The Journal of Logic and Algebraic Programming 81 (2012) 522–540

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Static Analysis of IMC

Nataliya Skrypnyuk [a,*], Flemming Nielson [a], Henrik Pilegaard [b]

[a] Informatics and Mathematical Modelling, Richard Petersens Plads Bldg. 321, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark
[b] Kapow Software, Dr. Neergaards Vej 5A, DK-2970 Hørsholm, Denmark

ARTICLE INFO

Article history:
Available online 21 March 2012

Keywords:
Process algebras
Data Flow Analysis
Pathway Analysis
IMC

ABSTRACT

Process algebras formalism is highly suitable for producing succinct descriptions of reactive concurrent systems. Process algebras allow to represent them in a compositional way, as processes that run in parallel and interact, for example, through synchronisation or message passing. On the other hand, checking properties on process algebraic descriptions is often hard, while "unfolding" them into the Labelled Transition Systems can lead to the infamous state space explosion problem.

In this work we use a subtype of Data Flow Analysis on systems defined by finite-state process algebras with CSP-type synchronisation – in particular, on our variant of IMC with a more permissive syntax, i.e. with a possibility to start a bounded number of new processes. We prove that the defined Pathway Analysis captures all the properties of the systems, i.e. is precise. The results of the Pathway Analysis can be therefore used as an intermediate representation format, which is more concise than the Labelled Transition System with all the states explicitly represented and more suitable for devising efficient verification algorithms of concurrent systems than their process algebraic descriptions – see, for example, the reachability algorithm in Skrypnyuk and Nielson (2011) [17].

1. Introduction

Many interesting concurrent systems can be modelled by means of process algebras (also called process calculi). The last provide operators that make it possible to specify systems in a compositional way, by building more complex systems out of simple ones. The syntax of process algebras is namely suitable for specifying subsystems and interactions between them separately, leading to concise and compositional syntactic descriptions.

We are usually interested in some qualitative or quantitative information about the systems – for example, in questions concerning safety or liveness properties, performance, dependability guarantees, etc. The usual method of answering such questions is to "unfold" a process description in the syntax of the corresponding process algebra into the Labelled Transition System (LTS). The subsystems distinguishable in the process algebra are not recognisable anymore in the LTS; in return, all the behaviour of the system is explicitly represented. Consequently, model-checking algorithms [2] or another kinds of analysis are run on the constructed LTS in order to verify whether the system has the required properties.

One potential problem with this approach is the infamous state space explosion problem: a small description of a process in the corresponding algebra's syntax can give rise to a large or even infinite LTS. Another possible approach is to use so-called Static Analysis techniques to analyse the system's syntax directly. Static Analysis has been initially developed in the field of imperative programming languages. It was often used in order to check for errors in a program without actually executing it, purely by analysing the code (see [7] for a detailed description of different Static Analysis techniques). Several Static Analysis techniques have been adapted to a variety of process algebras with a similar purpose – to deduce properties of system's

* Corresponding author.
E-mail addresses: nsk@imm.dtu.dk (N. Skrypnyuk), nielson@imm.dtu.dk (F. Nielson), henrik@pilegaard.org (H. Pilegaard).

1567-8326/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jlap.2012.03.011

behaviour directly from the syntax, thus avoiding building the full LTS. In particular, Control Flow and Data Flow Analysis have been applied to the calculi of CCS, BioAmbients, π-calculus, etc. (see [6,9–11,13]).

In this work we will use the Data Flow Analysis method – see, for example, [5] for the theoretical background of the Monotone Data Flow Analysis Frameworks. Data Flow Analysis can be computed in the size of the syntax of the program or process algebraic expression to which it is applied, and for process algebras it can be computed in a compositional way, i.e. after adding a new parallel process to the system the previously computed analysis results can be reused. We call our method Pathway Analysis in order to distinguish it from the classical Data Flow Analysis. The name has been chosen in relation to the previous work of two of the authors of this paper [8,13] where similar methods have been applied to the BioAmbients calculus in order to compute all possible pathways in a cell.

In the previous work (e.g. [10,11]) the Data Flow Analysis was used on process algebraic expressions in order to construct finite systems that would simulate original systems that might have an infinite state space. In the current work we have a different purpose. We intend, first, to show that Data Flow Analysis is also applicable to a stochastic process algebra and to a process algebra with the CSP-style synchronisation (see [4] for the definition of the CSP calculus), i.e. on the algebra of Interactive Markov Chains or IMC (see [1]). We show how to employ a more permissive syntax, with a bounded number of new processes that can be started, and still guarantee the finiteness of the system's state space (through checking the *well-formedness* conditions). Moreover, we prove that Pathway Analysis operators deliver results on IMC systems that are stable under any number of transitions. They represent thus an alternative and often more convenient format for describing concurrent systems with a CSP-style synchronisation (recursion unfolding is not necessary). In particular, the algorithms for computing an overapproximation of reachable states (see [17]), as well as for computing bisimulation relations and average time reachability (see Ph.D. thesis [16]) have been devised based on the precomputed Pathway Analysis.

We present the syntax and the Structural Operational Semantics of our variant of IMC (so-called "guarded" IMC, or IMC^{G}) in Section 2. Our analysis is applied to "well-behaved" IMC^{G} expressions, called IMC^{G} *programs*. The conditions for IMC^{G} expressions to be considered programs, in particular, the *well-formedness* condition, are described in Section 3. In Section 4 we introduce a number of operators on the syntax of IMC^{G} that either have been adopted for IMC^{G} from the previous work (see, for example, [10]) or have been defined anew. The main theoretical results concerning the correctness of the proposed Pathway Analysis and it's stability under any number of transitions are proved in Section 5. In order to illustrate that the Pathway Analysis results are still valid after any number of transitions, we present the so-called Worklist algorithm in Section 5 which builds a transition systems strongly bisimilar to the transition system induced by the analysed IMC^{G} program based on the Pathway Analysis results (only those states are merged together by the Worklist algorithm which differ in their copied into the syntax process definitions). The paper is concluded with a discussion of the perspectives of the presented method in Section 6. The proofs for all the lemmas are presented in Appendix A.

This work is a part of the Ph.D. thesis of the first author ([16]), with a notation and proofs being simplified in the current work, and has been presented as the work in progress on the NWPT workshop [18]. As already mentioned, some of the introduced operators on the syntax of IMC are based on the previous work. The well-formedness conditions for IMC are inspired by the well-formedness conditions for BioAmbients in Pilegaard et al. [13]. In general, however, our approach and proof strategies differ considerably from the previous work.

2. Introduction to IMC^{G} calculus

2.1. Syntax

We have chosen to perform the analysis of the algebra of Interactive Markov Chains, or IMC (see [1,3]), which is an orthogonal extension of both Markov Chains and traditional algebras. This means that IMC processes can execute both actions and exponentially distributed delays, and there is no direct connection between them (i.e. actions are assumed to take no time). Actions can sometimes only be executable together with other actions (i.e. *synchronised* with other actions) while delays do not synchronise or in other way influence each other. Note that even though we only have exponentially distributed delays in IMC, we can approximate any delay duration through them. We shall write processes in a guarded variant of Interactive Markov Chains (see the syntax rules (1–2) in Table 1) for which we have coined the name IMC^{G}.

When defining this language we shall assume a countable set of *actions*, **Act**, and a distinguished internal action, τ, such that $\tau \notin \mathbf{Act}$. An infinite set of constants, $\mathbf{Rate} \subseteq \mathbb{R}^+$, shall be used to describe Markovian rates. We shall use α to range over $(\mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate})$. Furthermore, we shall draw upon a countable set of labels, **Lab**, in order to annotate action and rate prefixes that occur in processes. Finally, a countable set of process variables, also called process identifiers, **Var**, shall assist us in the definition of recursive processes.

The syntax of the language, which is shown in Table 1, comprises the following syntactic classes: *action* and *rate guarded process variables* (1–2), *action* and *rate prefixed processes* (3-4), *sums* or *choice constructs* (5), *scope restrictions* or *hide constructs* (6), *parallel compositions* or *synchronisation constructs* (7), *recursive process definitions* (8), and *terminal processes* (9). Scope restrictions and synchronisation compositions are parametrised by sets of action names from **Act**: these are namely actions that are "internalised" or hidden in the first case and synchronised in the second case.

The reason for using *guarded process variables* in the rules (1–2) is to ensure that process variables only occur in guarded positions, which will, in particular, guarantee that processes are well-defined: we are excluding, for example, $\underline{X := X}$. This

Table 1
Syntax of IMC^{G}: $a \in \mathbf{Act} \cup \{\tau\}$, $\lambda \in \mathbf{Rate}$, $\ell \in \mathbf{Lab}$, $X \in \mathbf{Var}$, $A \subseteq \mathbf{Act}$.

$P ::=$	$a^{\ell}.X$	(1)
	$\lambda^{\ell}.X$	(2)
	$a^{\ell}.P$	(3)
	$\lambda^{\ell}.P$	(4)
	$P + P$	(5)
	hide A in P	(6)
	$P \shortparallel A \shortparallel P$	(7)
	$\underline{X := P}$	(8)
	$\mathbf{0}$	(9)

will also make some of the proofs of the lemmas in this paper easier. Apart from this highly practical restriction, there is one more difference with the syntax of IMC in Hermanns [3]: we allow the application of synchronisation and internalisation constructs (rules (7) and (6)) not only on the highest syntactic level, i.e. they can also occur inside process definitions. This, as we will see from the definition of the semantics in Section 2.2, could potentially lead to an infinite state space. We will, however, exclude such cases with the help of the well-formedness conditions that will be introduced in Section 3.

We also assign labels to all the actions and rates. As we will see in Table 3 describing the Structural Operational Semantics of IMC^{G}, labels do not have a special semantic meaning, but we will make an active use of them in the definition of our analysis and proving its correctness. Synchronisation in IMC^{G} is similarly to IMC defined in the CSP-style (compare to the definition of the CSP calculus in Hoare [4]), i.e. any number of actions can be synchronised. Process definitions in IMC^{G}, as in the IMC-variant from Brinksma and Hermanns [1], perform two functions: they define process variables and represent process expressions at the same time. Therefore, for example, $\underline{X := P}$ represents the process X where $X = P$. As usual, we consider only IMC^{G} processes with finite syntactic definitions, with a finite number of action names, delay rates, variables and labels occurring in them.

An IMC^{G} expression is called *process identifier closed* (or simply *closed*) if all process variables in it are *bound* or not *free*, i.e. are contained in process recursive definitions complying with the syntactic rule (8) from Table 1. For example, $\underline{X := a^{\ell}.X}$ is considered to be closed while $a^{\ell_1}.X.\underline{X := a^{\ell_2}.X}$ is not, because the first X is not bound.

We will often conduct proofs by induction on the syntax of IMC^{G} expressions. This means that we will be proving that some property holds for all subexpressions of some IMC^{G} expression. Formally speaking, the property holds for all E' such that $E' \preceq E$. The subexpression relation $\preceq$ is defined in the intuitive way: i.e. E' can be turned into E by applying to it any number of syntactic constructs from Table 1. In particular, E is a subexpression of itself. We also define the function *Labs* that returns a set of labels that occur inside its argument: for example, $Labs(\underline{X := a^{\ell}.X}) = \{\ell\}$.

2.2. Semantics

In this section we will present the Structural Operational Semantics (introduced in Plotkin [14]), shortly SOS, of IMC^{G} and prove several results that will be useful in our further considerations. We have adopted the SOS rules from Brinksma and Hermanns [1] with one exception: transitions are additionally decorated with so-called *multisets of labels* that are elements from the domain $\mathfrak{M}$ introduced in Table 2. Elements of $\mathfrak{M}$ represent labels that are involved in the transition derivation. They will also be used in defining and proving the correctness of our analysis.

We define $\mathfrak{M}$ as a set of functions that assign each label from **Lab** a positive natural number or zero – the number of occurrences. The least element of $\mathfrak{M}$ is denoted $\bot_{\mathfrak{M}}$ and is defined in a natural way, as a function that assigns zero to all the labels. The sum, upper and lower bound operators on the elements of $\mathfrak{M}$ are defined in a straightforward way, while for the subtraction we have to pay attention that labels cannot be assigned negative numbers in $\mathfrak{M}$: the smallest number of label's occurrences is zero.

The domains $\mathfrak{N}$ and $\mathfrak{V}$ defined in Table 2 are sets of functions from accordingly labels and variables into $\mathfrak{M}$ (they will be used in the analysis), while the domain $\mathfrak{L}$ contains mappings between labels and their corresponding names (if the mapping is not known then the symbol ? is returned). The mapping between labels and names for a particular expression can be computed by the function **ln** defined in Table 9. The function **name** returns an action name or delay rate for a multiset of labels if it is the same for all the labels mapped to a positive number in the multiset (a set of such labels is returned by the function **dom**). The internal action τ will be returned instead of some external action a by the function $\mathbf{name}^{\mathbf{h}}$ if the last is parameterised by some set A not containing a (A will in general contain action names not internalised by the **hide**-construct).

Two expressions E and E' are connected by a transition relation if $E \xrightarrow[C]{\alpha} E'$ can be derived from the rules in Table 3 for some $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ and $C \in \mathfrak{M}$. We call the transition $\xrightarrow[C]{\alpha}$ "enabled" or "executable" for E, and we say that E' is a derivative expression or simply derivation of E obtained after the execution of the transition $\xrightarrow[C]{\alpha}$. A *Labelled Transition System*

Table 2
Definition of data structures $\mathfrak{M}, \mathfrak{N}, \mathfrak{V}$ and $\mathfrak{L}$, with $M_1, M_2, C \in \mathfrak{M}, N_1, N_2 \in \mathfrak{N}, V_1, V_2 \in \mathfrak{V}, L_1, L_2, L \in \mathfrak{L}$.

$\mathfrak{M}$	$::=$	$\mathbf{Lab} \to \mathbb{N}_0$	
$\bot_{\mathfrak{M}}(\ell)$	$=$	0	for all $\ell \in \mathbf{Lab}$
$M_1 \leq M_2$	$\Leftrightarrow$	$M_1(\ell) \leq M_2(\ell)$	for all $\ell \in \mathbf{Lab}$
$[M_1 + M_2](\ell)$	$=$	$M_1(\ell) + M_2(\ell)$	for all $\ell \in \mathbf{Lab}$
$[M_1 - M_2](\ell)$	$=$	$M_1(\ell) - M_2(\ell)$	if $M_1(\ell) \geq M_2(\ell)$
$[M_1 - M_2](\ell)$	$=$	0	if $M_1(\ell) < M_2(\ell)$
$[M_1 \sqcup M_2](\ell)$	$=$	$max(M_1(\ell), M_2(\ell))$	for all $\ell \in \mathbf{Lab}$
$[M_1 \sqcap M_2](\ell)$	$=$	$min(M_1(\ell), M_2(\ell))$	for all $\ell \in \mathbf{Lab}$
$\mathbf{dom}(C)$	$=$	$\{\ell \in \mathbf{Lab} \mid C(\ell) > 0\}$	
$\mathfrak{N}$	$::=$	$\mathbf{Lab} \to \mathfrak{M}$	
$\bot_{\mathfrak{N}}(\ell)$	$=$	$\bot_{\mathfrak{M}}$	for all $\ell \in \mathbf{Lab}$
$[N_1 \sqcup N_2](\ell)$	$=$	$N_1(\ell) \sqcup N_2(\ell)$	for all $\ell \in \mathbf{Lab}$
$\mathfrak{V}$	$::=$	$\mathbf{Var} \to \mathfrak{M}$	
$\bot_{\mathfrak{V}}(X)$	$=$	$\bot_{\mathfrak{M}}$	for all $X \in \mathbf{Var}$
$[V_1 \sqcup V_2](X)$	$=$	$V_1(X) \sqcup V_2(X)$	for all $X \in \mathbf{Var}$
$\mathfrak{L}$	$::=$	$\mathbf{Lab} \to \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate} \cup \{?\}$	
$\bot_{\mathfrak{L}}(\ell)$	$=$	$?$	for all $\ell \in \mathbf{Lab}$
$[L_1 \sqcup L_2](\ell)$	$=$	α	if $L_1(\ell) = L_2(\ell) = \alpha$
$[L_1 \sqcup L_2](\ell)$	$=$	$?$	if $L_1(\ell) \neq L_2(\ell)$
$L_1 \leq L_2$	$\Leftrightarrow$	$L_1(\ell) = L_2(\ell)$	for all ℓ s.t. $L_1(\ell) \neq ?$
$\mathbf{name}_{\Lambda}(C)$	$=$	α	if $\Lambda(\ell) = \alpha$ for all $\ell \in \mathbf{dom}(C)$
$\mathbf{name}_{\Lambda}(C)$	$=$	$?$	if $\nexists \alpha$ s.t. $\Lambda(\ell) = \alpha$ for all $\ell \in \mathbf{dom}(C)$
$\mathbf{name}^{\mathbf{h}}{}_{\Lambda,A}(C)$	$=$	$\mathbf{name}_{\Lambda}(C)$	if $\mathbf{name}_{\Lambda}(C) \in \mathbf{Rate} \cup A \cup \{?\}$
$\mathbf{name}^{\mathbf{h}}{}_{\Lambda,A}(C)$	$=$	τ	otherwise

(LTS) can be constructed for an IMC$^{\mathrm{G}}$ expression E by registering all the possible transition relations for E and its derivative expressions. The relation $\xrightarrow{*}$ is defined a reflexive and transitive closure of the relations $\xrightarrow[C]{\alpha}$ for all $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ and $C \in \mathfrak{M}$.

The rules in Table 3 show how syntactic terms from IMC$^{\mathrm{G}}$ can be put into correspondence with Labelled Transition Systems in a compositional manner. Most of the rules closely follow usual SOS rules for process algebras: i.e. the rules for prefixing – (1) and (10), choice – (2–3) and (11–12), synchronisation – (4–6) and (13–14), hiding – (7–8) and (15), and recursion unfolding – (9) and (16). The term $E\{\underline{X := E}/X\}$ denotes the expression E with every free occurrence of the variable X in it substituted by the expression $\underline{X := E}$. In the following we may refer to an action, delay rate, or their corresponding label as being "executable" if they occur in the rules (1) and (10) that are used in the derivation of some transition.

Note that the rules for transitions decorated with action names and decorated with rates ("actions"- and "delay"- or "Markovian" transitions) are very similar in Table 3. The only difference is that we have conditions $F \overset{\tau}{\not\to}$ and $E \overset{\tau}{\not\to}$ for Markovian transitions of the choice and synchronisation constructs in the rules (11–14) (i.e. no internal transitions should be enabled for F and E). This rule is due to different interpretations of actions and rates: actions are executed instantaneously, while executing a rate means that the system performs a delay with the duration which is exponentially distributed with the corresponding rate. The duration of the delay cannot be zero, therefore an internal action which can be executed immediately has precedence, see [1]. We need to take into account all enabled Markovian transitions in order to correctly compute average waiting time and transition probabilities. The SOS of IMC$^{\mathrm{G}}$ is therefore defined as a multirelation for delay transitions (i.e. $\underline{X := \lambda^{\ell}.X + \lambda^{\ell}.X}$ has two different delay transitions), similarly to IMC in Brinksma and Hermanns [1].

We will now state two lemmas concerning the SOS rules of IMC$^{\mathrm{G}}$ that will be used later on in the proofs. Lemma 1 asserts that closed IMC$^{\mathrm{G}}$ expressions give rise only to closed IMC$^{\mathrm{G}}$ expressions as a result of any number of semantic transitions, which shows that the semantics is well-defined. Lemma 2 states that no internal transition is possible for an IMC$^{\mathrm{G}}$ process with an enabled delay transition (which is quite obvious but useful to be stated explicitly).

Lemma 1 (Preservation of IMC$^{\mathrm{G}}$ syntax)**.** *Given a closed* IMC$^{\mathrm{G}}$ *expression* E *and* $E \xrightarrow[C]{\alpha} E'$*, then* E' *is also a closed* IMC$^{\mathrm{G}}$ *expression.*

Lemma 2 (Delay transitions)**.** *Given an* IMC$^{\mathrm{G}}$ *expression* E *such that* $E \xrightarrow{\alpha}$ *for some* $\alpha \in \boldsymbol{Rate}$*, then* $E \overset{\tau}{\not\to}$.

Table 3
Structural operational semantics of IMC^{G}: $\mathsf{a} \in \mathbf{Act} \cup \{\tau\}$, $C \in \mathfrak{M}$, $\lambda \in \mathbf{Rate}$, $\ell \in \mathbf{Lab}$, $X \in \mathbf{Var}$, $A \subseteq \mathbf{Act}$.

$$\mathsf{a}^{\ell}.E \xrightarrow[\bot_{\mathfrak{M}}\,[\ell \mapsto 1]]{\mathsf{a}} E \quad (1) \qquad (\lambda^{\ell}).E \xrightarrow[\bot_{\mathfrak{M}}\,[\ell \mapsto 1]]{\lambda} E \quad (10)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E'}{E + F \xrightarrow[C]{\mathsf{a}} E'} \quad (2) \qquad \frac{E \xrightarrow[C]{\lambda} E' \quad F \overset{\tau}{\not\to}}{E + F \xrightarrow[C]{\lambda} E'} \quad (11)$$

$$\frac{F \xrightarrow[C]{\mathsf{a}} F'}{E + F \xrightarrow[C]{\mathsf{a}} F'} \quad (3) \qquad \frac{F \xrightarrow[C]{\lambda} F' \quad E \overset{\tau}{\not\to}}{E + F \xrightarrow[C]{\lambda} F'} \quad (12)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E' \quad \mathsf{a} \notin A}{E \parallel A \parallel F \xrightarrow[C]{\mathsf{a}} E' \parallel A \parallel F} \quad (4) \qquad \frac{E \xrightarrow[C]{\lambda} E' \quad F \overset{\tau}{\not\to}}{E \parallel A \parallel F \xrightarrow[C]{\lambda} E' \parallel A \parallel F} \quad (13)$$

$$\frac{F \xrightarrow[C]{\mathsf{a}} F' \quad \mathsf{a} \notin A}{E \parallel A \parallel F \xrightarrow[C]{\mathsf{a}} E \parallel A \parallel F'} \quad (5) \qquad \frac{F \xrightarrow[C]{\lambda} F' \quad E \overset{\tau}{\not\to}}{E \parallel A \parallel F \xrightarrow[C]{\lambda} E \parallel A \parallel F'} \quad (14)$$

$$\frac{E \xrightarrow[C_1]{\mathsf{a}} E' \quad F \xrightarrow[C_2]{\mathsf{a}} F' \quad \mathsf{a} \in A}{E \parallel A \parallel F \xrightarrow[C_1 + C_2]{\mathsf{a}} E' \parallel A \parallel F'} \quad (6)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E' \quad \mathsf{a} \notin A}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow[C]{\mathsf{a}} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (7) \qquad \frac{E \xrightarrow[C]{\lambda} E'}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow[C]{\lambda} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (15)$$

$$\frac{E \xrightarrow[C]{\mathsf{a}} E' \quad \mathsf{a} \in A}{\mathsf{hide}\ A\ \mathsf{in}\ E \xrightarrow[C]{\tau} \mathsf{hide}\ A\ \mathsf{in}\ E'} \quad (8)$$

$$\frac{E\{\underline{X := E}/X\} \xrightarrow[C]{\mathsf{a}} E'}{\underline{X := E} \xrightarrow[C]{\mathsf{a}} E'} \quad (9) \qquad \frac{E\{\underline{X := E}/X\} \xrightarrow[C]{\lambda} E'}{\underline{X := E} \xrightarrow[C]{\lambda} E'} \quad (16)$$

3. Well-formed IMC^{G}

Syntactic rules for IMC^{G} in Table 1 are very permissive and allow for infinite state spaces. This is, however, not our intention. In this section we will introduce a number of rules such that if the syntactic description of an IMC^{G} system fulfils these rules, then it is suitable for our analysis. In particular, its state space is guaranteed to be finite.

Before presenting the conditions, we will introduce two auxiliary operators on the syntax of IMC^{G} which will be used in the definitions below. The operator **fn** in Table 4 captures the notion of so-called *free names*. These are external action names that occur in IMC^{G} expressions and are not internalised, i.e. are not contained in any set A of the hide-construct applied on top of the action appearance. For example, $\mathbf{fn}(\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_1}.\mathbf{0}) = \emptyset$ but $\mathbf{fn}(\mathsf{a}^{\ell_1}.\mathbf{0}) = \{\mathsf{a}\}$. Free action names decorate semantic transitions in an unchanged way and can synchronise with each other, while hidden names decorate corresponding transitions with the τ-action and are not involved in any synchronisation.

Table 4
Operator $\mathbf{fn} : \text{IMC}^{G} \to 2^{\mathbf{Act}}$ computing free (i.e. non-internalised) names of IMC^{G} expressions.

$$\begin{aligned}
\mathbf{fn}(\mathsf{a}^{\ell}.X) &= \{\mathsf{a}\} && \text{if } \mathsf{a} \in \mathbf{Act} && (1)\\
\mathbf{fn}(\lambda^{\ell}.X) &= \emptyset && \text{otherwise} && (2)\\
\mathbf{fn}(\mathsf{a}^{\ell}.P) &= \{\mathsf{a}\} \cup \mathbf{fn}(\mathbf{P}) && && (3)\\
\mathbf{fn}(\lambda^{\ell}.P) &= \mathbf{fn}(P) && && (4)\\
\mathbf{fn}(P_1 + P_2) &= \mathbf{fn}(P_1) \cup \mathbf{fn}(P_2) && && (5)\\
\mathbf{fn}(\mathsf{hide}\ A\ \mathsf{in}\ P) &= \mathbf{fn}(P) \backslash A && && (6)\\
\mathbf{fn}(P_1 \parallel A \parallel P_2) &= \mathbf{fn}(P_1) \cup \mathbf{fn}(P_2) && && (7)\\
\mathbf{fn}(\underline{X := P}) &= \mathbf{fn}(P) && && (8)\\
\mathbf{fn}(\mathbf{0}) &= \emptyset && && (9)
\end{aligned}$$

Table 5
Operator $\mathbf{fpi} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathbf{Var}}$ computing free (i.e. not bound) process identifiers of $\mathrm{IMC}^{\mathrm{G}}$ expressions.

$$
\begin{array}{rcll}
\mathbf{fpi}(\mathsf{a}^{\ell}.X) & = & \{X\} & (1)\\
\mathbf{fpi}(\lambda^{\ell}.X) & = & \{X\} & (2)\\
\mathbf{fpi}(\mathsf{a}^{\ell}.P) & = & \mathbf{fpi}(P) & (3)\\
\mathbf{fpi}(\lambda^{\ell}.P) & = & \mathbf{fpi}(P) & (4)\\
\mathbf{fpi}(P_1 + P_2) & = & \mathbf{fpi}(P_1) \cup \mathbf{fpi}(P_2) & (5)\\
\mathbf{fpi}(\mathsf{hide}\ A\ \mathsf{in}\ P) & = & \mathbf{fpi}(P) & (6)\\
\mathbf{fpi}(P_1 \shuffle A \shuffle P_2) & = & \mathbf{fpi}(P_1) \cup \mathbf{fpi}(P_2) & (7)\\
\mathbf{fpi}(\underline{X := P}) & = & \mathbf{fpi}(P) \backslash \{X\} & (8)\\
\mathbf{fpi}(\mathbf{0}) & = & \emptyset & (9)
\end{array}
$$

The *free process identifiers* operator **fpi** is defined Table 5. It returns for a given $\mathrm{IMC}^{\mathrm{G}}$ expression a set of process variables that occur in it free, i.e. are not bound. For example, $\mathbf{fpi}(\mathsf{a}^{\ell}.X) = \{X\}$ but $\mathbf{fpi}(\underline{X := \mathsf{a}^{\ell}.X}) = \emptyset$. It is clear that an $\mathrm{IMC}^{\mathrm{G}}$ expression F is process identifier closed iff $\mathbf{fpi}(F) = \emptyset$.

Well-formedness rules for $\mathrm{IMC}^{\mathrm{G}}$ are given in Table 6. An $\mathrm{IMC}^{\mathrm{G}}$ expression E is considered well-formed if, first, only process identifier closed processes are put in parallel (rule (7)), and second, no free name can be hidden (rule (6)). The first condition is checked inductively on the syntax of E, while for the second condition we pass a set of free names as a parameter: i.e. we check whether $\vdash_{\mathbf{fn}(E)} E$ holds, see Definition 1. Note that, according to the well-formedness rules, E is also well-formed if we prove that $\vdash_S E$ for some $S \supseteq \mathbf{fn}(E)$. Additionally, the set of free names is enlarged in the rule (7) by the free names of synchronising processes in order to exclude cases with changed abilities to synchronise (see the example below).

Definition 1 (Well-formed $\mathrm{IMC}^{\mathrm{G}}$)**.** An $\mathrm{IMC}^{\mathrm{G}}$ expression E is called well-formed if $\vdash_{\mathbf{fn}(E)} E$ holds.

Without the rule (6) in Table 6, there could be cases where initially external actions become internal after several transitions, which we would like to avoid. For example, after the transition
$\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \xrightarrow[\perp_{\mathfrak{M}}[\ell_2 \mapsto 1]]{\tau} \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X}$
the initially free external action a^{ℓ_1} becomes hidden and will be executed as τ. It appears, however, that in order to make the "behaviour" of $\mathrm{IMC}^{\mathrm{G}}$ expressions predictable it is not enough to only rule out expressions where "globally" free action names can become hidden after a number of transitions. Consider the following $\mathrm{IMC}^{\mathrm{G}}$ transitions:

$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \shuffle \{\mathsf{a}\} \shuffle \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}) \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_3 \mapsto 1]]{\tau}$$
$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \shuffle \{\mathsf{a}\} \shuffle \mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.\underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}) \text{ and}$$

$$\mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ (\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{hide}\ \{\mathsf{a}\}\ \mathsf{in}\ \mathsf{a}^{\ell_2}.X} \shuffle \{\mathsf{a}\} \shuffle \underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}) \xrightarrow[\perp_{\mathfrak{M}}[\ell_2 \mapsto 1, \ell_3 \mapsto 1]]{\tau}$$

Table 6
Well-formedness rules for $\mathrm{IMC}^{\mathrm{G}}$ expressions, with $S \subseteq \mathbf{Act}$.

$$
\begin{array}{rcll}
 & \vdash_S \mathsf{a}^{\ell}.X & & (1)\\
 & \vdash_S \lambda^{\ell}.X & & (2)\\
\dfrac{\vdash_S P}{\vdash_S \mathsf{a}^{\ell}.P} & & & (3)\\
\dfrac{\vdash_S P}{\vdash_S \lambda^{\ell}.P} & & & (4)\\
\dfrac{\vdash_S P_1 \quad \vdash_S P_2}{\vdash_S P_1 + P_2} & & & (5)\\
\dfrac{\vdash_S P}{\vdash_S \mathsf{hide}\ A\ \mathsf{in}\ P} & & \text{if } A \cap S = \emptyset & (6)\\
\dfrac{\vdash_{S \cup \mathbf{fn}(P_1)} P_1 \quad \vdash_{S \cup \mathbf{fn}(P_2)} P_2}{\vdash_S P_1 \shuffle A \shuffle P_2} & & \text{if } (\mathbf{fpi}(P_1) = \emptyset) \wedge (\mathbf{fpi}(P_2) = \emptyset) & (7)\\
\dfrac{\vdash_S P}{\vdash_S \underline{X := P}} & & & (8)\\
 & \vdash_S X & & (9)\\
 & \vdash_S \mathbf{0} & & (10)
\end{array}
$$

hide {a} in (hide {a} in ($\underline{X := \mathsf{a}^{\ell_1}.X + \text{hide } \{\mathsf{a}\} \text{ in } \mathsf{a}^{\ell_2}.X}$) ⫴ {a} ⫴ $\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.\underline{Y := \mathsf{a}^{\ell_3}.\mathsf{a}^{\ell_4}.\mathsf{b}^{\ell_5}.Y}$).

The action a is globally hidden in all the expressions above. However, in the first case the actions a^{ℓ_1} and a^{ℓ_4} will synchronise with each other, while in the second case a^{ℓ_1} becomes internalised and can be executed on its own and a^{ℓ_4} cannot be executed at all. This is the reason why the sets of free names are updated in the rule (7) in Table 3.

We also exclude synchronising IMC$^{\mathrm{G}}$ processes with free process identifiers (rule (7)) because such expressions can potentially "grow" by means of creating several copies of the same process after recursion unfoldings. Consider the following example, where two copies of the process Y have been created:

$\underline{X := \mathsf{a}^{\ell_1}.X} \mathbin{⫴} \{\} \mathbin{⫴} \underline{Y := \mathsf{b}^{\ell_2}.Y} \xrightarrow{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X} \mathbin{⫴} \{\} \mathbin{⫴} \underline{Y := \mathsf{b}^{\ell_2}.Y} \mathbin{⫴} \{\} \mathbin{⫴} \underline{Y := \mathsf{b}^{\ell_2}.Y}$

We would like to exclude such cases from being considered well-formed because they are hard to analyse – in particular, they can lead to an infinite state space.

Note the symmetry in the well-formedness conditions on the "global level" of the whole IMC$^{\mathrm{G}}$ expression and on the "local level" of its synchronising processes: we require that in both cases we do not have any free process identifiers and do not hide free names. These two conditions are directly stated for synchronising processes in the rule (7) from Table 6, while for the whole expression we have to check them explicitly, by passing the set of globally free names as parameter and by considering only process identifier closed IMC$^{\mathrm{G}}$ expressions.

"Well-behaved" IMC$^{\mathrm{G}}$ expressions will be called IMC$^{\mathrm{G}}$ *programs*, see Definition 2. Besides closeness and well-formedness, we require that IMC$^{\mathrm{G}}$ programs are uniquely labelled and have unique process identifiers. Unique labelling helps to differentiate between two occurrences of the same action name or the same delay rate: for example, in $\underline{X := \mathsf{a}^{\ell_1}.\mathsf{a}^{\ell_2}.\mathsf{b}^{\ell_3}.X}$ we can differentiate between a^{ℓ_1} and a^{ℓ_2}. The unique labelling of an IMC$^{\mathrm{G}}$ expression is in general not preserved under transitions. For example, $\underline{X := \mathsf{a}^{\ell_1}.(\mathsf{a}^{\ell_2}.X + \mathsf{b}^{\ell_3}.X)}$ is uniquely labelled, while the derived from it in one step expression $\mathsf{a}^{\ell_2}.\underline{X := \mathsf{a}^{\ell_1}.(\mathsf{a}^{\ell_2}.X + \mathsf{b}^{\ell_3}.X)} + \mathsf{b}^{\ell_3}.\underline{X := \mathsf{a}^{\ell_1}.(\mathsf{a}^{\ell_2}.X + \mathsf{b}^{\ell_3}.X)}$ is not. However, these different occurrences of the same label are acceptable because they are the result of copying the same process definition several times.

Definition 2 (IMC$^{\mathrm{G}}$ program)**.** An IMC$^{\mathrm{G}}$ program is an IMC$^{\mathrm{G}}$ expression which is closed, well-formed, uniquely labelled and in which all the process identifiers used in process definitions are different.

In Lemma 3 we state that the well-formedness is preserved under transitions. This shows, first, that the well-formedness is a well-defined concept, and second, that we can use the properties that follow from the well-formedness of some F in the whole LTS induced by the semantics of F – for example, the property that labels of synchronising processes stay disjoint after any number of semantic transitions.

Lemma 3 (Well-formedness)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F *and* $F \xrightarrow{*} E$*, then* $\vdash_{\mathbf{fn}(E)} E$ *holds and* $Labs(E_1) \cap Labs(E_2) = \emptyset$ *for all* $E_1 \mathbin{⫴} A \mathbin{⫴} E_2 \preceq E$.

4. Pathway Analysis

In this section we will present a number of operators on the syntax of IMC$^{\mathrm{G}}$ that together represent so-called Pathway Analysis of IMC$^{\mathrm{G}}$. This is a subtype of Data Flow Analysis which is a well-know method in the field of Static Analysis.

Data Flow Analysis method has been introduced in the field of program analysis as one of the methods for analysing programs' code. The idea is to define for each program's basic block (for example, for each command) a so-called *transfer function* which determines how the program's states (for example, values of the variables) before and after the execution of the basic block are connected. Finally, the overapproximation of the program's states at program points before and after each basic block is computed from the known initial/final program states and the transfer functions. Overapproximation is in particular due to the fact that, as basic blocks can be executed many (even infinitely many) times during the program's execution, transfer functions in general do not describe data transformations caused by them in a precise way.

If all the transfer functions are monotone, than the whole construction is often called Monotone Data Flow Analysis Frameworks, see [5] for details. If the transfer functions cause some pieces of data to become obsolete and others to be added to the current data, then the scheme is called Bitvector Frameworks. The transfer function for a basic block b would then take a simple form

$$f_b(D) = (D - kill_b) + generate_b,$$

where D represents valid data before the execution of the basic block b (e.g. live variables), $kill_b$ represents the information that becomes obsolete/is killed after the execution of the basic block (e.g. variables redefined in b) and $generate_b$ represents newly generated information (e.g. variables used in b), see [7] for details.

Data Flow Analysis in the field of process algebras is applied to the syntax of process algebraic expressions, see [8,10–12]. There are, however, some differences compared to Data Flow Analysis as a program analysis method. In traditional programming languages syntax is static and execution only changes the memory state and the program pointer; hence it is

natural to compute data flow information in terms of memory state approximations that are localised to program points. In process algebra, however, the syntax *is* the state and therefore changes dynamically during execution. Thus, when developing the Data Flow Analysis of process algebras, the goal was to approximate syntactic expressions rather than memory states. In order to do this, a notion of *exposed labels* or *exposed prefixes* has been coined in Nielson and Nielson [10].

Intuitively, exposed labels of a process are those labels that might decorate a transition enabled for this process. For example, for the IMC$^{\mathsf{G}}$ process $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{a}^{\ell_2}.\mathsf{b}^{\ell_3}.X}$ the exposed labels are ℓ_1 and ℓ_2, but not ℓ_3. Thus, for the purposes of the analysis, we shall abstractly characterise system states by their multisets of exposed labels (there could be more than one exposed label of the same kind). We shall then compute the system's behaviour by tracking how these multisets evolve when transitions occur.

A transition may cause some exposed labels to disappear, and others to emerge. Transfer functions are used to express how multisets of exposed labels are changed when transitions decorated by one or more exposed labels are "fired". Much akin to Bitvector Frameworks, transfer functions take the form

$$f_\ell(D) = (D - \mathit{kill}_\ell) + \mathit{generate}_\ell,$$

where D denotes exposed labels of a state and for each label ℓ participating in the transition there exists a transfer function consisting of kill_ℓ and $\mathit{generate}_\ell$ functions [10]. The difference with the previous equality is that the transfer functions are defined for separate labels rather than for the whole program state/basic block.

Exposed labels and transfer functions have been computed by the use of a number of operators on the syntax of a process algebra in the previous work. In our work we have also defined the exposed, kill and generate operators on the syntax of IMC$^{\mathsf{G}}$. In contrast to the previous work, we analyse a process algebra with the CSP-style, i.e. multiway, synchronisation [4]; and in order to account for this, we have introduced a new, so-called *chains* operator, which determines which labels synchronise with each other during the execution. We have also introduced a number of auxiliary operators. In this section we will present all the mentioned operators and show how to model semantic transitions enabled for an IMC$^{\mathsf{G}}$ expression E from the results of these operators on E.

We call our analysis the Pathway Analysis for IMC$^{\mathsf{G}}$. This term has originated in the computational biology and has been used in Nielson et al. [8] to denote the application of Static Analysis to biological systems (in particular, to the calculus of BioAmbients, see [15]) with the purpose of extracting the information on their possible dynamic evolution (on so-called *pathways*). The current work is not explicitly targeted at biological processes. However, the purpose of the analysis (analysing systems' dynamics) and the analysis methods are close to the ones in Nielson et al. [8] etc., therefore we have decided to reuse the name Pathway Analysis for it.

4.1. Exposed labels

The *exposed operator* is denoted $\mathcal{E}$ and is defined inductively on the syntax of IMC$^{\mathsf{G}}$, see Table 7. The idea is that all the labels of the prefixes on the highest syntactic level are exposed, as ℓ_1 in the process $\mathsf{a}^{\ell_1}.\mathsf{b}^{\ell_2}.\mathbf{0}$ (see the rules (1–2) in Table 7). Exposed labels of expressions connected by the choice and parallelisation constructs are added together, see the rules (3) and (5). The operator $\mathcal{E}$ returns an element from the multiset domain $\mathfrak{M}$ in order to account for possibility that several labels of the same type are exposed. We will see later in Section 5 that for an IMC$^{\mathsf{G}}$ program F its exposed labels together with the results computed by the *generate*, *kill* and *chains* operators on F (that will be defined below in Sections 4.2 and 4.3) fully characterise the LTS induced by the semantics of F.

The operator $\mathcal{E}$ makes use of some environment Γ in order to determine exposed labels of process variables, see the rule (7) in Table 7. The parameter Γ is a mapping from process variables to the exposed labels of their process definitions. In the following we will usually assume that Γ has been computed beforehand for all the variables occurring in the argument of $\mathcal{E}$ by, for example, the operator **vl** defined in Table 8. If an input expression of **vl** is closed and each process variable occurs in it only once, then **vl** will return for each process variable the exact result.

Table 7
Operator $\mathcal{E} : \text{IMC}^{\mathsf{G}} \to \mathfrak{M}$ computing exposed labels, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$, $\Gamma \in \mathfrak{V}$.

$$
\begin{aligned}
\mathcal{E}_\Gamma[\![\alpha^\ell.X]\!] &= \bot_{\mathfrak{M}}\ [\ell \mapsto 1] && (1)\\
\mathcal{E}_\Gamma[\![\alpha^\ell.P]\!] &= \bot_{\mathfrak{M}}\ [\ell \mapsto 1] && (2)\\
\mathcal{E}_\Gamma[\![P_1 + P_2]\!] &= \mathcal{E}_\Gamma[\![P_1]\!] + \mathcal{E}_\Gamma[\![P_1]\!] && (3)\\
\mathcal{E}_\Gamma[\![\text{hide } A \text{ in } P]\!] &= \mathcal{E}_\Gamma[\![P]\!] && (4)\\
\mathcal{E}_\Gamma[\![P_1 \between A \between P_2]\!] &= \mathcal{E}_\Gamma[\![P_1]\!] + \mathcal{E}_\Gamma[\![P_2]\!] && (5)\\
\mathcal{E}_\Gamma[\![X := P]\!] &= \mathcal{E}_\Gamma[\![P]\!] && (6)\\
\mathcal{E}_\Gamma[\![X]\!](\ell) &= \Gamma(X)(\ell) && (7)\\
\mathcal{E}_\Gamma[\![\mathbf{0}]\!] &= \bot_{\mathfrak{M}} && (8)
\end{aligned}
$$

Table 8
Operator $\mathbf{vl} : \mathrm{IMC}^{\mathrm{G}} \to \mathfrak{V}$ computing exposed labels of process definitions, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$.

$$
\begin{aligned}
\mathbf{vl}[\![\alpha^{\ell}.X]\!] &= \bot_{\mathfrak{V}} && (1)\\
\mathbf{vl}[\![\alpha^{\ell}.P]\!] &= \mathbf{vl}[\![P]\!] && (2)\\
\mathbf{vl}[\![P_1 + P_2]\!] &= \mathbf{vl}[\![P_1]\!] \sqcup \mathbf{vl}[\![P_2]\!] && (3)\\
\mathbf{vl}[\![\text{hide } A \text{ in } P]\!] &= \mathbf{vl}[\![P]\!] && (4)\\
\mathbf{vl}[\![P_1 \between A \between P_2]\!] &= \mathbf{vl}[\![P_1]\!] \sqcup \mathbf{vl}[\![P_2]\!] && (5)\\
\mathbf{vl}[\![\underline{X := P}]\!] &= \bot_{\mathfrak{V}}\left[X \mapsto \mathcal{E}_{\bot_{\mathfrak{V}}}[\![\mathbf{P}]\!]\right] \sqcup \mathbf{vl}[\![P]\!] && (6)\\
\mathbf{vl}[\![X]\!] &= \bot_{\mathfrak{V}} && (7)\\
\mathbf{vl}[\![\mathbf{0}]\!] &= \bot_{\mathfrak{V}} && (8)
\end{aligned}
$$

The main rule in defining **vl** is the rule (6). We use there the exposed operator with an empty environment on the body of a process definition. The empty environment can be justified with the help of Lemma 4 which states that the environment is not important in this case, so we can parametrise the exposed operator with an empty environment for simplicity. It is also stated that substitutions in $\mathrm{IMC}^{\mathrm{G}}$ expressions do not change their exposed labels. The lemma is basically due to the guardedness of the syntax of $\mathrm{IMC}^{\mathrm{G}}$ and will be used in many proofs of the lemmas below.

Lemma 4 (Exposed labels). *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *expression* E*, then* $\mathcal{E}_{\Gamma}[\![E]\!] = \mathcal{E}_{\bot_{\mathfrak{V}}}[\![E]\!] = \mathcal{E}_{\bot_{\mathfrak{V}}}[\![E\{X/E'\}]\!]$ *holds for all* $\Gamma \in \mathfrak{V}$ *and* $E' \in \mathrm{IMC}^{\mathrm{G}}$.

Lemma 5 states that the results returned by the operator **vl** on $\mathrm{IMC}^{\mathrm{G}}$ programs are in a sense invariant under semantic transitions. This will be used in the considerations concerning properties of $\mathrm{IMC}^{\mathrm{G}}$ programs that remain stable under any number of transitions. We will prove many similar results (i.e. that some property is stable under transitions) in the following.

Lemma 5 (Variable definitions under transitions). *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F *and* $F \xrightarrow{*} E$*, then for all* $\underline{X := E''} \preceq E$ *holds* $\mathbf{vl}[\![\underline{X := E''}]\!](X) = \mathbf{vl}[\![F]\!](X)$.

Note that, in case $F \xrightarrow{*} E$, then it is not necessarily strictly $\mathbf{vl}[\![F]\!] = \mathbf{vl}[\![E]\!]$, because some of the process definitions may appear in F but not E. For example, $\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{a}^{\ell_2}.Y} \xrightarrow[\bot_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X}$, and the definition of Y has "disappeared" from the process on the right. However, all the processes variables present in E will have the same exposed labels as in F.

We will now state in Lemma 6 that for $\mathrm{IMC}^{\mathrm{G}}$ programs all exposed labels stay unique after any number of semantic steps, i.e. exposed labels constitute a set. This is a necessary condition in order to model the semantics by our methods in a precise way. Note that we can use the same environment Γ in Lemma 6 both for F and for E'', because, according to Lemma 5, process definitions are stable under transitions. Also note that, according to Lemma 6, different transition derivation trees are guaranteed to correspond to different multisets decorating the transitions. This result means that for $\mathrm{IMC}^{\mathrm{G}}$ programs it is not necessary to have multisets of delay transitions as it was mentioned in Section 2.2, i.e. every such multiset would contain no more than one transition.

Lemma 6 (Exposed labels are a set). *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* F*,* $\Gamma = \mathbf{vl}[\![F]\!]$*,* $F \xrightarrow{*} E$ *and* $E'' \preceq E$*, then* $\mathcal{E}_{\Gamma}[\![E'']\!](\ell) \in \{0, 1\}$ *for all* $\ell \in \mathbf{Lab}$*. Moreover, for all transition pairs* $E \xrightarrow[C_1]{} G_1$ *and* $E \xrightarrow[C_2]{} G_2$ *such that their transition derivation trees are different* ($G_1 = G_2$ *is possible) holds* $C_1 \neq C_2$.

4.2. Chains operator

It is not enough for a label to be exposed in order to be "executable". It might lack synchronisation partners and therefore be not executable on its own. For example, ℓ_1 is not executable in $\underline{X := \mathsf{a}^{\ell_1}.X} \between \{\mathsf{a}\} \between \underline{Y := \mathsf{b}^{\ell_2}.\mathsf{a}^{\ell_3}.Y}$, because it can only be executed together with ℓ_3. We will say in the following that ℓ_1 and ℓ_3 constitute a *chain*. Before presenting the chains operator which computes all the chains in its input expression, we will discuss the correspondence between labels and names. This is important during the computation of chains because only labels with the same action name, that has moreover not been internalised, synchronise with each other.

For a uniquely labelled $\mathrm{IMC}^{\mathrm{G}}$ expression the correspondence between labels and action names or delay rates is uniquely defined. The mapping from labels to names can be computed by the operator **ln** in Table 9.

Internalised action names cannot become external again after any number of transitions. On the other hand, external action names can in general be hidden by applying the hide-construct on top of them after a number of transitions (as a

Table 9
Operator $\mathbf{ln} : \mathrm{IMC}^{\mathrm{G}} \to \mathfrak{L}$ computing a mapping between labels and their corresponding names, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$.

$$
\begin{array}{rcll}
\mathbf{ln}[\![\alpha^{\ell}.X]\!] & = & \bot_A[\ell \mapsto \alpha] & (1)\\
\mathbf{ln}[\![\alpha^{\ell}.P]\!] & = & \bot_A[\ell \mapsto \alpha] \sqcup \mathbf{ln}[\![P]\!] & (2)\\
\mathbf{ln}[\![P_1 + P_2]\!] & = & \mathbf{ln}[\![P_1]\!] \sqcup \mathbf{ln}[\![P_2]\!] & (3)\\
\mathbf{ln}[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathbf{ln}[\![P]\!] & (4)\\
\mathbf{ln}[\![P_1 \shuffle A \shuffle P_2]\!] & = & \mathbf{ln}[\![P_1]\!] \sqcup \mathbf{ln}[\![P_2]\!] & (5)\\
\mathbf{ln}[\![\underline{X := P}]\!] & = & \mathbf{ln}[\![P]\!] & (6)\\
\mathbf{ln}[\![\underline{X}]\!] & = & \bot_A & (7)\\
\mathbf{ln}[\![\mathbf{0}]\!] & = & \bot_A & (8)
\end{array}
$$

result of recursion unfoldings). However, Lemma 7 states that for well-formed $\mathrm{IMC}^{\mathrm{G}}$ expressions this does not occur: free action names (that are forwarded as a parameter to the well-formedness condition) stay free.

Lemma 7 (Hidden labels). *Given a well-formed* $\mathrm{IMC}^{\mathrm{G}}$ *expression* F, $\Lambda = \mathbf{ln}[\![F]\!]$ *and* $F \xrightarrow{*} E$, *then for all* $\ell \in \mathit{Labs}(E)$ *holds:* $\Lambda(\ell) \in \mathbf{fn}(E)$ *iff* $\Lambda(\ell) \in \mathbf{fn}(F)$.

The *chains* operator $\mathfrak{T}$ is presented in Table 10. The rules are straightforward, and the main logic is in the rule (5). There we take all possible combinations of chains from two parallel processes that synchronise on the same external action which is free in both processes. The operator $\mathfrak{T}$ makes use of the environment Λ which is a mapping from labels to their corresponding names (can be previously computed by the operator **ln**) and of the function $\mathbf{name}^{\mathbf{h}}$ defined in Table 2. The last returns the name for a multiset taking into account possible action hindings. It is clear from the construction that all the chains returned by $\mathfrak{T}$ have the same action name or delay rate, and only chains that correspond to actions can have more than one label in their domain.

Knowing chains and exposed labels of an $\mathrm{IMC}^{\mathrm{G}}$ expression E (for example, derived from some $\mathrm{IMC}^{\mathrm{G}}$ program, so E is guaranteed to have nice properties) is enough to predict all the transitions from E. This is stated in Lemma 9 below. For example, labels ℓ_1 and ℓ_3 are exposed in $\underline{X := \mathsf{a}^{\ell_1}.X} \shuffle \{\mathsf{a}\} \shuffle \mathsf{a}^{\ell_3}.\underline{Y := \mathsf{b}^{\ell_2}.\mathsf{a}^{\ell_3}.Y}$, and there is a chain containing both these labels. Lemma 9 would predict that there is an enabled transition decorated by these two labels, and this is indeed the case.

In order to prove Lemma 9 for transitions involving recursion unfolding, we need an additional fact about how chains evolve in case a substitution has been made. This is the subject of Lemma 8 which states that for well-formed expressions we just take a union of chains of the expression in which the substitution has been made and of the expression which has been inserted.

Lemma 8 (Chains under substitution). *Given well-formed* $\mathrm{IMC}^{\mathrm{G}}$ *expressions* $E' \preceq E$, $X \in \mathbf{fpi}(E')$, $\mathbf{ln}[\![E']\!] \leq \Lambda$ *and* $\mathbf{ln}[\![E]\!] \leq \Lambda$, *then it holds that* $\mathfrak{T}_\Lambda[\![E'\{X/E\}]\!] = \mathfrak{T}_\Lambda[\![E']\!] \cup \mathfrak{T}_\Lambda[\![E]\!]$ *and* $\mathfrak{T}_\Lambda[\![E']\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!]$.

Table 10
Operator $\mathfrak{T} : \mathrm{IMC}^{\mathrm{G}} \to 2^{\mathfrak{M}}$ computing so-called "chains", i.e. labels that can be executed only together, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$, $\Lambda \in \mathfrak{L}$.

$$
\begin{array}{rcll}
\mathfrak{T}_\Lambda[\![\mathsf{a}^{\ell}.X]\!] & = & \{\bot_{\mathfrak{M}}\ [\ell \mapsto 1]\} & (1)\\
\mathfrak{T}_\Lambda[\![\mathsf{a}^{\ell}.P]\!] & = & \{\bot_{\mathfrak{M}}\ [\ell \mapsto 1]\} \cup \mathfrak{T}_\Lambda[\![P]\!] & (2)\\
\mathfrak{T}_\Lambda[\![P_1 + P_2]\!] & = & \mathfrak{T}_\Lambda[\![P_1]\!] \cup \mathfrak{T}_\Lambda[\![P_2]\!] & (3)\\
\mathfrak{T}_\Lambda[\![\mathsf{hide}\ A\ \mathsf{in}\ P]\!] & = & \mathfrak{T}_\Lambda[\![P]\!] & (4)\\
\mathfrak{T}_\Lambda[\![P_1 \shuffle A \shuffle P_2]\!] & = & \{C | C \in \mathfrak{T}_\Lambda[\![P_1]\!], \mathbf{name}^{\mathbf{h}}{}_{\Lambda,\mathbf{fn}(P_1)}(C) \notin A\} \cup & \\
 & & \{C | C \in \mathfrak{T}_\Lambda[\![P_2]\!], \mathbf{name}^{\mathbf{h}}{}_{\Lambda,\mathbf{fn}(P_2)}(C) \notin A\} \cup & \\
 & & \bigcup_{\alpha \in A}\{C_1 + C_2 | C_1 \in \mathfrak{T}_\Lambda[\![P_1]\!], & \\
 & & \quad C_2 \in \mathfrak{T}_\Lambda[\![P_1]\!], & \\
 & & \quad \mathbf{name}^{\mathbf{h}}{}_{\Lambda,\mathbf{fn}(P_1)}(C_1) = \alpha, & \\
 & & \quad \mathbf{name}^{\mathbf{h}}{}_{\Lambda,\mathbf{fn}(P_2)}(C_2) = \alpha\} & (5)\\
\mathfrak{T}_\Lambda[\![\underline{X := P}]\!] & = & \mathfrak{T}_\Lambda[\![P]\!] & (6)\\
\mathfrak{T}_\Lambda[\![\underline{X}]\!] & = & \emptyset & (7)\\
\mathfrak{T}_\Lambda[\![\mathbf{0}]\!] & = & \emptyset & (8)
\end{array}
$$

Lemma 9 (Transition existence). *Given an* IMC$^{\mathsf{G}}$ *program* F, $F \xrightarrow{*} E$, $\Lambda = \mathbf{ln}[\![E]\!]$ *and* $\Gamma = \mathbf{vl}[\![E]\!]$, *then* $E \xrightarrow[C]{\alpha} E'$ *iff* $C \in \mathfrak{T}_\Lambda[\![E]\!]$, $C \leq \mathcal{E}_\Gamma[\![E]\!]$, $\alpha = \mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C)$, *and, in case* $\alpha \in \boldsymbol{Rate}$, *there is no* C' *such that* $C' \in \mathfrak{T}_\Lambda[\![E]\!]$, $C' \leq \mathcal{E}_\Gamma[\![E]\!]$ *and* $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C') = \tau$.

In Section 4.3 we will show how to compute exposed labels of expressions into which an IMC$^{\mathsf{G}}$ expression E evolves as a result of transitions predicted for E by Lemma 9.

4.3. Generate and kill operators

After a transition has occurred, some of the previously exposed labels cease to be exposed (are *killed*) and others become exposed (are *generated*). For example, labels ℓ_1 and ℓ_2 are exposed in the expression $\mathsf{a}^{\ell_1}.\mathbf{0} + \mathsf{b}^{\ell_2}.\mathsf{c}^{\ell_3}.\mathbf{0}$. After the transition $\mathsf{a}^{\ell_1}.\mathbf{0} + \mathsf{b}^{\ell_2}.\mathsf{c}^{\ell_3}.\mathbf{0} \longrightarrow \mathsf{c}^{\ell_3}.\mathbf{0}$ the labels ℓ_1 and ℓ_2 are killed and the label ℓ_3 is generated.

We associate kill and generate effects with separate labels instead of states. The generate ($\mathcal{G}$) and kill ($\mathcal{K}$) operators are defined by induction on the syntax of IMC$^{\mathsf{G}}$ in accordingly Tables 12 and 14. A label kills all the labels in the choice construct on its own syntactic level (the rules (1–3) in Table 14) and generates all the labels of the prefixes on the lower syntactic level (the rules (1–2) in Table 12). In the above example the labels ℓ_1 and ℓ_2 kill two choice alternatives (i.e. both ℓ_1 and ℓ_2), while only ℓ_2 generates ℓ_3. The generate operator makes use of the mapping Γ from process variables to the exposed labels of their process definitions.

Herewith we would like to put forward two remarks on the differences of our approach from the previous work. First, we assume that the input expression is uniquely labelled, therefore it is enough to take the least upper bounds while computing both the $\mathcal{G}$ and $\mathcal{K}$ operators – each label occurs in the input expression only once anyway. In the previous work the greatest lower bound was taken for $\mathcal{K}$ in order to assign to each label the least killed multiset. Second, in order to simplify the proofs on the properties of the $\mathcal{G}$ and $\mathcal{K}$ operators, we have defined two additional operators – $\mathcal{G}'$ and $\mathcal{K}'$, defined in Tables 11 and 13 – that compute the generate and kill effects only for the prefixes on the top syntactic level. It is clear by comparing the rules for $\mathcal{G}'$ and $\mathcal{G}$ and for $\mathcal{K}'$ and $\mathcal{K}$ that the only difference is in the rule (2).

Lemma 10 states that the $\mathcal{G}'$ and $\mathcal{K}'$ operators on well-formed IMC$^{\mathsf{G}}$ expressions before and after some variable substitution return the same results (for $\mathcal{G}'$ the well-formedness properties are used in order to prove the lemma). This lemma will be useful for reasoning on the effects of a transition in Lemma 11: a substitution is conducted in the SOS rules (9) and (16) in Table 3 during recursion unfolding.

Lemma 10 (Kill/generate under substitution). *Given well-formed* IMC$^{\mathsf{G}}$ *expressions* $E' \preceq E$, $X \in \mathbf{fpi}(E')$, $\Gamma(X) = \mathcal{E}_{\perp_{\mathfrak{V}}}[\![\mathbf{E}]\!]$, *then* $\mathcal{G}'_\Gamma[\![E'\{X/E\}]\!] = \mathcal{G}'_\Gamma[\![E']\!]$ *and* $\mathcal{K}'[\![E'\{X/E\}]\!] = \mathcal{K}'[\![E']\!]$ *hold.*

Table 11
Operator $\mathcal{G}' : \text{IMC}^{\mathsf{G}} \to \mathfrak{N}$ computing labels generated by the labels exposed on the top syntactic level, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$, $\Gamma \in \mathfrak{V}$.

$$
\begin{array}{rcll}
\mathcal{G}'_\Gamma[\![\alpha^\ell.X]\!] & = & \perp_{\mathfrak{N}}[\ell \mapsto \mathcal{E}_\Gamma[\![X]\!]] & (1)\\
\mathcal{G}'_\Gamma[\![\alpha^\ell.P]\!] & = & \perp_{\mathfrak{N}}[\ell \mapsto \mathcal{E}_\Gamma[\![P]\!]] & (2)\\
\mathcal{G}'_\Gamma[\![P_1 + P_2]\!] & = & \mathcal{G}'_\Gamma[\![P_1]\!] \sqcup \mathcal{G}'_\Gamma[\![P_2]\!] & (3)\\
\mathcal{G}'_\Gamma[\![\text{hide } A \text{ in } P]\!] & = & \mathcal{G}'_\Gamma[\![P]\!] & (4)\\
\mathcal{G}'_\Gamma[\![P_1 \shuffle A \shuffle P_2]\!] & = & \mathcal{G}'_\Gamma[\![P_1]\!] \sqcup \mathcal{G}'_\Gamma[\![P_2]\!] & (5)\\
\mathcal{G}'_\Gamma[\![X := P]\!] & = & \mathcal{G}'_\Gamma[\![P]\!] & (6)\\
\mathcal{G}'_\Gamma[\![X]\!] & = & \perp_{\mathfrak{N}} & (7)\\
\mathcal{G}'_\Gamma[\![\mathbf{0}]\!] & = & \perp_{\mathfrak{N}} & (8)
\end{array}
$$

Table 12
Operator $\mathcal{G} : \text{IMC}^{\mathsf{G}} \to \mathfrak{N}$ computing generated labels, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$, $\Gamma \in \mathfrak{V}$.

$$
\begin{array}{rcll}
\mathcal{G}_\Gamma[\![\alpha^\ell.X]\!] & = & \perp_{\mathfrak{N}}[\ell \mapsto \mathcal{E}_\Gamma[\![X]\!]] & (1)\\
\mathcal{G}_\Gamma[\![\alpha^\ell.P]\!] & = & \perp_{\mathfrak{N}}[\ell \mapsto \mathcal{E}_\Gamma[\![P]\!]] \sqcup \mathcal{G}_\Gamma[\![P]\!] & (2)\\
\mathcal{G}_\Gamma[\![P_1 + P_2]\!] & = & \mathcal{G}_\Gamma[\![P_1]\!] \sqcup \mathcal{G}_\Gamma[\![P_2]\!] & (3)\\
\mathcal{G}_\Gamma[\![\text{hide } A \text{ in } P]\!] & = & \mathcal{G}_\Gamma[\![P]\!] & (4)\\
\mathcal{G}_\Gamma[\![P_1 \shuffle A \shuffle P_2]\!] & = & \mathcal{G}_\Gamma[\![P_1]\!] \sqcup \mathcal{G}_\Gamma[\![P_2]\!] & (5)\\
\mathcal{G}_\Gamma[\![X := P]\!] & = & \mathcal{G}_\Gamma[\![P]\!] & (6)\\
\mathcal{G}_\Gamma[\![X]\!] & = & \perp_{\mathfrak{N}} & (7)\\
\mathcal{G}_\Gamma[\![\mathbf{0}]\!] & = & \perp_{\mathfrak{N}} & (8)
\end{array}
$$

Table 13
Operator $\mathcal{K}' : \mathrm{IMC}^{\mathrm{G}} \to \mathfrak{N}$ computing labels killed by the labels exposed on the top syntactic level, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$.

$$
\begin{array}{rcll}
\mathcal{K}'[\![\alpha^{\ell}.X]\!] & = & \bot_{\mathfrak{N}}[\ell \mapsto \bot_{\mathfrak{M}}[\ell \mapsto 1]] & (1)\\
\mathcal{K}'[\![\alpha^{\ell}.P]\!] & = & \bot_{\mathfrak{N}}[\ell \mapsto \bot_{\mathfrak{M}}[\ell \mapsto 1]] & (2)\\
\mathcal{K}'[\![P_1 + P_2]\!](\ell) & = & \mathcal{K}'[\![P_1]\!](\ell) + \mathcal{K}'[\![P_2]\!](\ell) & (3a)\\
 & & \text{if } \ell \in \mathcal{E}_{\bot_{\mathfrak{N}}}[\![\mathbf{P_1} + \mathbf{P_2}]\!] & \\
\mathcal{K}'[\![P_1 + P_2]\!](\ell) & = & \bot_{\mathfrak{M}} & (3b)\\
 & & \text{otherwise} & \\
\mathcal{K}'[\![\text{hide } A \text{ in } P]\!] & = & \mathcal{K}'[\![P]\!] & (4)\\
\mathcal{K}'[\![P_1 \between A \between P_2]\!] & = & \mathcal{K}'[\![P_1]\!] \sqcup \mathcal{K}'[\![P_2]\!] & (5)\\
\mathcal{K}'[\![\underline{X := P}]\!] & = & \mathcal{K}'[\![P]\!] & (6)\\
\mathcal{K}'[\![\underline{X}]\!] & = & \bot_{\mathfrak{N}} & (7)\\
\mathcal{K}'[\![\mathbf{0}]\!] & = & \bot_{\mathfrak{N}} & (8)
\end{array}
$$

Table 14
Operator $\mathcal{K} : \mathrm{IMC}^{\mathrm{G}} \to \mathfrak{N}$ computing killed labels, with $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$.

$$
\begin{array}{rcll}
\mathcal{K}[\![\alpha^{\ell}.X]\!] & = & \bot_{\mathfrak{N}}[\ell \mapsto \bot_{\mathfrak{M}}[\ell \mapsto 1]] & (1)\\
\mathcal{K}[\![\alpha^{\ell}.P]\!] & = & \bot_{\mathfrak{N}}[\ell \mapsto \bot_{\mathfrak{M}}[\ell \mapsto 1]] \sqcup \mathcal{K}[\![P]\!] & (2)\\
\mathcal{K}[\![P_1 + P_2]\!](\ell) & = & \mathcal{K}[\![P_1]\!](\ell) + \mathcal{K}[\![P_2]\!](\ell) & (3a)\\
 & & \text{if } \ell \in \mathcal{E}_{\bot_{\mathfrak{N}}}[\![\mathbf{P_1} + \mathbf{P_2}]\!] & \\
\mathcal{K}[\![P_1 + P_2]\!](\ell) & = & \mathcal{K}[\![P_1]\!](\ell) \sqcup \mathcal{K}[\![P_2]\!](\ell) & (3b)\\
 & & \text{otherwise} & \\
\mathcal{K}[\![\text{hide } A \text{ in } P]\!] & = & \mathcal{K}[\![P]\!] & (4)\\
\mathcal{K}[\![P_1 \between A \between P_2]\!] & = & \mathcal{K}[\![P_1]\!] \sqcup \mathcal{K}[\![P_2]\!] & (5)\\
\mathcal{K}[\![\underline{X := P}]\!] & = & \mathcal{K}[\![P]\!] & (6)\\
\mathcal{K}[\![\underline{X}]\!] & = & \bot_{\mathfrak{N}} & (7)\\
\mathcal{K}[\![\mathbf{0}]\!] & = & \bot_{\mathfrak{N}} & (8)
\end{array}
$$

Lemma 11 captures the essence of the generate and kill operators and predicts the outcome (in the sense of exposed labels) of a transition enabled for some E if $\mathcal{E}$, $\mathcal{G}'$ and $\mathcal{K}'$ have been previously computed on E. Note that there can only be one exposed label of each kind in E according to Lemma 6, therefore it is enough to account for labels in the domain of the chain only once. In the previous work the labelling was in general non-unique, and instead of the equality relation there was smaller or equal relation between the actual outcome of a transition and the predicted one, see, for example, [10].

Lemma 11 (Kill/generate). *Given an* $\mathrm{IMC}^{\mathrm{G}}$ *program* $F, \Gamma = \mathbf{vl}[\![F]\!]$ *and* $F \xrightarrow{*} E \xrightarrow[C]{\alpha} E'$, *then* $\mathcal{E}_{\Gamma}[\![E']\!] = \mathcal{E}_{\Gamma}[\![E]\!] - \sum_{C(\ell)>0} \mathcal{K}'[\![E]\!](\ell) + \sum_{C(\ell)>0} \mathcal{G}'_{\Gamma}[\![E]\!](\ell)$.

5. Pathway Analysis is precise

In this section we will prove several results about the Pathway Analysis of $\mathrm{IMC}^{\mathrm{G}}$ programs. These results will be enough in order to prove that the Pathway Analysis captures all the semantic properties of $\mathrm{IMC}^{\mathrm{G}}$ programs, avoiding any imprecision stemming from the analysis technique. To illustrate this, we will show how to build LTSs strongly bisimilar to the LTSs induced by the semantics of analysed $\mathrm{IMC}^{\mathrm{G}}$ programs in Section 5.2.

5.1. Pathway Analysis results are stable under transitions

In this section we will prove that the results of the generate, kill and chains operators on an $\mathrm{IMC}^{\mathrm{G}}$ program F are mostly transferable to all the states reachable from F. The word "mostly" refers to the fact that some of the behaviour possible in the original state of F can cease to be possible after a number of transitions. However, if some label still occurs in an $\mathrm{IMC}^{\mathrm{G}}$ expression E reachable from F, then its "behaviour" (i.e. chains in which it participates, the generate and kill effects) are essentially the same as in F.

In particular, if for some chain C in $\mathfrak{T}_{\Lambda}[\![E]\!]$ all the constituting labels are in E, then the chain C is also the chain of E. On the other hand, all the chains in $\mathfrak{T}_{\Lambda}[\![E]\!]$ are also the chains of F. This is stated in Lemma 13. In the proof of this lemma we use the auxiliary Lemma 12 which shows that chains of subexpressions are also chains of larger $\mathrm{IMC}^{\mathrm{G}}$ expressions if there is no synchronisation involved.

Note that some chains may be "lost" after a transition. For example, for $\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathsf{a}^{\ell_2}.\mathbf{0} \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathbf{0}$ we compute $\mathfrak{T}_\Lambda[\![\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathsf{a}^{\ell_2}.\mathbf{0}]\!] = \{\perp_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]\}$ but $\mathfrak{T}_\Lambda[\![\underline{X := \mathsf{a}^{\ell_1}.X} \parallel \mathsf{a} \parallel \mathbf{0}]\!] = \emptyset$. This illustrates the statement of Lemma 12 that no new chains are created after transitions, while those lacking the labels are lost.

Lemma 12 (Chains of subexpressions). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Lambda = \mathbf{ln}[\![F]\!]$, $F \xrightarrow{*} E$ *and* $E'' \preceq E' \preceq E$ *such that no sychronisation construct is applied to* E'' *in* E', *then for all* C *such that* $dom(C) \subseteq Labs(E'')$ *holds:* $C \in \mathfrak{T}_\Lambda[\![E'']\!]$ *iff* $C \in \mathfrak{T}_\Lambda[\![E']\!]$.

Lemma 13 (Chains under transitions). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Lambda = \mathbf{ln}[\![F]\!]$, $F \xrightarrow{*} E$ *and* $\mathbf{dom}(C) \subseteq Labs(E)$, *then* $C \in \mathfrak{T}_\Lambda[\![F]\!]$ *iff* $C \in \mathfrak{T}_\Lambda[\![E]\!]$ *and* $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C) = \mathbf{name^h}_{\Lambda,\mathbf{fn}(E)}(C)$.

In Lemma 14 we state that the generate functions of labels that occur in some E reachable from some IMC$^{\mathsf{G}}$ program F are the same both in F and E (in Lemma 14 we use $\mathcal{G}'$ and $\mathcal{K}'$ on E instead of $\mathcal{G}$ and $\mathcal{K}$ in order to simplify the reasoning in case a label occurs in E several times). This is obviously not the case for the labels occurring in F but not in E. For example, for $\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y} \xrightarrow[\perp_{\mathfrak{M}}[\ell_1 \mapsto 1]]{\mathsf{a}} \underline{X := \mathsf{a}^{\ell_1}.X}$ we have $\mathcal{G}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}]\!](\ell_1) = \mathcal{G}'_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X}]\!](\ell_1) = \perp_{\mathfrak{M}}[\ell_1 \mapsto 1]$, but $\mathcal{G}_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}]\!](\ell_2) = \perp_{\mathfrak{M}}[\ell_2 \mapsto 1] \neq \perp_{\mathfrak{M}} = \mathcal{G}'_\Gamma[\![\underline{X := \mathsf{a}^{\ell_1}.X}]\!](\ell_2)$.

For the kill operator, the killed multisets can be strictly smaller after a number of transitions. In the above example, $\mathcal{K}[\![\underline{X := \mathsf{a}^{\ell_1}.X} + \underline{Y := \mathsf{b}^{\ell_2}.Y}]\!](\ell_1) = \perp_{\mathfrak{M}}[\ell_1 \mapsto 1, \ell_2 \mapsto 1]$ but $\mathcal{K}'[\![\underline{X := \mathsf{a}^{\ell_1}.X}]\!](\ell_1) = \perp_{\mathfrak{M}}[\ell_1 \mapsto 1]$. We show, however, in Lemma 14 that the kill operator results are equivalent when constrained to the exposed labels of any reachable state. In this way the results of the kill operator are also invariant under transitions, because $\mathcal{E}_\Gamma[\![E'']\!] - \mathcal{K}'[\![E'']\!](\ell) = \mathcal{E}_\Gamma[\![E'']\!] - \mathcal{K}[\![F]\!](\ell)$ for $\ell \in \mathbf{dom}(\mathcal{E}_\Gamma[\![E'']\!])$, with F and E'' as in Lemma 14.

Lemma 14 ([Kill/generate under transitions). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Gamma = \mathbf{vl}[\![F]\!]$ *and* $F \xrightarrow{*} E$, *then for all* $E'' \preceq E$, $\ell \in \mathbf{dom}(\mathcal{E}_\Gamma[\![E'']\!])$ *holds:* $\mathcal{G}'_\Gamma[\![E'']\!](\ell) = \mathcal{G}_\Gamma[\![F]\!](\ell)$ *and* $\mathcal{K}'[\![E'']\!](\ell) = \mathcal{K}[\![F]\!](\ell) \sqcap \mathcal{E}_\Gamma[\![E'']\!]$.

5.2. Worklist algorithm

In the previous work (see, for example, the Data Flow Analysis for CCS in Nielson and Nielson [10]) the analysis constructions have been completed by devising the so-called Worklist algorithm. The algorithm would build a finite LTS based on the analysis results that could simulate the LTS of a process algebraic expression for which the analysis has been conducted. In Table 15 we present a similar algorithm which we have named *buildLTS* that would complete our Pathway Analysis. The difference with the previous work is, in particular, that no states are merged during the construction, therefore no so-called granularity function is made use of as in Nielson and Nielson [10].

The algorithm *buildLTS* accepts as input an IMC$^{\mathsf{G}}$ program F. It computes a number of the Pathway Analysis operators on it and, starting from the state characterised by the exposed labels of F, computes all the states reachable according to the results of the operators $\mathcal{G}$, $\mathcal{K}$ and $\mathfrak{T}$ on F (returned in the set *States*) and all the transitions between the states (returned in the set *Transitions*). The resulting LTS is strongly bisimilar to the LTS induced by the semantics of F, and the algorithm is guaranteed to terminate – this is stated in Lemma 15.

The definition of the strong bisimulation on IMC$^{\mathsf{G}}$ is given in Definition 3. This is essentially the same definition as the definition in Hermanns [3] of the strong bisimulation on IMC systems, just with transitions decorated by multisets of labels. Note that in the condition 2 the second sum is taken over all derivable delay transitions, i.e. over all derivation trees. However, in case each delay transition has a unique multiset decorating the transition (as it is the case for all the states reachable from some IMC$^{\mathsf{G}}$ program according to Lemma 6), then the sum can be taken over all the chains decorating the transitions. This consideration will be used in the proof of Lemma 15.

Definition 3 (Bisimulation). An equivalence relation $\mathcal{R}$ on IMC$^{\mathsf{G}}$ expressions with unique exposed labels is a strong bisimulation relation if for all $(E_1, E_2) \in \mathcal{R}$ holds:

- if $E_1 \xrightarrow[C_1]{\alpha} E'_1$ for some $\alpha \in \mathbf{Act} \cup \{\tau\}$ then there exist E'_2 and C_2 such that $E_2 \xrightarrow[C_2]{\alpha} E'_2$ and $(E'_1, E'_2) \in \mathcal{R}$;
- if $E_1 \overset{\tau}{\nrightarrow}$ then $\sum_{E \in S} \sum_{\{E_1 \xrightarrow[C]{\lambda} E \mid \lambda \in \mathbf{Rate}\}} \lambda = \sum_{E \in S} \sum_{\{E_2 \xrightarrow[C]{\lambda} E \mid \lambda \in \mathbf{Rate}\}} \lambda$ for all equivalence classes $S \in \mathrm{IMC}^{\mathsf{G}}/\mathcal{R}$.

We denote $E_1 \sim E_2$ if there exists a strong bisimulation relation $\mathcal{R}$ such that $(E_1, E_2) \in \mathcal{R}$.

Table 15
The algorithm *buildLTS* taking as argument an IMC$^{\mathsf{G}}$ program F, making use of the procedures *enabledChains* (argument in $\mathfrak{M}$) and *move* (arguments in $\mathfrak{M}$).

proc *buildLTS(F)* is
1: $\Gamma := \mathbf{vl}[\![F]\!]$; $\Lambda := \mathbf{ln}[\![F]\!]$; $G := \mathcal{G}_\Gamma[\![F]\!]$; $K := \mathcal{K}[\![F]\!]$; $T := \mathfrak{T}_\Lambda[\![F]\!]$;
2: $M := \mathcal{E}_\Gamma[\![F]\!]$; *Worklist* $:= \{M\}$; *States* $:= \{M\}$; *Transitions* $:= \emptyset$;
3: while *(Worklist* $\neq \emptyset$*)* do
4: $M :=$ *getElement(Worklist)*; *Worklist:=Worklist* $- \{M\}$;
5: $T :=$ *enabledChains*(M);
6: for all $(C \in T)$ do
7: $M' :=$ *move*(M, C);
8: if $(M' \notin$ *States*$)$ then
9: *States:=States* $\cup \{M'\}$;
10: *Worklist:=Worklist* $\cup \{M'\}$;
11: *Transitions:=Transitions* $\cup \{(M, \mathbf{name}^{\mathbf{h}}{}_{\Lambda,A}(C), C, M')\}$;
12: return *(States,Transitions)*

proc *enabledChains*(M) is
13: *SC*:=$\{C \in T | C \leq M\}$;
14: if $(\exists C \in CS$ such that $\mathbf{name}_\Lambda(C) \in \mathbf{Rate})$ then
15: $CS := \{C \in CS | \mathbf{name}^{\mathbf{h}}{}_{\Lambda,A}(C) \neq \tau\}$;
16: return *SC*

proc *move*(M, C) is
17: return $M - (\sum_{\ell \in \mathbf{dom}(C)} K(\ell)) + \sum_{\ell \in \mathbf{dom}(C)} G(\ell)$

The LTS constructed by the algorithm *buildLTS* can in general be smaller than the LTS induced by the semantics of F. The "gain" in the system's size will be, however, not significant for most of IMC$^{\mathsf{G}}$ systems because it is only due to copies of process definitions which have been created during recursion unfolding (i.e. states with one or more copies of process definitions correspond to only one state in the LTS returned by *buildLTS*). For example, two following expressions $\underline{Y := \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.Y}}$ and $\underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.\underline{Y := \underline{X := \mathsf{a}^{\ell_1}.X + \mathsf{b}^{\ell_2}.Y}}}$ have the same exposed labels and the same behaviour (the second is actually derivable from the first after one transition) but they are syntactically different. The Worklist algorithm in Table 15 will map them to the same state in the constructed LTS. In general, the presented algorithm is not a new algorithm for computing bisimulations on IMC systems (see [3] for such an algorithm) but rather a proof that the Pathway Analysis captures all the properties of analysed systems.

Theorem 15 (Worklist algorithm)**.** *Given an* IMC$^{\mathsf{G}}$ *program F, then the Worklist algorithm terminates on F and constructs the LTS, whose initial state $\mathcal{E}_\Gamma[\![F]\!]$ with $\Gamma = \mathbf{vl}[\![F]\!]$ is strongly bisimilar to the LTS induced by the semantics of F, i.e. $F \sim \mathcal{E}_\Gamma[\![F]\!]$.*

6. Conclusions

In this paper we have applied Static Analysis methods to process algebraic expressions. In particular, we have adapted the Data Flow Analysis for process algebras (devised for the first time in Nielson and Nielson [10]) to a stochastic process algebra – the algebra of Interactive Markov Chains. We have introduced a guarded variant of IMC, i.e. IMC$^{\mathsf{G}}$, with a more permissive syntax than IMC, and have devised the so-called well-formedness conditions. These conditions are quite restrictive: we cannot, for example, model a process duplicating itself. Nevertheless, the subclass of well-formed IMC$^{\mathsf{G}}$ still allows to model many interesting concurrent systems – for example, those with a bounded number of newly created processes.

A number of Pathway Analysis operators, i.e. the exposed, generate and kill operators, have been redefined from Nielson and Nielson [10] so that they become applicable to IMC$^{\mathsf{G}}$. A new operator (the chains operator) has been introduced in order to match the synchronisation construct in IMC$^{\mathsf{G}}$. The main difference with the previous work is that the Pathway Analysis captures the semantics of well-formed IMC$^{\mathsf{G}}$ systems fully, i.e. without any imprecision stemming from the analysis technique. This is possible because such systems are essentially finite-state systems. The proof of the correspondence between the semantic properties of IMC$^{\mathsf{G}}$ systems and the Pathway Analysis operators is the main theoretical contribution of this work.

Apart from the theoretical value, the work opens the possibilities for developing algorithms for checking system's properties based on Pathway Analysis operators results which, as we believe, are easier to handle than process algebraic expressions and do not have the state space explosion problem as if the corresponding Labelled Transition System would be constructed. In particular, we could introduce optimisations into the Worklist algorithm in order to merge on-the-fly states with in some sense irrelevant differences in their behaviour. In order to do this, we could, for example, merge labels beforehand based

on the results of the generate, kill and chains operators for them. We have also devised algorithms for computing states' reachability (see [17]) and bisimulation relations (see [16]) which require the Pathway Analysis to be conducted first.

We are confident that our method is applicable to a larger subclass of IMC than well-formed IMC$^{\mathrm{G}}$ expressions if we adopt a more flexible approach in the definition of transfer functions. We have assumed until now that generate, kill and chains operators results should be applicable after any number of semantics steps, i.e. they should not change. In the future work we could allow their results to change in a predictable way and correctly model a larger class of systems, i.e. systems with infinite number of states. We are also considering other process algebras and in general formalisms defined in a compositional way for future applications of our methods.

Appendix A. Proofs

Lemma 1 (Preservation of IMC$^{\mathrm{G}}$ syntax)**.** *Given a closed* IMC$^{\mathrm{G}}$ *expression* E *and* $E \xrightarrow[C]{\alpha} E'$*, then* E' *is also a closed* IMC$^{\mathrm{G}}$ *expression.*

Proof. We can prove the statement by induction on the transition derivation using Table 3. The rules (1) and (10) are base cases and the statement is clear for them because the right side of the transition cannot be a variable due to the closeness of E. The rest of the rules follow from the induction hypothesis. For the rules (9) and (16) we have to show that $E\{X/\underline{X := E}\}$ is a closed IMC$^{\mathrm{G}}$ expression if this holds for $\underline{X := E}$. The closeness for $E\{X/\underline{X := E}\}$ follows from the fact that the only free process variable in E can be X, therefore no free process identifiers are present in $E\{X/\underline{X := E}\}$ after the substitution. Moreover, if $\underline{X := E}$ is an IMC$^{\mathrm{G}}$ expression then also E is an IMC$^{\mathrm{G}}$ expression. From the rules (1–4) in Table 1 follows that we can use an IMC$^{\mathrm{G}}$ expression instead of a variable – the result will still be a valid IMC$^{\mathrm{G}}$ expression. □

Lemma 2 (Delay transitions)**.** *Given an* IMC$^{\mathrm{G}}$ *expression* E *such that* $E \xrightarrow{\alpha}$ *for some* $\alpha \in \boldsymbol{Rate}$*, then* $E \xrightarrow{\tau}$.

Proof. We can prove the statement by induction on the structure of E. The statement is clear for the base cases – the syntax rules (1–4) and (9) in Table 1: for the rules (1–4) only one transition is possible according to the semantic rules (1) and (10) in Table 3 and for the rule (9) no transitions are possible. Most of the rest of syntactic rules follow by the induction hypothesis. For example, the rule (5) follows from the induction hypothesis and the semantic rules (11) and (12) in Table 3: the second summand does not have any derivable τ-transition. For the rule (8) we have to additionally prove that $\underline{X := E} \xrightarrow{\alpha}$ iff $E \xrightarrow{\alpha}$ for all IMC$^{\mathrm{G}}$ expressions E and $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$. This can be proved by induction on the structure of E and is essentially due to the guardedness of the syntax of IMC$^{\mathrm{G}}$. □

Lemma 3 (Well-formedness)**.** *Given an* IMC$^{\mathrm{G}}$ *program* F *and* $F \xrightarrow{*} E$*, then* $\vdash_{\mathbf{fn}(E)} E$ *holds and* $Labs(E_1) \cap Labs(E_2) = \emptyset$ *for all* $E_1 \shuffle A \shuffle E_2 \preceq E$.

Proof. We prove the lemma by induction on the number of transitions in $F \xrightarrow{*} E$. The statement is trivially true for F, therefore it is enough to prove the induction step, i.e. if $F \xrightarrow{*} E \longrightarrow E'$ and $\vdash_{\mathbf{fn}(E)} E$ holds then also $\vdash_{\mathbf{fn}(E')} E'$ holds. Note that we can prove for simplicity the well-formedness preservation of E' with $\mathbf{fn}(E)$, i.e. that $\vdash_{\mathbf{fn}(E)} E'$ holds. This will also prove $\vdash_{\mathbf{fn}(E')} E'$ because, as it is easy to see, $\mathbf{fn}(E') \subseteq \mathbf{fn}(E)$. We will actually prove that from $\vdash_S E$ follows $\vdash_S E'$ for any set $S \subseteq \mathbf{Act}$, i.e. a more general statement.

The base cases are the rules (1) and (10) in Table 3 and the statement follows for them from the well-formedness rules (1–4) in Table 6 and the statement about the disjoint labelling is obvious. The rules concerning the choice and hide operators follow by induction. The rules about the parallel operator are a bit more complicated. For example, the statement concerning disjoint labelling of E' and F' in the rule (6) in Table 3 follows from the disjoint labelling of E and F and from the obvious fact that if $E \xrightarrow[C]{\alpha} E'$ then $Labs(E') \subseteq Labs(E)$. For the well-formedness of $E' \shuffle A \shuffle F'$ we can apply our induction hypothesis to the sets $S \cup \mathbf{fn}(E)$ and $S \cup \mathbf{fn}(F)$. We know that $\vdash_{S \cup \mathbf{fn}(E)} E$ and $\vdash_{S \cup \mathbf{fn}(F)} F$ hold. From the induction hypothesis follows that $\vdash_{S \cup \mathbf{fn}(E)} E'$ and $\vdash_{S \cup \mathbf{fn}(F)} F'$ hold as well. From $\mathbf{fn}(E') \subseteq \mathbf{fn}(E)$ and $\mathbf{fn}(F') \subseteq \mathbf{fn}(F)$ follow $\vdash_{S \cup \mathbf{fn}(E')} E'$ and $\vdash_{S \cup \mathbf{fn}(F')} F'$, which means due to the rule (7) in Table 6 that $\vdash_S E' \shuffle A \shuffle F'$ holds.

For proving the lemma for the rules (9) and (16) in Table 3 we need to show that the lemma's conditions hold for $E\{X/\underline{X := E}\}$ if they hold both for E and $\underline{X := E}$. Labels of synchronising processes are disjoint in $E\{X/\underline{X := E}\}$ because this is the case for both E and $\underline{X := E}$ and no substitution is conducted in the synchronising processes. There no substitutions in the synchronising processes because E is well-formed and X cannot be free in a synchronising process due to the closeness of the last, see the rule (7) in Table 6. It is left to prove that $E\{X/\underline{X := E}\}$ is well-formed relative to an action set S if this is the case for both E and $\underline{X := E}$.

In order to prove this fact by induction on the syntactic structure of E we will show that for any IMC$^{\mathrm{G}}$ expression E'' the expression $E''\{X/\underline{X := E}\}$ is well-formed relative to some action set S if this is the case for both E'' and $\underline{X := E}$ (no transition

is derivable for $E'' \in \mathbf{Var}$, $E'' \neq X$, so we can omit this case). We prove this by induction on the structure of E''. The base cases are the rules (1–2) in Table 1 and the statement follows from the well-formedness rules (3–4) in Table 6. The rules (3–6) and (8) follow from the induction hypothesis and the rules in Table 6. The rule (7) follows because E'' is well-formed, therefore both synchronising sides are closed and no substitutions can be made in them. □

Lemma 4 (Exposed labels). *Given an* IMC$^{\mathrm{G}}$ *expression* E, *then* $\mathcal{E}_\Gamma[\![E]\!] = \mathcal{E}_{\perp_{\mathfrak{V}}}[\![E]\!] = \mathcal{E}_{\perp_{\mathfrak{V}}}[\![E\{X/E'\}]\!]$ *holds for all* $\Gamma \in \mathfrak{V}$ *and* $E' \in \mathrm{IMC}^{\mathrm{G}}$.

Proof. This lemma is due to the guardedness in the syntax of IMC$^{\mathrm{G}}$, stating that the initial behaviour is fixed for all IMC$^{\mathrm{G}}$ expressions. We can prove the statement by induction on the structure of E following the rules from Table 1. Base cases are rules (1–4) and (9) from Table 1 and the statement easily follows for them from the rules (1–2) and (8) for the exposed operator in Table 7. The rest of the rules follows from the induction hypothesis and the rules for the exposed operator. □

Lemma 5 (Variable definitions under transitions). *Given an* IMC$^{\mathrm{G}}$ *program* F *and* $F \xrightarrow{*} E$, *then for all* $\underline{X := E''} \preceq E$ *holds* $\mathbf{vl}[\![\underline{X := E''}]\!](X) = \mathbf{vl}[\![F]\!](X)$.

Proof. The statement can be shown by induction on the number of steps in $F \xrightarrow{*} E$. It is essentially due to the fact that "new" occurrences of the variable definitions in the derivative expression E are only those that have been "copied" by the rules for the recursion unfolding (9) and (16) in Table 3. Exposed labels of E'' do not change after the copying, therefore the statement of the lemma holds. □

Lemma 6 (Exposed labels are a set). *Given an* IMC$^{\mathrm{G}}$ *program* F, $\Gamma = \mathbf{vl}[\![F]\!]$, $F \xrightarrow{*} E$ *and* $E'' \preceq E$, *then* $\mathcal{E}_\Gamma[\![E'']\!](\ell) \in \{0, 1\}$ *for all* $\ell \in \mathbf{Lab}$. *Moreover, for all transition pairs* $E \xrightarrow[C_1]{} G_1$ *and* $E \xrightarrow[C_2]{} G_2$ *such that their transition derivation trees are different* ($G_1 = G_2$ *is possible) holds* $C_1 \neq C_2$.

Proof. The statement of the lemma obviously holds for uniquely labelled F and all its subexpressions. Otherwise we will assume that the statement holds for E and we will show that it also holds for E', such that $E \xrightarrow{*} E'$, by induction on the transition derivation.

Most of the rules in Table 3 are obvious because the right side is a subexpression of the left side. For the rules (5–7) and (13–14) we have to use the fact proved in Lemma 3 that the labels of two synchronising processes are disjoint (this is in fact the most important reason why the statement of the lemma is true). For the rules (9) and (16) we have to prove that the statement holds for all subexpressions of $E\{X/\underline{X := E}\}$ if it holds for all subexpressions of $\underline{X := E}$ if $X \prec E$. It is clear for all $E'' \in \mathbf{Var}$ or $E'' \preceq \underline{X := E}$. Otherwise we have shown in Lemma 4 that $\mathcal{E}_\Gamma[\![E''\{X/\underline{X := E}\}]\!] = \mathcal{E}_\Gamma[\![E'']\!]$ and the induction hypothesis is applicable.

We can now prove the lemma's second statement by induction on the structure of E, using the result about the uniqueness of exposed labels. The base cases are the rules (1–4) and (9) in Table 1 and there is only one (or zero – for the rule (9)) possible transition derivation and only one exposed label. For the rules (5–7) we can use the induction hypothesis and the uniqueness of exposed labels due to which the transition derivation tree is always uniquely defined. For the rule (8) we can prove that transition derivation trees for E and $E\{X/\underline{X := E}\}$ are essentially the same, compare to Lemma 4 on exposed labels before and after substitution. □

Lemma 7 (Hidden labels). *Given a well-formed* IMC$^{\mathrm{G}}$ *expression* F, $\Lambda = \mathbf{ln}[\![F]\!]$ *and* $F \xrightarrow{*} E$, *then for all* $\ell \in \mathit{Labs}(E)$ *holds:* $\Lambda(\ell) \in \mathbf{fn}(E)$ *iff* $\Lambda(\ell) \in \mathbf{fn}(F)$.

Proof. It is easy to show (by induction on the transition derivation) that from $\Lambda(\ell) \in \mathbf{fn}(E)$ follows $\Lambda(\ell) \in \mathbf{fn}(F)$, i.e. internalised actions in F cannot become external in E. The well-formedness of F is not necessary in proving this.

On the other hand, if $\Lambda(\ell) \in \mathbf{fn}(F)$ then $\vdash_{\{\Lambda(\ell)\}} F$ holds. According to the well-formedness rule (6) in Table 6, for all hide A in $P \preceq F$ holds $\Lambda(\ell) \notin A$, i.e. free action names of F do not appear in the hide-constructs, therefore these action names will stay free in E as well after any number of semantic steps. □

Lemma 8 (Chains under substitution). *Given well-formed* IMC$^{\mathrm{G}}$ *expressions* $E' \preceq E$, $X \in \mathbf{fpi}(E')$, $\mathbf{ln}[\![E']\!] \leq \Lambda$ *and* $\mathbf{ln}[\![E]\!] \leq \Lambda$, *then it holds that* $\mathfrak{T}_\Lambda[\![E'\{X/E\}]\!] = \mathfrak{T}_\Lambda[\![E']\!] \cup \mathfrak{T}_\Lambda[\![E]\!]$ *and* $\mathfrak{T}_\Lambda[\![E']\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!]$.

Proof. The idea of the proof is that E will not appear in $E'\{X/E\}$ in parallel with some other IMC$^{\mathrm{G}}$ expressions, because this would be against the well-formedness rule (7) in Table 6 (synchronising processes should be closed) and E' would not be well-formed. As a consequence, all the chains of E will be directly included into the chains of the resulting expression ($\mathfrak{T}_\Lambda[\![E'\{X/E\}]\!] = \mathfrak{T}_\Lambda[\![E']\!] \cup \mathfrak{T}_\Lambda[\![E]\!]$). On the other hand, E' will not appear in parallel with some process inside E because

$\mathbf{fpi}(E') \neq \emptyset$, and E would not be well-formed in this case. Therefore all the chains of E' are also the chains of E ($\mathfrak{T}_\Lambda[\![E']\!] \subseteq \mathfrak{T}_\Lambda[\![E]\!]$).

The lemma can be proved formally by induction on the syntactic structure of E', i.e. for all E'' such that $E'' \preceq E'$, using all the syntactic constructs in Table 1 besides the synchronisation rule (7) and the rules for the chains operator in Table 10. □

Lemma 9 (Transition existence). *Given an* IMC$^{\mathsf{G}}$ *program* F, $F \xrightarrow{*} E$, $\Lambda = \mathbf{ln}[\![E]\!]$ *and* $\Gamma = \mathbf{vl}[\![E]\!]$, *then* $E \xrightarrow[C]{\alpha} E'$ *iff* $C \in \mathfrak{T}_\Lambda[\![E]\!]$, $C \leq \mathcal{E}_\Gamma[\![E]\!]$, $\alpha = \mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C)$, *and, in case* $\alpha \in \boldsymbol{Rate}$, *there is no* C' *such that* $C' \in \mathfrak{T}_\Lambda[\![E]\!]$, $C' \leq \mathcal{E}_\Gamma[\![E]\!]$ *and* $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C') = \tau$.

Proof. It is easy to see that it is enough to prove the first part of the statement, because, if there would exist an exposed chain C' with $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C') = \tau$, then there would exist a τ-transition from E, which is not possible according to Lemma 2.

Note that it is also easy to see that in case of transition existence $\alpha = \mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C)$ holds: it is enough to check the SOS rules in Table 3, in particular, the rules (1) and (10), and the rules for the operator **ln** in Table 9 on one hand and Lemma 7 (free action names in F stay free in E) on the other hand.

It is therefore left to show that $E \xrightarrow[C]{\alpha} E'$ iff $C \in \mathfrak{T}_\Lambda[\![E]\!]$, $C \leq \mathcal{E}_\Gamma[\![E]\!]$. We prove this by induction on the syntactic structure of E, taking into account the rules for the exposed and the chains operators in Tables 7 and 10. The rules (1–4) and (9) in Table 1 are base cases, and the statement of the lemma is clearly true for them (there is only one label that can be exposed, is in the chain and can be executed). For the rest of the rules the statement follows from the induction hypothesis and due to the definitions of the exposed and chains operators which are defined inductively on the syntax of E. In particular, compare the semantic rules (4–6) and (13–14) in Table 3 to the rule (5) of the chains operator in Table 10. For the rule (8), i.e. recursion unfolding, remember that $\mathcal{E}_\Gamma[\![\underline{X := E}]\!] = \mathcal{E}_\Gamma[\![E\{X/\underline{X := E}\}]\!]$ and $\mathfrak{T}_\Lambda[\![\underline{X := E}]\!] = \mathfrak{T}_\Lambda[\![E\{X/\underline{X := E}\}]\!]$ according to Lemmas 4 and 8.

In case synchronisation is applied on top of recursion unfolding, then considerations similar to the ones in Lemma 7 should be used. In this case there exists some P (to which synchronisation is applied) such that $\underline{X := E} \preceq P \preceq E$ and $\vdash_{\mathbf{fn}(P)} P$ (this can be proved by induction on the transition sequence $F \xrightarrow{*} E$). If some action name a is free in $\underline{X := E}$, then it also will be free in $E\{X/\underline{X := E}\}$ because no hide-construct with a is allowed in E according to the well-formedness rules. □

Lemma 10 (Kill/generate under substitution). *Given well-formed* IMC$^{\mathsf{G}}$ *expressions* $E' \preceq E$, $X \in \mathbf{fpi}(E')$, $\Gamma(X) = \mathcal{E}_{\perp_{\mathfrak{V}}}[\![\mathbf{E}]\!]$, *then* $\mathcal{G}'_\Gamma[\![E'\{X/E\}]\!] = \mathcal{G}'_\Gamma[\![E']\!]$ *and* $\mathcal{K}'[\![E'\{X/E\}]\!] = \mathcal{K}'[\![E']\!]$ *hold.*

Proof. The statement is easy to prove for the kill operator because from Lemma 4 follows $\mathcal{E}_\Gamma[\![E'\{X/E\}]\!] = \mathcal{E}_\Gamma[\![E']\!]$, and from the rule for the $\mathcal{K}'$ operator in Table 13 it is clear that $\mathcal{K}'[\![E']\!](\ell) = \mathcal{E}_\Gamma[\![E']\!]$ if $\ell \in \mathbf{dom}(\mathcal{E}_\Gamma[\![E']\!])$ and $\mathcal{K}'[\![E']\!](\ell) = \perp_{\mathfrak{M}}$ otherwise.

For the generate operator, in case $E' = \alpha^\ell.X$ for some $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$ and $\ell \in \mathbf{Lab}$, then we can use the fact that $\Gamma(X) = \mathcal{E}_{\perp_{\mathfrak{V}}}[\![\mathbf{E}]\!]$ and the rules for the $\mathcal{G}'$ in Table 11. Otherwise the statement can be proved by induction on the syntax of $\mathcal{G}'$ using the rules in Table 11. □

Lemma 11 (Kill/generate). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Gamma = \mathbf{vl}[\![F]\!]$ *and* $F \xrightarrow{*} E \xrightarrow[C]{\alpha} E'$, *then* $\mathcal{E}_\Gamma[\![E']\!] = \mathcal{E}_\Gamma[\![E]\!] - \sum_{C(\ell)>0} \mathcal{K}'[\![E]\!](\ell) + \sum_{C(\ell)>0} \mathcal{G}'_\Gamma[\![E]\!](\ell)$.

Proof. Note that all the exposed labels of E are unique (Lemma 6) and all the labels in C are exposed (Lemma 9), therefore all the labels in C are unique and it is enough to use the generate and kill operator for each label ℓ in C (such that $C(\ell) > 0$) only once.

We prove the lemma by induction on the transition derivation $E \xrightarrow[C]{\alpha} E'$, thus proving that the statement also holds for all $E'' \preceq E$. For the rules (1) and (10) in Table 3 the statement follows from the rules for the generate and kill operators of the exposed labels in Tables 11 and 13 (currently exposed labels are killed and the exposed labels of E' are generated). For the most of the rest of the rules in Table 3 the statement follows from the induction hypothesis, taking into account that labels of parallel processes are disjoint and we can therefore apply the induction hypothesis to them separately.

It is left to show that the statement also holds for the recursion unfolding rules (9) and (16) in Table 3, i.e. that the statement holds for $\underline{X := E}$ if it holds for $E\{X/\underline{X := E}\}$. We can, however, use the fact that $\mathcal{K}'[\![\underline{X := E}]\!] = \mathcal{K}'[\![E]\!] = \mathcal{K}'[\![E\{X/\underline{X := E}\}]\!]$ and $\mathcal{G}'_\Gamma[\![\underline{X := E}]\!] = \mathcal{G}'_\Gamma[\![E]\!] = \mathcal{G}'_\Gamma[\![E\{X/\underline{X := E}\}]\!]$, taking into account that $\Gamma = \mathbf{vl}[\![F]\!]$ contains an element characterising exposed labels of $\underline{X := E}$ (see also Lemma 5) and using Lemma 10. Altogether we can derive that if the statement of the lemma holds for $E\{X/\underline{X := E}\}$ then it also holds for $\underline{X := E}$. □

Lemma 12 (Chains of subexpressions). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Lambda = \mathbf{ln}[\![F]\!]$, $F \xrightarrow{*} E$ *and* $E'' \preceq E' \preceq E$ *such that no sychronisation construct is applied to* E'' *in* E', *then for all* C *such that* $dom(C) \subseteq Labs(E'')$ *holds:* $C \in \mathfrak{T}_\Lambda[\![E'']\!]$ *iff* $C \in \mathfrak{T}_\Lambda[\![E']\!]$.

Proof. The statement holds for $F = E$ due to the unique labelling of F and also because in this case $\mathfrak{T}_\Lambda[\![E'']\!] \subseteq \mathfrak{T}_\Lambda[\![E']\!]$, taking into account the rules for the chains operator in Table 10 (only parallel operator does not include the chains of subexpressions directly). Otherwise assume that the statement holds for E, $E \longrightarrow E'$, and we will prove that the statement holds for E' as well by induction on the transition derivation.

The rules (1–10) in Table 3 are clear because the right side is a subexpression of the right side without synchronisation construct. For the rest of the rules the statement follows from the induction hypothesis besides the rules (9) and (16) where we can use Lemma 8. The last states that $\mathfrak{T}_\Lambda[\![E''\{X/\underline{X := E}\}]\!] = \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ for all $X \prec E'' \prec \underline{X := E}$, and the statement is clear. □

Lemma 13 (Chains under transitions). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Lambda = \mathbf{ln}[\![F]\!]$, $F \xrightarrow{*} E$ *and* $\mathbf{dom}(C) \subseteq \mathit{Labs}(E)$, *then* $C \in \mathfrak{T}_\Lambda[\![F]\!]$ *iff* $C \in \mathfrak{T}_\Lambda[\![E]\!]$ *and* $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C) = \mathbf{name^h}_{\Lambda,\mathbf{fn}(E)}(C)$.

Proof. We prove the statement by induction on the number of steps in $F \xrightarrow{*} E$. It is trivially true for F. Otherwise we will prove that for $E \longrightarrow E'$ and $\mathbf{dom}(C) \subseteq \mathit{Labs}(E')$ holds: $C \in \mathfrak{T}_\Lambda[\![E]\!]$ iff $C \in \mathfrak{T}_\Lambda[\![E']\!]$. We will prove this by induction on the transition derivation, using Lemma 12. Note that $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C') = \mathbf{name^h}_{\Lambda,\mathbf{fn}(E)}(C')$ is true according to Lemma 7.

The statement is clear for the base cases – the rules (1) and (10) in Table 3 – because the right side of the rules is a subexpression of the left side (without applying the synchronisation construct), and Lemma 12 is applicable. For the most of the other rules the induction hypothesis is directly applicable. For the rules with a synchronisation construct a chain C should be eventually "divided" into chains C_1 and C_2 between two expressions put in parallel (this is possible, because their labels are disjoint), in order to apply the statement in Lemma 12 to two expressions separately. Additionally, we should apply Lemma 7 to both parallel processes which are well-formed (see the well-formedness rules in Table 6), therefore all the labels that are initially not hidden stay not hidden after the transition. Altogether two chains C_1 and C_2 in parallel processes in E' (that exist according to the induction hypothesis) are combined together into the chain $C = C_1 + C_2$, $C \in \mathfrak{T}_\Lambda[\![E']\!]$.

For the rules (9) and (16) we have $\mathfrak{T}_\Lambda[\![E\{X/\underline{X := E}\}]\!] = \mathfrak{T}_\Lambda[\![\underline{X := E}]\!]$ according to Lemma 8, therefore if the statement is true for $E\{X/\underline{X := E}\}$ (this is the case due to the induction hypothesis), then it is also true for $\underline{X := E}$. □

Lemma 14 ([Kill/generate under transitions). *Given an* IMC$^{\mathsf{G}}$ *program* F, $\Gamma = \mathbf{vl}[\![F]\!]$ *and* $F \xrightarrow{*} E$, *then for all* $E'' \preceq E$, $\ell \in \mathbf{dom}(\mathcal{E}_\Gamma[\![E'']\!])$ *holds:* $\mathcal{G}'_\Gamma[\![E'']\!](\ell) = \mathcal{G}_\Gamma[\![F]\!](\ell)$ *and* $\mathcal{K}'[\![E'']\!](\ell) = \mathcal{K}[\![F]\!](\ell) \sqcap \mathcal{E}_\Gamma[\![E'']\!]$.

Proof. The statement clearly holds for $E = F$ because F is uniquely labelled and each label occurs in it only once. Note that $\mathcal{K}'[\![E'']\!](\ell) \prec \mathcal{K}[\![F]\!](\ell)$ is in fact possible if we have $E'' + G \preceq F$ for some G, because $\mathcal{K}[\![F]\!](\ell)$ would "kill" all the exposed labels of G as well. This is, however, taken into account by stating that only exposed labels of E'' can be killed in E'' by executing ℓ (by taking the greatest lower bound).

Otherwise we will prove the statement by induction of the transition derivation $E \longrightarrow E'$, i.e. prove that the statements holds for E' assuming that it holds for E. It is clear for the base case – the semantic rules (1) an (10) in Table 3 – because the right side is a subexpression of the left side. For the rest of the rules we can use the induction hypothesis concerning E (remember that parallel processes have disjoint labels) besides the rules (9) and (16). For them we have to prove that if the statement holds for $\underline{X := E}$ then it also holds for $E''\{X/\underline{X := E}\}$ if $E'' \preceq E$ and $X \prec E''$. This can be proved by induction on the syntactic structure of E'' using Lemmas 4 (concerning the exposed operator) and 10 (concerning the generate and kill operators). From the two lemmas we can derive that $\mathcal{G}'_\Gamma[\![E''\{X/\underline{X := E}\}]\!](\ell) = \mathcal{G}'_\Gamma[\![E'']\!](\ell)$ and $\mathcal{K}'[\![E''\{X/\underline{X := E}\}]\!](\ell) \sqcap \mathcal{E}_\Gamma[\![E''\{X/\underline{X := E}\}]\!] = \mathcal{K}[\![E'']\!](\ell) \sqcap \mathcal{E}_\Gamma[\![E'']\!]$ for all $E'' \preceq E$ and $\ell \in \mathbf{dom}(\mathcal{E}_\Gamma[\![E'']\!])$, and the induction hypothesis is applicable. □

Theorem 15 (Worklist algorithm). *Given an* IMC$^{\mathsf{G}}$ *program* F, *then the Worklist algorithm terminates on* F *and constructs the LTS, whose initial state* $\mathcal{E}_\Gamma[\![F]\!]$ *with* $\Gamma = \mathbf{vl}[\![F]\!]$ *is strongly bisimilar to the LTS induced by the semantics of* F, *i.e.* $F \sim \mathcal{E}_\Gamma[\![F]\!]$.

Proof. The idea of the proof is to show that the relation $\mathcal{R}$ defined on the set $\{E, \mathcal{E}_\Gamma[\![E]\!] | F \xrightarrow{*} E\}$ as $\mathcal{R} = \{(E_1, E_2) | \mathcal{E}_\Gamma[\![E_1]\!] = \mathcal{E}_\Gamma[\![E_2]\!]\} \cup \{(\mathcal{E}_\Gamma[\![E]\!], \mathcal{E}_\Gamma[\![E]\!])\} \cup \{(E, \mathcal{E}_\Gamma[\![E]\!])\} \cup \{(\mathcal{E}_\Gamma[\![E]\!], E)\}\}$ (i.e. the relation on states with the same exposed labels) is a strong bisimulation relation. First, it is clearly an equivalence relation. It is obviously reflexive, symmetric and transitive (can be proved by checking all the cases). Second, it would be enough to prove that each transition from E has a corresponding transition from $\mathcal{E}_\Gamma[\![E]\!]$ and vice versa, such that both transitions are decorated with the same action name or delay rate and lead to bisimilar states. More formally, we will prove that $E \xrightarrow[C]{\alpha} E'$ iff $\mathcal{E}_\Gamma[\![E]\!] \xrightarrow[C]{\alpha} \mathcal{E}_\Gamma[\![E']\!]$ for all $\alpha \in \mathbf{Act} \cup \{\tau\} \cup \mathbf{Rate}$: the first condition in Definition 3 (concerning $\alpha \in \mathbf{Act} \cup \{\tau\}$) holds directly, and the second follows by summing all the rates in the transitions into each equivalence class. The statement will hold due to the properties of the Pathway Analysis operators.

By combining together Lemmas 9 and 13 (existence of the transition and using chains of F instead of E) and Lemmas 11 and 14 (generate and kill operator can predict exposed labels of E', generate and kill operators computed for F can be used instead of the ones computed for E) follows that all the transitions from E are constructed by the Worklist algorithm for

$\mathcal{E}_\Gamma[\![E]\!]$ and vice versa for each E reachable from F. Formally, $F \xrightarrow{*} E \xrightarrow[C]{\alpha} E'$ iff there exists $C \in \mathfrak{T}_\Lambda[\![F]\!]$ such that $C \leq \mathcal{E}_\Gamma[\![E]\!]$, $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C) = \alpha$, $\mathcal{E}_\Gamma[\![E']\!] = \mathcal{E}_\Gamma[\![E]\!] - \sum_{C(\ell)>0} \mathcal{K}[\![F]\!](\ell) + \sum_{C(\ell)>0} \mathcal{G}_\Gamma[\![F]\!](\ell)$ and, moreover, if $\alpha \in \mathbf{Rate}$ then there is no C' such that $C' \in \mathfrak{T}_\Lambda[\![F]\!]$, $C' \leq \mathcal{E}_\Gamma[\![E]\!]$ and $\mathbf{name^h}_{\Lambda,\mathbf{fn}(F)}(C') = \tau$.

For each state E reachable from F a corresponding state $\mathcal{E}_\Gamma[\![E]\!]$ will be created by the Worklist algorithm – this can be proved by induction on the number of steps in $F \xrightarrow{*} E$. On the other hand, for each state M reachable from $\mathcal{E}_\Gamma[\![F]\!]$ there exist a (possibly more than one) state in the LTS induced by F which is bisimilar to M. This follows from the definition of the Worklist algorithm which only creates states that are "predicted" by the Pathway Analysis results, i.e. which have their corresponding states in the semantics of F. Altogether, $E \sim \mathcal{E}_\Gamma[\![E]\!]$ for all E reachable from F, therefore $F \sim \mathcal{E}_\Gamma[\![F]\!]$ as well.

The Worklist algorithm terminates on F because the number of states that can be constructed are maximally $2^{|Labs(F)|}$ (according to Lemma 6, all the states reachable from F have maximal one exposed label of each kind) and, as we have already mentioned, for each state M created by the Worklist algorithm there exists a state E such that $F \xrightarrow{*} E$ and $\mathcal{E}_\Gamma[\![E]\!] = M$. □

References

[1] Ed Brinksma, Holger Hermanns, Process algebra and Markov chains, in: Ed Brinksma, Holger Hermanns, Joost-Pieter Katoen (Eds.), Euro Summer School on Trends in Computer Science, Lecture Notes in Computer Science, vol. 2090, Springer, 2000, pp. 183–231.ISBN 3-540-42479-2

[2] Edmund M. Clarke, E. Allen Emerson, A. Prasad Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Systems 8 (1986) 244–263.

[3] Holger Hermanns, Interactive Markov Chains: The Quest for Quantified Quality, Lecture Notes in Computer Science, vol. 2428, Springer, 2002.ISBN 3-540-44261-8. URL: http://dx.doi.org/10.1007/3-540-45804-2

[4] Charles A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.ISBN 0-13-153271-5

[5] John B. Kam, Jeffrey D. Ullman, Monotone data flow analysis frameworks, Acta Inform. 7 (1977) 305–317.

[6] Sebastian Nanz, Flemming Nielson, Hanne Riis Nielson, Modal abstractions of concurrent behaviour, in: María Alpuente, Germán Vidal (Eds.), Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, Lecture Notes in Computer Science, vol. 5079, Springer, 2008, pp. 159–173.ISBN 978-3-540-69163-1

[7] Flemming Nielson, Hanne Riis Nielson, Chris L. Hankin, Principles of Program Analysis, Springer, 1999.(Second printing, 2005)

[8] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, Debora Schuch da Rosa, Static analysis for systems biology, in: Proceedings of WISICT '04, Trinity College Dublin, 2004.

[9] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, Debora Schuch da Rosa, Control flow analysis for BioAmbients, Electron. Notes Theoret. Comput. Sci. 180 (3) (2007) 65–79.

[10] Hanne Riis Nielson, Flemming Nielson, Program Analysis and Compilation, Theory and Practice, Data Flow Analysis for CCS, Springer, Berlin, Heidelberg, 2007.pp. 311–327. ISBN 3-540-71315-8, 978-3-540-71315-9. Available from: <http://dl.acm.org/citation.cfm?id=1805793.1805810>

[11] Hanne Riis Nielson, Flemming Nielson, A monotone framework for CCS, Comput. Lang. Syst. Struct. 35 (4) (2009) 365–394.ISSN 1477-8424

[12] Henrik Pilegaard, Language Based Techniques for Systems Biology, Ph.D. thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2007. Available from: <http://www2.imm.dtu.dk/pubdb/p.php?5352>.

[13] Henrik Pilegaard, Flemming Nielson, Hanne Riis Nielson, Pathway analysis for BioAmbients, J. Logic Algebr. Program. 77 (1–2) (2008) 92–130.

[14] Gordon Plotkin, A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, University of Aarhus, 1981. Available from: <http://citeseer.ist.psu.edu/plotkin81structural.html>.

[15] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, Ehud Y. Shapiro, BioAmbients: an abstraction for biological compartments, Theoret. Comput. Sci. 325 (1) (2004) 141–167.

[16] Nataliya Skrypnyuk, Verification of Stochastic Process Calculi, Ph.D. thesis, Technical University of Denmark, Department of Informatics and Mathematical Modeling, Language-Based Technology, Kgs. Lyngby, Denmark, 2011.

[17] Nataliya Skrypnyuk, Flemming Nielson, Reachability for finite-state process algebras using Static Analysis, in: Giorgio Delzanno, Igor Potapov (Eds.), Reachability Problems, 5th International Workshop, RP 2011, Genova, Italy, September 28–30, 2011, Proceedings, Lecture Notes in Computer Science, vol. 6945, Springer, 2011, pp. 231–244. ISBN 978-3-642-24287-8.

[18] Nataliya Skrypnyuk, Flemming Nielson, Henrik Pilegaard, Pathway Analysis for IMC, in: Proceedings of the 21st Nordic Workshop on Programming Theory (NWPT'09), Kgs. Lyngby, Denmark, October 2009, Technical University of Denmark, 2009.