they execute their respective summands, can be minimised.

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 K, J and U, and the conditions c, and action α and synchronisation expressions h of a component. To define the expressions we use indexed sets where the index of each element, indicated by a subscript, determines the index of the summand to which the expression belongs.

Definition 4.2. Let $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \rightarrow \alpha_i . P(\vec{g}_i)$ be an LPE. A *separation tuple* for P is a 6-tuple (U,K,J,c^U,α^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, α^U and h^U are indexed sets of condition, action and synchronisation expressions respectively. We require that for all $i \in (J \setminus K)$ it holds that $\mathsf{FV}(c_i^U) \cup \mathsf{FV}(\alpha_i^U) \cup \mathsf{FV}(\vec{h}_i^U) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$, and for all $i \in K$ it holds that $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_{i|U}) \subseteq \mathsf{Vars}(\vec{d}_{|U}) \cup \{e_i\}$.

A separation tuple induces an LPE, where $U^c = \mathbb{N} \setminus U$, as follows:

$$P_U(\vec{d}_{|U}:\vec{D}_{|U}) = \!\!\!\!\underset{i\in(J\setminus K)}{+}\;\;\underset{e_i:E_i,\vec{d}_{|U^c}:\vec{D}_{|U^c}}{\sum} c_i^U \rightarrow \alpha_i^U|\mathsf{sync}_U^i(\vec{h}_i^U) . P_U(\vec{g}_{i|U})$$

$$+ \underset{i\in K}{+} \sum_{e_i:E_i} c_i \rightarrow \alpha_i|\mathsf{tag} . P_U(\vec{g}_{i|U})$$

We assume that action label sync_U^i, for any $i \in I$, and label tag does not occur in α_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 needed to properly deal with overlapping multi-actions, as we illustrate below in Example 4.3.

Example 4.3. Consider the following LPE.

$$P(x:Bool,y:Bool) = x \rightarrow a . P(\mathsf{false},y)$$
$$+ y \rightarrow b . P(x,\mathsf{false})$$
$$+ (x \wedge \neg y) \rightarrow a|b . P(\mathsf{false},\mathsf{false})$$

Suppose we decompose LPE P using the separation tuple $(V,\{0\},\{0,2\},\{x_2\},\{a_2\},\{\langle\rangle_2\})$ and the tuple $(W,\{1\},\{1,2\},\{(\neg y)_2\},\{b_2\},\{\langle\rangle_2\})$, where $V = \{0\}$ and $W = \{1\}$. Now assume that we had omitted the tag action in Definition 4.2, in which case these separation tuples would induce the following LPEs:

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

Now, observe both $P_V(\mathsf{true}) \xrightarrow{\langle a\rangle} P_V(\mathsf{false})$ and $P_W(\mathsf{true}) \xrightarrow{\langle b\rangle} P_W(\mathsf{false})$ are transitions for these components. This also means that due to (among others) rule PAR, $P_V(\mathsf{true}) \parallel P_W(\mathsf{true})$ can perform action $\langle a,b\rangle$. Note that process $P(\mathsf{true},\mathsf{true})$ does *not* have an outgoing transition labelled with $\langle a,b\rangle$, but (the reachable) process $P(\mathsf{true},\mathsf{false})$ *does* have an outgoing $\langle a,b\rangle$ transition. There is, however, no composition expression that prevents $\langle a,b\rangle$ in $P_V(\mathsf{true}) \parallel P_W(\mathsf{true})$ and allows $\langle a,b\rangle$ in $P_V(\mathsf{true}) \parallel P_W(\mathsf{false})$. The tag label provides the tools for making this distinction.

The components, induced by two separation tuples, can be (re)combined in a context that enforces 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 $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i . P(\vec{g}_i)$ be an LPE and $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be separation tuples for P. Let $P_V(\vec{d}_{|V} : \vec{D}_{|V}) = \phi_V$ and $P_W(\vec{d}_{|W} : \vec{D}_{|W}) = \phi_W$ be the induced LPEs according to Definition 4.2. Let $\iota : \vec{D}$ be a closed expression. Then the composition expression is defined as:

$$\tau_{\{tag\}} \left(\nabla_{\{\underline{\alpha_i} \mid i \in I\} \cup \{\alpha_i | tag \mid i \in (K^V \cup K^W)\}} \left(\tau_{\{sync^i \mid i \in I\}} \left(\Gamma_{\{sync_V^i | sync_W^i \to sync^i \mid i \in I\}} \left(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \right) \right) \right) \right)$$

Before we proceed to identify the conditions under which two separation tuples induce a valid decomposition using the above context, we revisit Example 2.7 to illustrate the concepts introduced so far.

Example 4.5. Reconsider the LPE presented in Example 2.7 with $V = \{0\}$ and $W = \{1\}$. The separation tuple $(V, \{0\}, \{0, 1\}, \{(n \approx 0)_1\}, \{\tau_1\}, \{\langle s \rangle_1\})$ and $(W, \emptyset, \{1\}, \{true_1\}, \{toggle_1\}, \{\langle s \rangle_1\})$ for Machine induce component $Machine_V$ and $Machine_W$ respectively.

$$Machine_V(n : Nat) = \quad (n > 0) \to count|tag . Machine_V(n - 1)$$
$$+ \sum_{s:Bool} (n \approx 0) \to \tau|sync_V^1(s) . Machine_V(if(\neg s, 3, 1))$$
$$Machine_W(s : Bool) = true \to toggle|sync_W^1(s) . Machine_W(\neg s)$$

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

$$\tau_{\{tag\}} \left(\nabla_{\{\underline{toggle}, \underline{count}, \underline{count,tag}\}} \left(\tau_{\{sync^0, sync^1\}} \right.\right.$$
$$\left.\left. \Gamma_{\{sync_V^0 | sync_W^0 \to sync^0, sync_V^1 | sync_W^1 \to sync^1\}} (Machine_V(0) \parallel Machine_W(false)) \right) \right)$$

The state space of this expression is strongly bisimilar to the state space of $Machine(0, false)$ shown in Figure 1. Note that the state space of $Machine_V(0)$ has four states and transitions, and the state space of $Machine_W(false)$ 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 Figure 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 true in Example 4.5 of the summand in P_W by 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.

Consider a decomposition of an LPE P according to Definition 4.4, induced by separation tuples $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$. We abbreviate the composition expression of Definition 4.4 by $C[P_V(\vec{d}_{|V}) \| P_W(\vec{d}_{|W})]$. 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 $\mathsf{C}[P_V(\vec{d}_{|V})||P_W(\vec{d}_{|W})]$. Bisimilarity requires that states that are related can mimic each other's steps. Since three LPEs are involved (the LPE P 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.

<div align="center">

(a) Independent: $i \in K^V$ (b) Independent: $i \in K^W$

</div>

$$P(\vec{d}) \cdots\cdots \mathsf{C}[P_V(\vec{d}_{|V})||P_W(\vec{d}_{|W})] \qquad\qquad P(\vec{d}) \cdots\cdots \mathsf{C}[P_V(\vec{d}_{|V})||P_W(\vec{d}_{|W})]$$

$$\Big\downarrow \alpha_i \qquad\qquad \Big\downarrow \alpha_i \qquad\qquad\qquad\qquad \Big\downarrow \alpha_i \qquad\qquad \Big\downarrow \alpha_i$$

$$P(\vec{g}_i) \cdots\cdots \mathsf{C}[P_V(\vec{g}_{i|V})||P_W(\vec{d}_{|W})] \qquad\qquad P(\vec{g}_i) \cdots\cdots \mathsf{C}[P_V(\vec{d}_{|V})||P_W(\vec{g}_{i|W})]$$
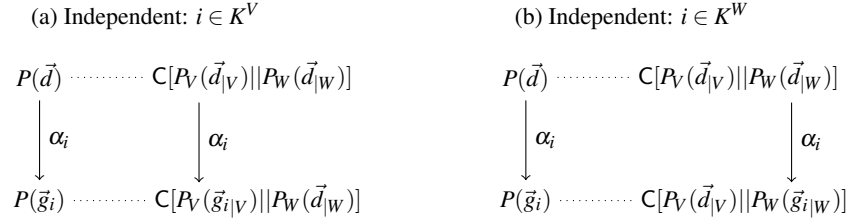
Figure 3: Two of the possible situations that must be considered when showing the validity of the decomposition of Definition 4.4: the execution of independent summands depicted in situations (a) and (b).

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

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 Figure 3—Definition 4.4 guarantees that the free variables of their condition, action and update expressions are taken from $\vec{d}_{|V}$; (multi-)action α_i matches (multi-)action $\alpha_i|\mathsf{tag}$ 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 Figure 3 is dual. Formally, we require (IND): for all $r \in K^V$ we have $\vec{g}_{r|W} = \vec{d}_{|W}$ and for all $r \in K^W$ we demand $\vec{g}_{r|V} = \vec{d}_{|V}$. Note that in case K^V and K^W overlap, condition (IND) guarantees that the involved summands only induce self-loops. Finally, observe that (IND) is also a sufficient condition for the monolithic LPE P to match a (multi-)action α_i due to P_V or P_W in either of these two cases.

$$P(\vec{d}) \cdots\cdots \mathsf{C}[P_V(\vec{d}_{|V})||P_W(\vec{d}_{|W})]$$

$$\Big\downarrow \alpha_i \qquad\qquad\qquad \Big\downarrow \alpha_i$$

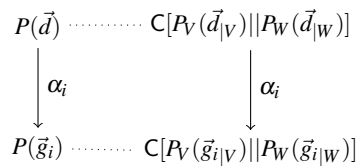$$P(\vec{g}_i) \cdots\cdots \mathsf{C}[P_V(\vec{g}_{i|V})||P_W(\vec{g}_{i|W})]$$

Figure 4: The third possible situation that must be considered when showing the validity of the decomposition of Definition 4.4: the synchronous execution of summands.

The more complex scenario that must be considered is when P_V and P_W (must) synchronise to mimic the behaviour of P; see Figure 4. Suppose again that the monolithic LPE P can execute an α_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 (SYN): $J^V = I \setminus K^W$ and $J^W = I \setminus K^V$.

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 $\alpha_i^V | \alpha_i^W$ must reduce to α_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 the executing the r-indexed summands, indeed is the same as could have resulted from executing the r-indexed summand in P. Contrary to requirement (ORI), this property is not guaranteed by the construction of Definition 4.2, so 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.

Definition 4.6. Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i . P(\vec{g}_i)$ be an LPE and $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be separation tuples for P as defined in Definition 4.2. The two separation tuples are a *cleave* of P iff the following requirements hold.

SYN. $J^V = I \setminus K^W$ and $J^W = I \setminus K^V$.

IND. For all $r \in K^V$, $\vec{g}_{r|W} = \vec{d}_{|W}$, and for all $r \in K^W$, $\vec{g}_{r|V} = \vec{d}_{|V}$.

ORI. For all $r \in (J^V \cap J^W)$ and substitutions σ satisfying $[\![\sigma(c_r)]\!]$, also:
- $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma(c_r^W)]\!]$, and
- $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma(\vec{h}_r^W))]\!]$, and
- $[\![\sigma(\alpha_r^V | \alpha_r^W)]\!] = [\![\sigma(\alpha_r)]\!]$.

COM. For all $r \in (J^V \cap J^W)$ and substitutions σ and σ' satisfying $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma'(c_r^W)]\!]$ and $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$, there is a substitution ρ such that $[\![\rho(\vec{d}_{|V})]\!] = [\![\sigma(\vec{d}_{|V})]\!]$ and $[\![\rho(\vec{d}_{|W})]\!] = [\![\sigma'(\vec{d}_{|W})]\!]$ and:
- $[\![\rho(c_r)]\!]$, and
- $[\![\sigma(\alpha_r^V) | \sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r)]\!]$, and
- $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$, and
- $[\![\sigma'(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$.

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 σ 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, $[\![\sigma(\langle s \rangle)]\!] = [\![\sigma(\langle s \rangle)]\!]$ and $[\![\sigma(\tau | \text{toggle})]\!] = [\![\sigma(\text{toggle})]\!]$ by definition. For the requirement COM consider

any two substitutions σ and σ' such that both $[\![\sigma(n \approx 0)]\!]$ and $[\![\sigma'(\text{true})]\!]$ hold and $[\![\sigma(\langle s \rangle)]\!] = [\![\sigma'(\langle s \rangle)]\!]$. For substitution ρ we can choose n to be zero and s to be equal to $[\![\sigma(\langle s \rangle)]\!]$. The most interesting observation is that then indeed $[\![\sigma(\text{if}(\neg s, 3, 1))]\!] = [\![\rho(\text{if}(\neg s, 3, 1))]\!]$ 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 ρ meeting the conditions in COM when substitutions σ and σ' 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. The complete proof can be found in the technical report [15].

Theorem 4.8. Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i . P(\vec{g}_i)$ be an LPE and let $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be a cleave as defined in Definition 4.6. For a closed expression $\vec{\iota} : \vec{D}$ the interpretation of the composition expression defined in Definition 4.4 is strongly bisimilar to $[\![P(\vec{\iota})]\!]$ and as such a valid decomposition according to Definition 3.1.

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 Machine. For instance, also the decomposition we obtained in Example 4.1 can be achieved by means of a cleave. The infinite branching of $\text{Machine}_W(\text{false})$ 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 $\text{Machine}_W(\text{false})$ 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 *state invariants* [1] can be used for this purpose. These state invariants are typically formulated by the user based on the understanding of the model behaviour.

Definition 5.1. Given an LPE $P(d : D) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i . P(g_i)$. A boolean expression ψ such that $\text{FV}(\psi) \subseteq \{d\}$ is called a *state invariant* iff the following holds: for all $i \in I$ and closed expressions $\iota : D$ and $l : E_i$ such that $[\![[d \leftarrow \iota, e_i \leftarrow l](c_i \wedge \psi)]\!]$ holds then $[\](\psi)]\!]$ holds as well.

The essential property of a state invariant is that whenever it holds for the initial state it is guaranteed to hold for all reachable states in the state space. 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.

Definition 5.2. Given an LPE $P(d : D) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i . P(g_i)$, a boolean expression ψ such that $\text{FV}(\psi) \subseteq \{d\}$ and a set of indices $J \subseteq I$. We define the restricted LPE, denoted by $P^{\psi, J}$, as follows:

$$P^{\psi, J}(d : D) = +_{i \in J} \sum_{e_i : E_i} c_i \wedge \psi \to \alpha_i . P^{\psi, J}(g_i)$$
$$+ +_{i \in (I \setminus J)} \sum_{e_i : E_i} c_i \to \alpha_i . P^{\psi, J}(g_i)$$

Note that if the boolean expression ψ in Definition 5.2 is a state invariant for the given LPE then for all closed expressions $\vec{\iota} : \vec{D}$ such that $[\]\!]$ holds, it holds that $[\![P(\vec{\iota})]\!] \Leftrightarrow [\![P^{\psi, J}(\vec{\iota})]\!]$, 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. A proof can be found in the technical report [15].

Theorem 5.3. Let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i . P(\vec{g}_i)$ be an LPE and $(V, K^V, J^V, c^V, \alpha^V, h^V)$ and $(W, K^W, J^W, c^W, \alpha^W, h^W)$ be separation tuples as defined in Definition 4.2. Let ψ be a state invariant of P. Given a closed expression $\vec{\iota} : \vec{D}$ such that $[\ ]\!]$ holds, the following expression, where $C = J^V \cap J^W$, is a valid decomposition:

$$\tau_{\{\mathsf{tag}\}} \left(\nabla_{\{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i} \mid \mathsf{tag} \mid i \in (Iind_V \cup K_W)\}} \left(\tau_{\{\mathsf{sync}^i \mid i \in I\}} \left(\Gamma_{\{\mathsf{sync}^i_V \mid \mathsf{sync}^i_W \to \mathsf{sync}^i \mid i \in I\}} (P^{\psi, C}_V(\vec{\iota}_{|V}) \parallel P^{\psi, C}_W(\vec{\iota}_{|W})) \right) \right) \right)$$

Observe that the predicate $n \leq 3$ is a state invariant of the LPE Machine in Example 2.7. Therefore, we can consider the process $\mathrm{Machine}^{\psi, I}_W$ in Example 4.1 for the composition expression, which is finite. This would yield two finite components, but the state space of $\mathrm{Machine}^{\psi, I}_W$ is 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 Theorems 4.8 and 5.3, and Definition 4.6 together provide the conditions that guarantee that a cleave yields a valid decomposition, requirements (ORI) and (COM) of the latter definition are difficult to ensure due to the semantic nature of these requirements. In practice, we need to effectively approximate these correctness requirements using static analysis.

While we leave it to future work to investigate to what extent a precise and efficient static analysis is possible, we have implemented an automated prototype translation that, given a user-supplied partitioning of the process parameters of the monolithic LPE, exploits a simple static analysis to obtain components that are guaranteed to satisfy the requirements of a cleave. First of all the prototype identifies the independent summands in both components. Furthermore, we decide on each clause of a conjunctive condition and action in the action expression where it belongs. This analysis is based on the observation that whenever all free variables of an expression occur in one component then that expression should be kept in that component, and thus be removed from the other component.

7   Case Study

We have used our prototype to carry out several experiments using specifications written in the high-level language mCRL2 [11], a process algebra generalising the one of Section 2.3. To apply the decomposition technique we use the LPEs that the mCRL2 toolset [4] generates as part of the pre-processing step the toolset uses 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 artefact [14].

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 [11]. 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 (ABP_V) contains the parameters of the sender and one communication channel, and the other component (ABP_W) contains the parameters of the receiver and the other communication channel. See Table 1 for details concerning their state spaces. We observe that component ABP_V 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. However, the composition of the minimised components, listed under $\mathsf{ABP}_V \| \mathsf{ABP}_W$, shows that it is possible to derive a (slightly) smaller state space.

Table 1: Metrics for the alternating bit protocol.

Model	original		minimised	
	#states	#trans	#states	#trans
ABP	182	230	48	58
ABP_V	204	512	204	512
ABP_W	64	196	60	192
$\mathsf{ABP}_V \| \mathsf{ABP}_W$	172	220	48	58
ABP_V^{ψ}	52	90	22	44
ABP_W^{ψ}	22	44	20	42
$\mathsf{ABP}_V^{\psi} \| \mathsf{ABP}_W^{\psi}$	172	220	48	58
ABP_V'	5	35	5	35
ABP_W'	78	126	28	46
$\mathsf{ABP}_V' \| \mathsf{ABP}_W'$	76	90	48	58

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. The second cleave ($\mathsf{ABP}_V^{\psi} \| \mathsf{ABP}_W^{\psi}$) 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 ABP_V' and ABP_W' where the partitioning is not based on the original processes. This yields a very effective cleave as shown in Table 1.

7.2   Practical Examples

The Chatbox specification [19] 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. The size of the components ($\mathsf{Chatbox}_V$ and $\mathsf{Chatbox}_W$) before and after minimisation modulo strong bisimulation are presented to show that these are small and can be further reduced. The composition ($\mathsf{Chatbox}_V \| \mathsf{Chatbox}_W$) shows that indeed

Table 2: State space metrics for various practical specifications.

Model	Ref	exploration		minimised	
		#states	#transitions	#states	#transitions
Chatbox	[19]	65 536	2 621 440	16	144
Chatbox$_V$		128	4 352	128	3 456
Chatbox$_W$		512	37 888	8	440
Chatbox$_V$ ∥ Chatbox$_W$		1 024	22 528	16	144
Register	[12]	914 048	1 885 824	1 740	3 572
Register$_V$		464	10 672	464	10 672
Register$_W$		97 280	273 408	5 760	16 832
Register$_V$ ∥ Register$_W$		76 416	157 952	1 740	3 572
WMS	[18]	155 034 776	2 492 918 760	44 526 316	698 524 456
WMS$_V$		212 992	5 144 576	212 992	2 801 664
WMS$_W$		1 903 715	121 945 196	414 540	26 429 911
WMS$_V$ ∥ WMS$_W$		64 635 040	1 031 080 812	44 526 316	698 524 456

the decomposition technique can be used quite successfully, because the result under exploration is much smaller than the original state space. Finally, we have also listed the size of the minimised original state space (which is equal to the minimised composition) as it indicates the best possible result. The Register specification [12] describes a wait-free handshake register and the WMS specification a workload management system [18], used at CERN. For the latter two experiments we found that partitioning the parameters into a set of so-called *control flow parameters* and remaining parameters yields the best results.

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, for which the results can be found in Table 3. The execution times in **s**econds or **h**ours required to obtain the state space under 'exploration' in Table 2, 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.

Table 3: 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	0.8h	11.8GB

7.3 Connect Four

The Connect Four specification models the behaviour of a game played by two players on a board with seven columns and four rows. For this specification we can show that the decomposition can be applied

recursively to obtain more than two components. Using the decomposition procedure we first obtain a monolithic process for the left-most column and one process for the six remaining columns. Next, we apply the decomposition to the process for the six remaining columns until we have one monolithic process for every column. In Table 4 we can see the state space of the process for only column seven, which is similar to the size of the other components. We then compose columns six and seven, which is the state space listed under Columns $6\cdots7$ and the composition of column five, six and seven is listed under Columns $5\cdots7$, *etcetera*. Repeating this process until we have composed all the columns shows that we can obtain a state space that is roughly a quarter in size compared to the original state space in the number of states and transitions. Both the monolithic exploration and the decomposition take about 30 hours. However, the monolithic exploration required about 500GB memory whereas the decomposition requires about 234GB of main memory. Furthermore, the state space is also immediately smaller.

Table 4: State space metrics for the connect four specification.

Model	original		minimised	
	#states	#trans	#states	#trans
Connect Four	4 571 392 011	17 968 443 566	418 390 653	2 079 589 075
Column 7	31	1 664	31	161
Columns $6\cdots7$	961	7 853	961	7 751
Columns $5\cdots7$	29 791	298 579	22 821	230 003
Columns $4\cdots7$	707 451	8 179 465	336 537	4 138 521
Columns $3\cdots7$	11 362 647	112 362 633	6 112 522	60 088 448
Columns $2\cdots7$	189 489 112	1 824 751 492	92 251 708	908 605 682
Columns $1\cdots7$	1 049 255 356	4 889 577 305	418 390 653	2 079 589 075

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 [8]. The reason for this is that τ-actions might be extended with synchronisation actions and tags. As a result they become visible, effectively reducing branching bisimilarity to strong bisimilarity.

Acknowledgement

References

[1] Marc Bezem & Jan Friso Groote (1994): *Invariants in Process Algebra with Data*. In Bengt Jonsson & Joachim Parrow, editors: *CONCUR*, *LNCS* 836, Springer, pp. 401–416, doi:10.1007/978-3-540-48654-1_30.

[2] Pierre Bouvier, Hubert Garavel & Hernán Ponce de León (2020): *Automatic Decomposition of Petri Nets into Automata Networks - A Synthetic Account*. In Ryszard Janicki, Natalia Sidorova & Thomas Chatain, editors: *Application and Theory of Petri Nets and Concurrency - 41st International Conference, PETRI NETS 2020, Paris, France, June 24-25, 2020, Proceedings*, Lecture Notes in Computer Science 12152, Springer, pp. 3–23, doi:10.1007/978-3-030-51831-8_1.

[3] Ed Brinksma, Rom Langerak & Peter Broekroelofs (1993): *Functionality Decomposition by Compositional Correctness Preserving Transformation*. In Costas Courcoubetis, editor: *CAV*, *LNCS* 697, Springer, pp. 371–384, doi:10.1007/3-540-56922-7_31.

[4] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*. In Tomás Vojnar & Lijun Zhang, editors: *TACAS*, *LNCS* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[5] Shing-Chi Cheung & Jeff Kramer (1996): *Context Constraints for Compositional Reachability Analysis*. *ACM Trans. Softw. Eng. Methodol.* 5(4), pp. 334–377, doi:10.1145/235321.235323.

[6] Pepijn Crouzen & Frédéric Lang (2011): *Smart Reduction*. In: *FASE*, Lecture Notes in Computer Science 6603, Springer, pp. 111–126, doi:10.1007/978-3-642-19811-3_9.

[7] Hubert Garavel, Frédéric Lang & Laurent Mounier (2018): *Compositional Verification in Action*. In Falk Howar & Jiri Barnat, editors: *FMICS*, *LNCS* 11119, Springer, pp. 189–210, doi:10.1007/978-3-030-00244-2_13.

[8] R. J. van Glabbeek, B. Luttik & N. Trčka (2009): *Branching Bisimilarity with Explicit Divergence*. *Fundam. Inform.* 93(4), pp. 371–392, doi:10.3233/FI-2009-109.

[9] Susanne Graf, Bernhard Steffen & Gerald Lüttgen (1996): *Compositional Minimisation of Finite State Systems Using Interface Specifications*. *Formal Asp. Comput.* 8(5), pp. 607–616, doi:10.1007/BF01211911.

[10] Jan Friso Groote & Faron Moller (1992): *Verification of Parallel Systems via Decomposition*. In Rance Cleaveland, editor: *CONCUR*, *LNCS* 630, Springer, pp. 62–76, doi:10.1007/BFb0084783.

[11] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and Analysis of Communicating Systems*. MIT Press, doi:10.7551/mitpress/9946.001.0001.

[12] Wim H. Hesselink (1998): *Invariants for the Construction of a Handshake Register*. *Inf. Process. Lett.* 68(4), pp. 173–177, doi:10.1016/S0020-0190(98)00158-6.

[13] Sung-Shik T. Q. Jongmans, Dave Clarke & José Proença (2016): *A procedure for splitting data-aware processes and its application to coordination*. *Sci. Comput. Program.* 115-116, pp. 47–78, doi:10.1016/j.scico.2014.02.017.

[14] M. Laveaux (2021): *Downloadable sources for the case study*. doi:10.5281/zenodo.5091850.

[15] Maurice Laveaux & Tim A. C. Willemse (2020): *Decompositional Minimisation of Monolithic Processes*. *CoRR* abs/2012.06468. Available at https://arxiv.org/abs/2012.06468.

[16] Robin Milner (1983): *Calculi for Synchrony and Asynchrony*. *Theor. Comput. Sci.* 25, pp. 267–310, doi:10.1016/0304-3975(83)90114-7.

[17] Jaco van de Pol & Mark Timmer (2009): *State Space Reduction of Linear Processes Using Control Flow Reconstruction*. In Zhiming Liu & Anders P. Ravn, editors: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*, Lecture Notes in Computer Science 5799, Springer, pp. 54–68, doi:10.1007/978-3-642-04761-9_5.

[18] Daniela Remenska, Tim A. C. Willemse, Kees Verstoep, Wan J. Fokkink, Jeff Templon & Henri E. Bal (2012): *Using Model Checking to Analyze the System Behavior of the LHC Production Grid*. In: *12th IEEE/ACM*

International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012, IEEE Computer Society, pp. 335–343, doi:10.1109/CCGrid.2012.90.

[19] Judi Romijn & Jan Springintveld (1998): *Exploiting Symmetry in Protocol Testing*. In Stanislaw Budkowski, Ana R. Cavalli & Elie Najm, editors: *FORTE XI / PSTV XVIII, IFIP Conference Proceedings* 135, Kluwer, pp. 337–352, doi:10.1007/978-0-387-35394-4_29.

[20] Kuo-Chung Tai & Pramod V. Koppol (1993): *Hierarchy-based incremental analysis of communication protocols*. In: *ICNP*, IEEE Computer Society, pp. 318–325, doi:10.1109/ICNP.1993.340896.

[21] Kuo-Chung Tai & Pramod V. Koppol (1993): *An Incremental Approach to Reachability Analysis of Distributed Programs*. In Jack C. Wileden, editor: *IWSSD*, IEEE Computer Society, pp. 141–151, doi:10.1109/IWSSD.1993.315504.