 Electronic Notes in Theoretical Computer Science 171 (2007) 155–170 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

The Decidability of the Structural Congruence for Beta-binders

# Corrado Priami[1](#_bookmark0) , Alessandro Romanel[2](#_bookmark1)

*The Microsoft Research - University of Trento Centre for Computational and Systems Biology Trento, Italy*

Abstract

Beta-binders is a recent process algebra developed for modeling and simulating biological systems. As usual for process calculi, the semantic definition heavily relies on a structural congruence. The treatment of the structural congruence is essential for implementation. The proof of the decidability of this congruence, reported in this paper, is a first step towards implementations.

*Keywords:* Process Calculi, Structural Congruence, Decidability.

# Introduction

Systems Biology studies the behaviour and relationships of the elements composing a particular biological system. Recently, some authors [[16](#_bookmark40)] argue that concurrency theory and *process algebras* are usuful to specify and simulate the behaviour of living matter. As a consequence, a number of process calculi has been adapted or newly developed for applications in systems biology [[14](#_bookmark37),[15](#_bookmark39),[2](#_bookmark26),[12](#_bookmark36)]. The operational semantics of such process algebras allows to describe the dynamical evolution of a system. This semantics is near to the implementation and is usually strongly related to the concept of *structural congruence*. This paper focuses on *Beta-binders*, a process algebra introduced for better representing biological interactions. We develop on the structural congruence of both qualitative and quantitative [[3](#_bookmark27)] version of the calculus. The proof of the decidability of the structural congruence for Beta- binders, reported in this paper, is in fact a first step towards the implementation of a familiy of efficient stochastic simulators for Beta-binders.

1 Email: [priami@msr-unitn.unitn.it](mailto:priami@msr-unitn.unitn.it)

2 Email: [a.romanel@msr-unitn.unitn.it](mailto:%20a.romanel@msr-unitn.unitn.it)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2007.05.014

The remainder of the paper is structured as follows. In Sect. [2](#_bookmark2) a short intro- duction of Beta-binders is reported, along with the description of some particular normal forms and an overview of the decidability of the structural congruence for the *π*-calculus. In Sect. [3](#_bookmark4) and Sect. [4](#_bookmark6) the proof of the decidability of the structural congruence for Beta-binders is presented. In Sect. [5](#_bookmark23) a generalization of the proof is given.

# Preliminaries

In this section we report a short introduction to Beta-binders and a short overview of the most important results regarding the decidability of the structural congruence for the *π*-calculus.

* 1. *Beta-binders*

Beta-binders [[12](#_bookmark36),[13](#_bookmark38)] is a process algebra developed for better representing the interactions between biological entities. The main idea is to encapsulate *π*-calculus processes into *boxes* with interaction capabilities, also called *beta-processes*. Like the *π*-calculus also Beta-binders is based on the notion of *naming*. Thus, we assume the existence of a countably infinite set N of names (ranged over by lower-case letter). The processes wrapped into boxes, also called *pi-processes*, are given by the following context free grammar:

*P* ::= *nil* | *π.P* | *P* |*Q* | (*νy*)*P* | !*P*

*π* ::= *x*⟨*y*⟩ | *x*(*y*) | *τ* | *expose*(*x,* Γ) | *hide*(*x*) | *unhide*(*x*)

The syntax of the *π*-calculus is enriched by the last three options for *π* to manipulate the interactions *sites* of the boxes. Beta-processes are defined as pi-processes prefixed by specialised binders that represent interaction capabilities. An *elementary beta binder* has the form *β*(*x,* Γ) (active) or *βh*(*x,* Γ) (hidden) where the name *x* is the subject of the beta binder and Γ represents the type of *x*. With *β* we denote either *β* or *βh*. A *well-formed* beta binder (ranged over by *B*, *B*1, *B*',

^

·· ·) is a non-empty string of elementary beta binder where subjects are all distinct. The function *sub*(*B*) returns the set of all the beta binder subjects in *B*. Moreover, *B*∗ denote either a well-formed beta binder or the empty string. *Beta-processes* (ranged over by *B*, *B*1, *B*', ·· ·) are generated by the following context free grammar:

*B* ::= *N il* | *B*[*P* ] | *B* || *B*

The system is either the deadlock beta-process *Nil* or a parallel composition of boxes *B*[*P* ]. The structural congruence for Beta-binders is defined through a structural congruence over pi-processes and a structural congruence over beta-processes.

Definition 2.1 The structural congruence over pi-processes, denoted ≡, is the

smallest relation which satisfies the laws in Fig. [1](#_bookmark3) (group a) and the structural congruence over beta-processes, denoted ≡, is the smallest relation which satisfies the laws in Fig. [1](#_bookmark3) (group b).

|  |  |
| --- | --- |
| group *a* - pi-processes | group *b* - beta-processes |
| * 1. *P*1 ≡ *P*2   if *P*1 and *P*2 are *α*-equivalent   * 1. *P*1 | (*P*2 | *P*3) ≡ (*P*1 | *P*2) | *P*3   2. *P*1 | *P*2 ≡ *P*2 | *P*1   3. *P* | *nil* ≡ *P*   4. (*νz*)(*νw*)*P* ≡ (*νw*)(*νz*)*P*   5. (*νz*)*P* ≡ *P* if *x* /∈ *fn*(*P* )   6. (*νz*)(*P*1 | *P*2) ≡ *P*1 | (*νz*)*P*2 if *z* /∈ *fn*(*P*1)   7. !*P* ≡ *P* | !*P* | * 1. B[*P*1] ≡ B[*P*2] if *P*1 ≡ *P*2   2. *B*1 || (*B*2 || *B*3) ≡ (*B*1 || *B*2) || *B*3   3. *B*1 || *B*2 ≡ *B*2 || *B*1   4. *B* || *Nil* ≡ *B*   5. B1B2[*P* ] ≡ B2B1[*P* ]   6. B∗*β*b(*x* : Γ)[*P* ] ≡ B∗*β*b(*y* : Γ)[*P* {*y/x*}] with *y* fresh in *P* and *y* /∈ *sub*(B∗) |

Fig. 1. Structural laws for Beta-binders.

Notice that the same symbol is used to denote both congruences. The intended relation is disambiguated by the context of application.

In the stochastic extension of Beta-Binders [[3](#_bookmark27)] the syntax is enriched in order to allow a Gillespie Stochastic Simulation Algorithm (SSA) implementation [[9](#_bookmark33)]. The prefix *π.P* is replaced by (*π, r*)*.P* , where *r* is the single parameter defining an expo- nential distribution that drives the stochastic behaviour of the action corrisponding to the prefix *π*. Moreover, the classical replication !*P* is replaced by the so called *guarded replication* !*π.P* . In order to manage this type of replication, the structural law !*P* ≡ *P* | !*P* is replaced by the law !(*π, r*)*.P* ≡ (*π, r*)*.*(*P* |!(*π, r*)*.P* ).

Notice that for the purpose of this paper we are not interested in the semantic of the language. We refer the reader to [[12](#_bookmark36),[13](#_bookmark38),[3](#_bookmark27)] for a more detailed description of both the qualitative and quantitative version of Beta-binders.

* 1. *Normal forms*

In [[7](#_bookmark31)] two normal forms for *π*-calculus processes, called *webform* and *super webform*, are introduced.

A process *P* is *fresh* if *x* /∈ *fn*(*P* ) whenever (*νx*) is not in the scope of any guard or replication (called *outer restriction* ) in *P* , and every restriction (*νx*) occurs at most once as outer restriction in *P* . For each process *P* there exists a fresh process *P* ' such that *P* ' ≡*α P* . Let *P* be a fresh process. Let os(*P* ), the *outer subterms* of *P* , be the set of occurrences of subterm *π.Q* and !*Q* of *P* that are not in the scope of any guard or replication. Let or(*P* ), the *outer restrictions* of *P* , be the set of names *x* such that (*νx*) is not in the scope of any guard or replication in *P* and such that *x* occurs free in some outer subterm of *P* . Finally, let og(*P* ), the *outer graph* of *P* , be the undirected bipartite graph with nodes os(*P* ) ∪ or(*P* ) and with an edge between *R* ∈ os(*P* ) and *x* ∈ or(*P* ) if *x* ∈ *fn*(*R*).

A process *P* = (*νx*1)*...*(*νxk*)(*P*1 | *...* | *Pm*) with *k* ≥ 0 and *m* ≥ 1 is a *web* if: (1) every process *Pi* is a replication !*Q* or a guarded process *π.Q*; (2) *x*1*, ..., xk* are all distinct (*P* is fresh); (3) for each *xj* there exists a process *Pi* such that *xj* ∈ *fn*(*Pi*);

(4) *og*(*P* ) is connected. Every replication !*P* and every guarded process *π.P* is a web (with *k* = 0 and *m* = 1). No web is congruent to the inactive process *nil*. A web should be denoted with the set {*x*1*, ..., xk, P*1*, ..., Pm*} wich lists the names of the outer restrictions and the outer subterms. A *webform* of a fresh process *P* , denoted with *wf* (*P* ), is the composition of all the webs (*νx*1)*...*(*νxk*)(*P*1 | *...* | *Pm*) such that {*x*1*, ..., xk, P*1*, ..., Pm*} is a connected component of *og*(*P* ) (in [[7](#_bookmark31)] an inductively computation of *wf* (*P* ) is reported). If *og*(*P* ) is the empty graph, then *wf* (*P* )= *nil*. The *super webform* of a fresh process *P* , denoted with *swf* (*P* ), is inductively defined in the following way: *swf* (*P* )= *wf* (*subwf* (*P* )) where, by definition, *subwf* (*P* ) is obtained from *P* by replacing every outer subterm *π.Q* of *P* with *π.swf* (*Q*) and every outer subterm !*Q* with !*swf* (*Q*). See [[7](#_bookmark31)] for a more detailed description.

* 1. *The decidability of the structural congruence for the π-calculus*

The most important results for the decidability of the structural congruence for the *π*-calculus are those presented by J. Engelfriet in [[4](#_bookmark28)] and by J. Engelfriet e

T.E. Gelsema in [[5](#_bookmark29),[6](#_bookmark30),[8](#_bookmark32),[7](#_bookmark31)]. They consider the syntax of the *small π-calculus* (pre- sented in [[11](#_bookmark35)]) and the congruences over the set of processes generated by a subcol- lection of the structural laws presented in Fig. [2](#_bookmark5) (where, for our purpose, we add the congruence ≡min). The standard structural congruence, defined in [[4](#_bookmark28),[5](#_bookmark29)] and denoted with ≡std, is determined by the laws (*α*), (1*.*1), (1*.*2), (1*.*3), (2*.*1), (2*.*2), (2*.*3) and (3*.*1). In [[6](#_bookmark30)], the middle congruence, denoted with ≡md, was introduced to give a different view of the treatment of replication. The decidability of the middle congruence was shown in [[8](#_bookmark32)]. They reduce it to the decidability of extended struc- tural congruence, denoted with ≡ext, that was shown in [[5](#_bookmark29)]. In [[7](#_bookmark31)], instead, was shown the decidability of the replication free congruence, denoted with ≡!fr, and the decidability of the standard congruence for the subclass of *replication restricted* processes. Formally, a process *P* is *replication restricted* if for every subterm !*R* of *P* and every (*νx*) that covers !*R* in *P* , if *x* ∈ *fn*(*R*), then *x* ∈ *fn*(*S*) for every component *S* of *R* where with component we mean a *web*. The decidability of the structural congruence for this subclass of processes is reduced to the problem of solving certain systems of linear equations with coefficents in N.

# Structural congruence over beta-processes

The structural laws for Beta-binders, presented in Fig. [1](#_bookmark3), are divided in two groups: the laws for pi-processes (*group a*) and the laws for beta-processes (*group b*). From law *b.*1 it turns out that the decidability of the structural congruence over pi- processes is a necessary condition for the decidability of the structural congruence over beta-processes.

The congruences that we consider in this paper are ≡min and ≡std. Congruence

*bb bb*

≡min is generated by the structural laws of *group a* and the laws *b.*1, *b.*5 and *b.*6. Congruence ≡std is generated by all the structural laws of *group a* and *group b*.

*bb*

*bb*

First, we prove the decidability of the congruence ≡min making some assump-

*bb*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| rule | | ≡min | ≡*ν*fr | ≡!fr | ≡std | ≡md | ≡ext |
| (*α*) | *P*1 ≡ *P*2 if *P*1 and *P*2 are *α*-equivalent | + | + | + | + | + | + |
| (1*.*1) | *P* | *nil* ≡ *P* |  | + | + | + | + | + |
| (1*.*2) | *P*1 | *P*2 ≡ *P*2 | *P*1 | + | + | + | + | + | + |
| (1*.*3) | *P*1 | (*P*2 | *P*3) ≡ (*P*1 | *P*2) | *P*3) | + | + | + | + | + | + |
| (2*.*1) | (*νz*)(*νw*)*P* ≡ (*νw*)(*νz*)*P* | + | + | + | + | + | + |
| (2*.*2) | (*νz*)*P* ≡ *P* |  |  | + | + | + | + |
|  | if *x* /∈ *fn*(*P* ) |  |  |  |  |  |  |
| (2*.*3) | (*νz*)(*P*1 | *P*2) ≡ *P*1 | (*νz*)*P*2 |  |  | + | + | + | + |
|  | if *z* /∈ *fn*(*P*1) |  |  |  |  |  |  |
| (3*.*1) | !*P* ≡ *P* | !*P* |  | + |  | + | (+) | + |
| (3*.*6) | !(*P* | *Q*) ≡!( *P* | *Q*) | *P* |  |  |  |  | + | (+) |
| (3*.*2) | !(*P* | *Q*) ≡!*P* | !*Q* |  |  |  |  |  | + |
| (3*.*3) | !!*P* ≡!*P* |  |  |  |  |  | + |
| (3*.*4) | !*nil* ≡ *nil* |  |  |  |  |  | + |
| (2*.*4) | (*νx*)*π.P* ≡ *π.*(*νx*)*P* |  |  |  |  |  | + |
|  | if *x* /∈ *n*(*π*) |  |  |  |  |  | + |

Fig. 2. Structural laws for the *π*-calculus.

tions: (1) we restrict the *well-formedness* definition by assuming that a *well-formed* beta binder (ranged over by *B*, *B*1, *B*', ·· ·) is a non-empty string of elementary beta binder where subjects and types are all distinct; (2) we assume that the structural congruence over pi-processes is decidable, and therefore we assume that there exists a function *P iStdCong* : P × P → {*true, false*} that accepts two pi-processes as parameters and returns *true* if the pi-processes are structural congruent, and re- turns *false* otherwise; (3) we assume that the types of the beta binders are defined over algebric structures with decidable equality relation, and therefore we assume that there exists a function *Equal* :Γ × Δ → {*true, false*} that accepts two types as parameters and returns *true* if the types are equal, and returns *false* otherwise. Then, we prove the decidability of the congruence ≡std always under the previous assumptions. Finally, we will analyze in detail the decidability of the structural congruence over pi-processes.

*bb*

We consider two beta-processes *B*[*P* ] and *B*'[*P* ']. We notice that the laws of

*group b* related to the congruence ≡min

*bb*

only refers to the structure of the beta

binders lists *B* and *B*'. In fact, the two lists are considered congruent only if they are equal (law *b.*1), or if *B* is a permutation of *B*' that satisfies the laws *b.*5 and *b.*6.

For this reason the decidability of the congruence ≡min

*bb*

can be described

through a function *BBMinCong* : *B*[*P* ] × *B*[*P* ] → {*true, false*} defined by induction on the structure of beta-processes in the following way:

*BBMinCong*(*є*[*P* ]*, є*[*P* ']) = *P iStdCong*(*P, P* ')

*BBMinCong*(*є*[*P* ]*, B*'[*P* ']) = *BBMinCong*(*B*[*P* ]*, є*[*P* ']) = *false*

8>< *BBMinCong*(*B*∗[*P* {*z/x*}]*, B*∗*B*∗[*P* '{*z/y*}]) *if* (1)

*BBMinCong*(*β*(*x* : Γ)*B* [*P* ]*, B* [*P* ]) = *BBMinCong*(*B* [*P* ]*, B*1*B*2[*P* ]) *if* (2)

*false o.w.*

b >:

∗

'

'

∗

∗

∗

1

'

2

' ∗ ∗ ' ∗ ∗ ∗

1. *B* = *B*1 *β*b(*y* : Δ)*B*2 *with Equal*(Γ*,* Δ) ∧ (*x* /= *y*) ∧ *z* /∈ *fn*(*P* ) ∪ *fn*(*P* ) ∪ *sub*(*B* ) ∪ *sub*(*B*1 *B*2 )
2. *B* = *B*1 *β*b(*x* : Δ)*B*2 *with Equal*(Γ*,* Δ)

'

∗

∗

If the lists *B* and *B*' are not empty, then there are three different cases: (1) if a type corrispondence between the first beta binders *β*^(*x* : Γ) of *B* and one beta binder *β*^(*y* : Δ) of *B*' such that (*x* /= *y*) exists, then the function *BBMinCong* is recursively invoked on the beta-processes *B*1[*P* {*z/x*}] and *B*2[*P* '{*z/y*}], where *z* /∈ *fn*(*P* ) ∪ *fn*(*P* ') ∪ *sub*(*B*1) ∪ *sub*(*B*2), *B*1 is obtained from *B* deleting the beta binder *β*^(*x* : Γ) and *B*2 is obtained from *B*' deleting the beta binder *β*^(*y* : Δ); (2) if the first beta binder of the list *B* is equal to one beta binder of the list *B*', then the function *BBMinCong* is recursively invoked on the beta-processes *B*1[*P* ] and *B*2[*P* '], where *B*1 and *B*2 are respectively obtained from *B* and *B*' deleting the equal beta binders; (3) if no correspondence between the first beta binder of *B* and one beta binder of *B*' exists, then the function returns *false*.

If only one of the beta binders lists *B* and *B*' is empty, then the function returns

*false*.

If both *B* and *B*' are empty, then the function *P iStdCong* is invoked on the pi-processes *P* and *P* '. In this case the function *BBMinCong* returns the result of *P iStdCong*(*P, P* ').

We notice that the decidability of the structural congruence over pi-processes is not only necessary condition but also sufficient condition for the decidability of the congruence ≡min.

*bb*

Now we analyze the congruence ≡std. The law *b.*2 regards parallelization with the inactive beta-process *Nil* and the laws *b.*3 and *b.*4 are associtivity and

*bb*

commutativity rules. The decidability of the congruence ≡std

*bb*

can be described

through a function *BBStdCong* : *B* × *B* → {*true, false*} defined by induction on the structure of beta-processes in the following way:

*BBStdCong*(*Nil, B*')= *false if* (1)

(

*true o.w.*

' ( *BBStdCong*(*Nil, Remove*(*B*''[*P* '']*,B*')) *if* (2)

*false o.w.*

*BBStdCong*(*B*1[*P*1]*,B* )=

*BBStdCong*(*Nil* || *B, B*')= *BBStdCong*(*B, B*')

*BBStdCong*(*B*1[*P*1] || *B, B*')=

*BBStdCong*(*B, Remove*(*B*''[*P* '']*,B*')) *if* (2)

*false o.w.*

(

1. ∃ *j, n* ∈ N+ *with* (*B*' = *B*1||·· · ||*Bn*) ∧ (*j* ≤ *n*) ∧ (*Bj* = *B*''[*P* ''])
2. ∃ *j, n* ∈ N+ *with* (*B*' = *B*1||·· · ||*Bn*) ∧ (*j* ≤ *n*) ∧ (*Bj* = *B*''[*P* '']) ∧ *BBMinCong*(*B*1[*P*1]*, B*''[*P* ''])

where if *B*' = *B*1||··· ||*Bn* e *n* = 1 then *B*' is a box or the inactive beta- process *Nil*. The function *Remove* : *B*[*P* ] × *B* → *B* is defined in the following way:

*Remove*(*B*[*P* ]*,Nil*)= *Nil*

' ' ( *Nil if B*'[*P* ']= *B*[*P* ]

*Remove*(*B*[*P* ]*, B* [*P* ]) =

'

*Remove*(*B*[*P* ]*, B*1||*B* )=

*B*'[*P* '] *o.w.*

( *B*' *if* (1)

*B*1

|| *Remove*(*B*[*P* ]*,B*') *o.w.*

* 1. (*B*1 = *B*''[*P* '']) ∧ (*B*''[*P* '']= *B*[*P* ])

If *B* and *B*' are composed by a different number of boxes, then they are not congruent and the function returns *false*. If there exists a bijection between the boxes *Bi*[*Pi*] of *B* and the boxes *B*' [*P* '] of *B*' such that for each corrispondence it

*j j*

is *Bi*[*Pi*] ≡min *B*' [*P* '], then the two beta-processes are congruent and the function

*bb j j*

returns *true*. Otherwise the function returns *false*.

Lemma 3.1 *The decidability of the structural congruence over pi-processes is a necessary and sufficient condition for the decidability of the structural congruence over beta-processes.*

# Structural congruence over pi-processes

The results on which we base part of our work are those obtained from J. Engel- friet e T.E. Gelsema in [[7](#_bookmark31)] and reported in Sect. 2.3. In fact, the decidability of the structural congruence over beta-processes strongly depends on the structural congruence over pi-processes. Moreover, the pi-processes are *small pi-Calculus* pro- cesses with an extended set of actions, and the structural laws for the structural congruence over pi-processes are the same ones for the structural congruence over small pi-Calculus processes. Thereafter, the results presented in [[7](#_bookmark31)] for the standard congruence ≡std and the replication free congruence ≡!fr can also be used in this context because they do not depend on the specific types of actions contained in the processes.

Lemma 4.1 *The congruences* ≡*std and* ≡*min are decidable for the subclass of beta-*

*bb bb*

*processes with replication restricted pi-processes.*

Proof. Immediate from the definition of functions *BBStdCong* and *BBMinCong*

and from the results presented in [[7](#_bookmark31)].

We notice that this result is valid for the qualitative version of Beta-binders. Now consider the stochastic extension of Beta-binders. The classical replication is re- placed with the guarded replication and hence the syntax and the structural laws for pi-processes are modified substituting respectively !*P* with !*π.P* and !*P* ≡ *P* | !*P* with !*π.P* ≡ *π.*(*P* | !*π.P* ) [3](#_bookmark7) . The Fig. [3](#_bookmark8) shows the congruences over guarded repli- cation pi-processes that we will consider in the remainder of the paper.

3 For semplicity in the remainder of the paper we omit the rate *r* in the prefixes beacause not important for our purpose.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| rule | | ≡min | ≡!fr | ≡std |
| (*α*) | *P*1 ≡ *P*2 if *P*1 and *P*2 are *α*-equivalent | + | + | + |
| (1*.*1) | *P* | *nil* ≡ *P* |  | + | + |
| (1*.*2) | *P*1 | *P*2 ≡ *P*2 | *P*1 | + | + | + |
| (1*.*3) | *P*1 | (*P*2 | *P*3) ≡ (*P*1 | *P*2) | *P*3 | + | + | + |
| (2*.*1) | (*νz*)(*νw*)*P* ≡ (*νw*)(*νz*)*P* | + | + | + |
| (2*.*2) | (*νz*)*P* ≡ *P* |  | + | + |
|  | if *x* /∈ *fn*(*P* ) |  |  |  |
| (2*.*3) | (*νz*)(*P*1 | *P*2) ≡ *P*1 | (*νz*)*P*2 |  | + | + |
|  | if *z* /∈ *fn*(*P*1) |  |  |  |
| (3*.*1) | !*π.P* ≡ *π.*(*P* | !*π.P* ) |  |  | + |

Fig. 3. Structural laws for the small *π*-calculus with guarded replication.

A process that only uses guarded replication is, by definition, replication restricted. Therefore, the standard structural congruence over guarded replication pi-processes is decidable. More precisely, this result is valid if we consider the replication struc- tural law !*P* ≡ *P* | !*P* , whereas it must be prooved if we consider the replication structural law !*π.P* ≡ *π.*(*P* | !*π.P* ).

In this paper we want to face the problem of decidability of structural congru- ence for guarded replication pi-processes from another point of view. In particular, we will consider the structure of pi-processes that only use guarded replication. In [[7](#_bookmark31)], the main difficulty in showing the decidability of ≡std for replication restricted processes is the treatment of replication, which allows a process to grow indefinitely and without particular structure in its number of subterms. A process that uses guarded replication, instead, allows a process to grow indefinitely in its number of subterms mantaining structure.

Given a generic pi-process *P* , this characteristic allows us to define a function that recognizes and eliminates all the expanded replication in *P* .

This function, that we call *Implosion*, is defined by induction on the structure of processes:

*Implosion*(*nil*)= *nil*

*Implosion*(!*π.P* ') =!*Implosion*(*π.P* ')

*Implosion*((*νx*)*P* ')= (*νx*)*Implosion*(*P* ')

*Implosion*(*P*0|*P*1)= *Implosion*(*P*0) | *Implosion*(*P*1)

' !*π.Q if* (1)

*Implosion*(*π.P* )= (

*π.Implosion*(*P* ') *o.w.*

(1) ∃ *j, n* ∈ N+ *with* (*P* ' = *P*1 | · · · | *Pn*) ∧ (*j* ≤ *n*) ∧ (*Pj* =!*π.R*) ∧

(*Q* = *Implosion*(*RemoveP I*(*Pj ,P* '))) ∧ (*Q* ≡!fr *Implosion*(*R*))

where if *P* ' = *P*1 | ··· | *Pn* and *n* = 1 then *P* ' is in the form *nil*, *π.R*,

!*π.R*, or (*νx*)*R*. The function *RemovePI* : P × P → P is defined in the following way:

*RemoveP I*(*P, P* ')=

*P*1 *if* (*P* ' = *P*0 | *P*1) ∧ (*P*0 = *P* )

>< *P*0 | *RemoveP I*(*P, P*1) *if* (*P* ' = *P*0 | *P*1) ∧ (*P*0 /= *P* )

8>

>>: *P* ' *o.w.*

*nil if* (1)

(1) ((*P* ' = *nil*) ∨ (*P* ' = *π.R*) ∨ (*P* ' =!*π.R*) ∨ (*P* ' = (*νx*)*R*)) ∧ (*P* = *P* ')

Since the processes have finite length the function *Implosion* ends.

Lemma 4.2 *Let P be a pi-process that only uses guarded replication. Then*

*Implosion*(*P* ) ≡*std P.*

Proof. Every substitution and modification that the function *Implosion* carries out on the structure of the process *P* comes from the recursive invocation of *Implosion*(*π.P* '). This substitutions and modifications are equivalent to the appli- cation of a sequence of structural laws *α*, 1*.*1, 1*.*2, 1*.*3, 2*.*1, 2*.*1, 2*.*3, 3*.*1. This laws are the structural laws of the congruence ≡std. For this reason *Implosion*(*P* ) ≡std *P* .

Now consider the subclass of guarded replication pi-processes that does not contain expanded replications. We call this subclass P*rp*.

Lemma 4.3 *Let P and Q be pi-processes belonging to* P*rp. Then P* ≡*std Q iff*

*P* ≡*!fr Q.*

Proof. (⇒) To show this implication we prove that in *P* ≡std *Q* the law 3*.*1 is never used. Assume that *P* is obtainable from *Q* by applying, for some subterm of *Q*, the law 3*.*1. This means that one of the two processes has a subterm in the form *π.*(*R* | !*π.R*). But the subterm *π.*(*R* | !*π.R*) expands the replication !*π.R* and this contradicts our initial assumption that *P* ∈ P*rp*. Therefore, the law 3*.*1 is never used and the implication is true.

(⇐) Since the structural laws of the congruence ≡!fr are a subset of the structural laws of the congruence ≡std then *P* ≡!fr *Q* implies *P* ≡std *Q*.

Lemma 4.4 *Let P and Q be guarded replication pi-processes. Then P* ≡*std Q iff*

*Implosion*(*P* ) ≡*!fr Implosion*(*Q*)*.*

Proof. (⇒) Since *Implosion*(*P* ) ≡std *P* ≡std *Q* ≡std *Implosion*(*Q*) (using Lemma [4.2](#_bookmark9)) we obtain that *Implosion*(*P* ) ≡std *Implosion*(*Q*). Since the pi- processes *Implosion*(*P* ) and *Implosion*(*Q*) does not contain expanded replica- tion, we have that *Implosion*(*P* ) ≡std *Implosion*(*Q*) (using Lemma [4.3](#_bookmark10)) implies *Implosion*(*P* ) ≡!fr *Implosion*(*Q*).

(⇐) The structural laws of congruence ≡!fr are a subset of the structural laws of the congruence ≡std. For this reason *Implosion*(*P* ) ≡!fr *Implosion*(*Q*) im- plies *Implosion*(*P* ) ≡std *Implosion*(*Q*). For the Lemma [4.2](#_bookmark9) we have that *P* ≡std

*Implosion*(*P* ) ≡std *Implosion*(*Q*) ≡std *Q* and therefore, for transitivity, we have that *P* ≡std *Q*.

We notice that the function *Implosion* is intrinsically based on the congruence relation ≡!fr. So, we can assert that there exists a procedure that allows to verify the standard congruence over guarded replication pi-processes using only the laws of the replication free congruence. Therefore, this procedure is effectively decidable only if the replication free congruence is decidable. In [[7](#_bookmark31)] (Theorem 3.10) Engelfriet proves that

*P* ≡!fr *Q* ⇐⇒ *swf* (*P* ) ≡*α swf* (*Q*)

where, due to some initial conventions, with ≡*α* he means ≡min. For show- ing that ≡!fr is really decidable, we prove that the problem *P* ≡min *Q* is equivalent to an isomorphism problem over labelled directed acyclic graphs (lDAGs), that we know to be a decidibile problem.

Let *P* be a pi-process. We define a procedure that permits to construct the lDAG, denoted with *GS*(*P* ), that we will use in the next proof.

Definition 4.5 Let *P* be a pi-process. The graph *GS*(*P* ) is built from the syntax tree of *P* applying the following transformations:

1. the multiple composition of binary parallels are replaced with a unique n-ary parallel (Fig. [4](#_bookmark14));
2. the restriction sequences are transformed as shown in Fig. [5](#_bookmark15);
3. the output nodes, that have label *x*⟨*n*⟩, are replaced with a sequence of two nodes where the first has label *x* and the second has label ⟨*n*⟩ (Fig. [6](#_bookmark16));
4. An edge is added from each node that contains a binding occurrence for a name to all the nodes that contains names binded to this occurrence (Fig. [7](#_bookmark17));
5. Every name that binds something is replaced with 0 and every binded name is replaced with 1.

Without loss of generality we assume that 0 and 1 do not belong to the set of names N.

The *GS* graph can be built in polynomial time and is essential for the treatment of the *α*-conversion and the commutativity of restrictions. Let *P* = (*νx*)(*νy*)(*a*(*x*)*.nil* | *y*(*z*)*.b*⟨*z*⟩*.x*⟨*m*⟩*.nil*). Fig. [8](#_bookmark18) shows the building procedure of the graph *GS*(*P* ). With ∼= with denote the classical isomorphism relation between lDAGs, where the ismorphism is a bijection of nodes that mantains labels and adi- acency properties.

Lemma 4.6 *Let P and Q be pi-processes. Then P* ≡*min Q iff GS*(*P* ) ∼= *GS*(*Q*)*.*

Proof. Let *R* be a pi-process. Then the nodes of the graph *GS*(*R*) = (*VR, ER*) are enumerated with a pre-order starting from the root of the cover tree of the

J

*P*

*ccjc ,,*z*z*

*cc*| *,,,*

J

*cc ,,*

*c* | *,,*

*R*

*jc* z*z*

*,* *,,*

*,,,,* *ee ,, ¸¸¸¸*

*,,,e* | *¸,,¸¸¸*

*e* z*z ¸¸*z*˛*

*eee*

| *,*

*,,,*

*P*

*R*

*S*

*T*

*S*

*e* z*z*

*T*

Fig. 4. Binary parallel composition transformation.

J

(*νx*)

J

(*νy*)

J

(*νz*)

J

(*νx*)*¸¸* (*νy*) (*νz*)

,*s* J

*¸*

*¸*

*¸*

*¸*

*¸*

*¸*z

*¸*

*¸*

*¸*

*¸*z

J,

Fig. 5. Restriction sequences transformation.

J

*x*⟨*n*⟩

J

J

*x*

J

⟨*n*⟩

J

Fig. 6. Output node transformation.

J

(*νx*)

J

*x*(*n*)

J

*z*(*x*)

J

*x*(*m*)

J

J

(*νx*)

*x*(*n*)

z*˛*J

J

*z*(*x*)

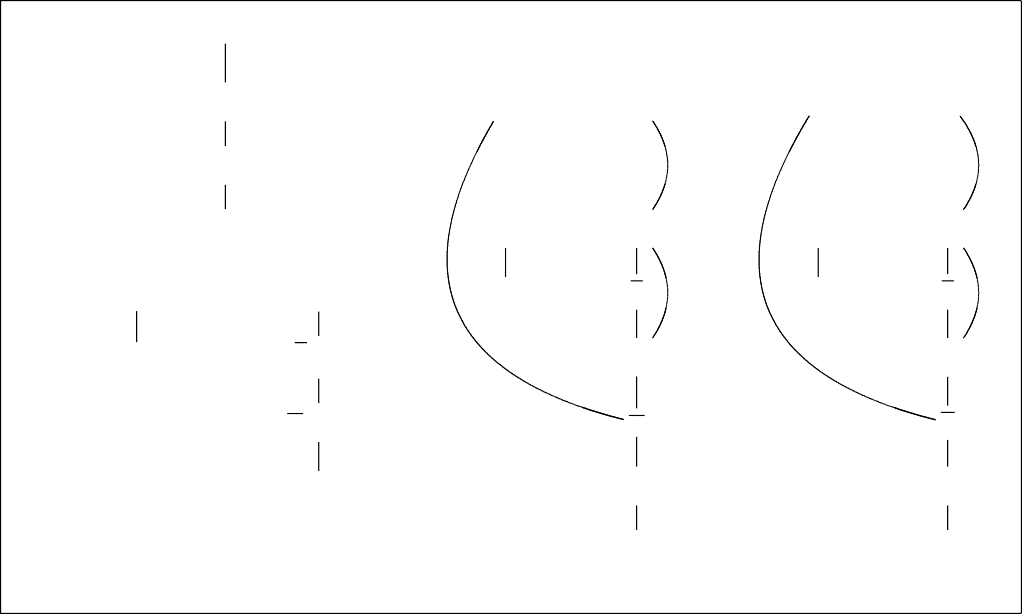
*x*(*m*)

J

J

Fig. 7. Edge addition. Notice that there is not edge between the restriction (*νx*) and the node *x*(*m*).

graph, without considering the added edges (Fig. [9](#_bookmark19)). (⇒) We assume by hypothesis that *P* ≡min *Q*. This means that *P* is obtainable from *Q* (and viceversa) by applying, in *Q*, a sequence *r*0*, ..., rn* of structural laws (we assume that *ri* supplies the information about where to apply the law in *Q*). Notice that we obtain the process *Qi* ≡min *Q* applying in *Q* the law *ri*. The construction of an isomorphism



*a*)

*b*)

J

*¸*

*¸*

*¸*

*¸*

*c*)

*,*

*¸*

*,,,,*

J

(*νy*)

J

(*νx*)*¸¸*

, *s* v*z*

(*νx*)

(*νy*)

, *s ,*v*z*

*¸¸*z

0 *¸¸*

*¸¸¸*z

0

, *¸¸¸¸*

| *j*

| *j*

| *¸¸¸¸¸*

*a*(*x*)

*s* v*z* *r*

*y*(*z*)

*,,,,*

*a*(*x*)

,*s* v*z* *r*

, *s ¸*z

*a*(*x*) *y*(*z*)

J

*nil*

J

*nil*

J

*b*

J

⟨*z*⟩

J

*nil*

J

*b*⟨*z*⟩

J

*x*⟨*m*⟩

J

*nil*

1(0)

J

*b*

J

⟨1⟩

*x*

J

⟨*m*⟩

J

*nil*

zJ

J

1

J

⟨*m*⟩

J

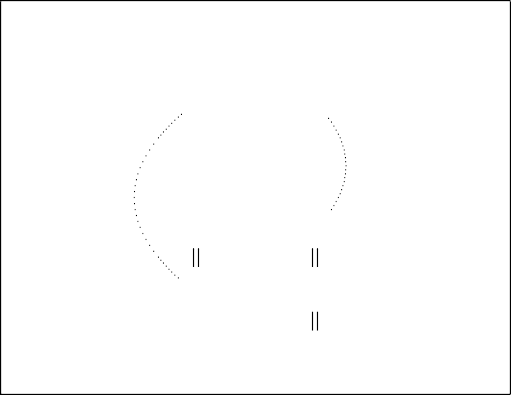
*nil*

z

Fig. 8. Transformation of the syntax tree of the pi-process *P* = (*νx*)(*νy*)(*a*(*x*)*.nil* | *y*(*z*)*.b*⟨*z*⟩*.x*⟨*m*⟩*.nil*) in

*GS*(*P* ). In *a*) is shown the syntax tree of *P* . In *b*) is shown the application of the transformations 1,2,3 and

4. In *c*) the transformation is completed.



*c*J,˜¸1` *,*z,*,,*

J,˜¸2` z,*c r* J,˜¸` z,

,*c****cc****c*

*,****,,*** z

*,,,,****,***

9

***,***

*c*J,˜¸3` *,*z,*,,*

z

*j*

*c*

*c*

*c*

*c*

J,˜¸4` z,*c r* J,˜¸6` z,

,*c****cc****c*

*,****,,*** z *r*

J,˜¸5` z,

zc *z*

c*z*

J,˜¸7` z,

c*z*

J,˜¸8` z,

Fig. 9. Example of graph node enumeration. The double lined arrows show the cover tree of the graph. The dotted arrows represent the added edge that we do not consider.

*φi* between *GS*(*Q*) and *GS*(*Qi*) depends on the structural law *ri* applied. We have three cases: (1) Suppose that *Qi* is obtained from *Q* by applying the law (2*.*1) on a subterm (*νx*)(*νy*)*Q*' of *Q*. Therefore, the only difference between *Q* and *Qi* is that in *Qi* the subterm (*νx*)(*νy*)*Q*' appears in the form (*νy*)(*νx*)*Q*'. Let *n*1 and *n*2 be the nodes in *GS*(*Q*) that represent respectively the restrictions (*νx*) and (*νy*) of the subterm (*νx*)(*νy*)*Q*'. In the graph *Qi* the representation is inverted. In fact, *n*1 represents (*νy*) while *n*2 represents (*νx*). Let *φi* be the mapping between the nodes of *GS*(*Q*) and *GS*(*Qi*) such that for each node *n* ∈ *VQ* with *n* /∈ {*n*1*, n*2} is *φi*(*n*) = *n* and such that *φi*(*n*1) = *n*2 and *φi*(*n*2) = *n*1. *φi* is an isomorphism because, for the *GS* construction, the nodes *n* ∈ *VQ* and *φi*(*n*) ∈ *VQi* have the same

labels and for each edge (*n, n*') ∈ *EQ* it is (*φi*(*n*)*, φi*(*n*')) ∈ *EQi* .

* 1. Suppose that *Qi* is obtained from *Q* by applying the law (1*.*2) on a subterm *Q*'|*Q*'' of *Q*. Thereafter, the only difference between *Q* and *Qi* is that in *Qi* the subterm *Q*'|*Q*'' appears in the form *Q*''|*Q*'. Let *n*0 and *n*1 be the nodes in *GS*(*Q*) that represent respectively the root node of the subgraph *GS*(*Q*') and the root node of the subgraph *GS*(*Q*''). In *GS*(*Qi*) the representation is inverted. In fact, *n*1 represents the root node of the subgraph *GS*(*Q*'') while *n*2 represents the root node of the subgraph *GS*(*Q*'). Let *φi* be the mapping between the nodes of *GS*(*Q*) and *GS*(*Qi*) such that for each node *n* ∈ *VQ*, with *n* /∈ {*GS*(*Q*')*, GS*(*Q*'')}, it is *φi*(*n*)= *n* and such that for each node *n*1 + *i*, with *i* ≥ 0 and *n*1 + *i* ∈ *GS*(*Q*'), and for each node *n*2 + *j*, with *j* ≥ 0 and *n*2 + *j* ∈ *GS*(*Q*''), it is *φi*(*n*1 + *i*) = *n*2 + *i* and *φi*(*n*2 + *j*)= *n*1 + *j*. Also in this case *φi* is an isomorphism because, for the *GS* construction, the nodes *n* ∈ *VQ* and *φi*(*n*) ∈ *VQi* have the same labels and for each edge (*n, n*') ∈ *EQ* it is (*φi*(*n*)*, φi*(*n*')) ∈ *EQ* .

*i*

* 1. If *Qi* is obtained from *Q* by applying *α*-conversion or the law (1*.*3) then the isomorphism *φi* is the identity *id* because, for the *GS* construction, the graphs *GS*(*Q*) and *GS*(*Qi*) are equal.

Being the isomorphism relation closed under composition, then the composition *φ*0 ◦ ··· ◦ *φn* is an isomorphism and precisely the isomorphism between *GS*(*Q*) and *GS*(*P* ) we wanted.

(⇐) Let *P* and *Q* pi-processes such that *GS*(*P* ) ~= *GS*(*Q*). We prove the implication by contradiction assuming that *P* /≡min *Q*. The proof is by induction on the

structure of the processes *P* and *Q*.

(Induction base) Let *P* = *nil*. Since *P* /≡min *Q* then *Q* /= *nil* and obviusly *GS*(*P* ) /~= *GS*(*Q*). (Case *P* = *x*(*y*)*.R*) if *Q* /= *x*(*y*)*.S* then *GS*(*P* ) /~= *GS*(*Q*) because in *Q*, by the graph *GS* construction, does not exists a node with the label and adiacency properties of the node that represent *x*(*y*) in *P* . Otherwise, if *Q* = *x*(*y*)*.S* we have that *R* /≡min *S*. By inductive hypothesis we obtain that *GS*(*R*) /~= *GS*(*S*) and since for each isomorphism the node that represent *x*(*y*) in *P* should be mapped into the node that represent *x*(*y*) in *Q*, it turns out that a total mapping does not exists and hence *GS*(*P* ) /~= *GS*(*Q*). (Case *P* = *x*⟨*y*⟩*.R* and *P* =!*π.R*) Similar to the previous case. (Case *P* = *R*1 | ··· | *Rn*) Let *P* = *R*1 | ··· | *Rn* (we intend all the processes in a form like (··· ((*R*1 | *R*2) | *R*3) | ··· | *Rn*)) such that *Ri* is not a parallel composition. If *Q* /= *S*1 | ··· | *Sn* (with *Si* be not a parallel composition) then, by the graph *GS* construction, *GS*(*P* ) /~= *GS*(*Q*). Otherwise, we have that E*Ri* such that 6*Sj* it is *Ri* /≡min *Sj* and therefore, by inductive hypothesis, 6*Sj* it is *GS*(*Ri*) /~= *GS*(*Sj*). Since all the subgraphs *Ri* in *P* and *Sj* in *Q* are disjunct we obtain that *GS*(*P* ) /~= *GS*(*Q*). (Case *P* = (*νx*1) ··· (*νxn*)*R*) Let *P* = (*νx*1) ··· (*νxn*)*R* (with *R* not in the form (*νx*)*R*'). if *Q* /= (*νy*1) ··· (*νyn*)*S* (with *S* not in the form (*νy*)*S*') then, by the graph *GS* construction, *GS*(*P* ) /~= *GS*(*Q*). Otherwise, we have that for each permutation of restrictions (*νy*1) ··· (*νyn*) and *α*-conversion it is *Q* = (*νx*1) ··· (*νxn*)*T* with *T* /≡min *R* and thus, by inductive hypothesis, *GS*(*R*) /~= *GS*(*T* ). Since, by the graph *GS* construction, the nodes that represents (*νx*1) ··· (*νxn*) should be mapped into the nodes that represents

(*νy*1) ··· (*νyn*) we have that *GS*(*P* ) ~/= *GS*(*Q*).

This contradict the assumption that *GS*(*P* ) ~= *GS*(*Q*) and therefore the impli- cation is valid.

The lDAG isomorphism problem [[10](#_bookmark34),[1](#_bookmark25)] is placed in the complexity class GI, which contains all the problems equivalent to the general graph isomorphism problem. The class GI is a particular complexity class. In fact, no polinomialy resolution algorithm for the problems in GI has been still found and it is not known if they are or not NP-complete. However, the congruence ≡min is decidable.

Theorem 4.7 *Let P and Q be guarded replication pi-processes. Then the evalua- tion of P* ≡*std Q is decidable.*

Proof. Using the Lemma [4.4](#_bookmark11), the Theorem 3.10 in [[7](#_bookmark31)] and the Lemma [4.6](#_bookmark13) we have that

*P* ≡std *Q*

⇐⇒

*Implosion*(*P* ) ≡!fr *Implosion*(*Q*)

⇐⇒

*swf* (*Implosion*(*P* )) ≡min *swf* (*Implosion*(*Q*))

⇐⇒

*GS*(*swf* (*Implosion*(*P* ))) ~= *GS*(*swf* (*Implosion*(*Q*)))

and therefore, for transitivity, we can conclude that

*P* ≡std *Q* ⇐⇒ *GS*(*swf* (*Implosion*(*P* ))) ~= *GS*(*swf* (*Implosion*(*Q*)))

where *GS*(*swf* (*Implosion*(*P* ))) ~= *GS*(*swf* (*Implosion*(*Q*))) is a decidable problem.

Corollary 4.8 *Let B*[*P* ] *and B*'[*P* '] *be boxes where P and P* ' *are guarded replication pi-processes. Then the evaluation of B*[*P* ] ≡*min B*'[*P* ] *is decidable.*

*bb*

Proof. Immediate from the definition of the function *BBMinCong* and the The- orem [4.7](#_bookmark20).

Corollary 4.9 *Let B e B*' *be beta-processes composed by boxes with guarded repli- cation pi-processes. Then the evaluation of B* ≡*std B*' *is decidable.*

*bb*

Proof. Immediate from the definition of the function *BBStdCong* and the Corol- lay [4.8](#_bookmark21).

# Generalization

Although we think that the restricted beta binder *well-formedness* definition, pre- sented in Sec.[3](#_bookmark4), gives enough expressive power, in this section we briefly show that

the congruence ≡min for the stochastic semantics of Beta-binders is decidable also considering the classical *well-formedness* definition, given in Sect.[2](#_bookmark2).

*bb*

Let *B*[*P* ] and *B*'[*P* '] be boxes where *P* and *P* ' are guarded replication pi- processes. We assume the existence of an injective, decidable and polynomial func-

^

tion ) : *β* × T → S where T is the set of beta binder types and S is a set

of strings such that 0 /∈ S and S ∩ L = ∅ (we assume L be the set of the pos- sible labels generated by the *GS* construction) . For deciding *B*[*P* ] ≡min *B*'[*P* '] we construct the lDAGs *GS*(*Q*) and *GS*(*Q*'), where *Q* = *swf* (*Implosion*(*P* )) and *Q*' = *swf* (*Implosion*(*P* ')), we interpret the beta binders lists *B* and *B*' as a set of top level restrictions and we put them on the top of the constructed lDAGs, modifying the binded nodes as described in Def.[4.5](#_bookmark12). The only difference is that a

*bb*

^

node that represents an elementary beta binder *β*(*x* : Γ) is labelled with the result

of the function *β*^*,* Γ) instead of 0. We call the obtained graphs *GS*(*B*[*Q*]) and

*GS*(*B*'[*Q*']). In Fig.[10](#_bookmark24) an example is given.

The *GS* graphs can be built in polynomial time and since the graphs *GS*(*B*[*Q*]) and *GS*(*B*'[*Q*']) differ from *GS*(*Q*) and *GS*(*Q*') only in the number and labels of nodes that represent restrictions, the Lemma [4.6](#_bookmark13) continues to hold and thus we have that:

Corollary 5.1 *Let B*[*P* ] *and B*'[*P* '] *be boxes where P and P* ' *are guarded replication*

*pi-processes. Then B*[*P* ] ≡*min*

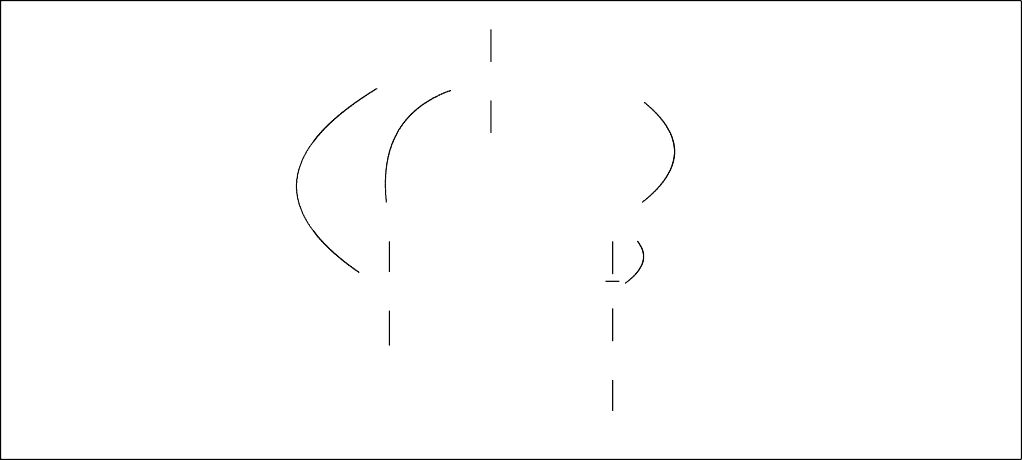
*bb*

*B*'[*P* '] *iff GS*(B[*Q*]) ~= *GS*(B'[*Q*'])*, where Q* =

*swf* (*Implosion*(*P* )) *and Q*' = *swf* (*Implosion*(*P* '))*.*

The function *BBStdCong* and the Corollaries [4.8](#_bookmark21) and [4.9](#_bookmark22) can be simply rede- fined considering the graph *GS* construction.





0 *¸¸¸*

*..s. .*

*.*

*.*

*.*

*.*

*¸*

*¸*

J

*¸*

*¸*

*¸*

*¸*

*¸*

z

*˛*

*β,* Γ)

*¸*

*βh,* Δ)

*¸*

*¸*

*¸* J

*¸*

z *.s.*

*.*

*.*

*.*

*.*

*.*

J*z*, *s*

1(0)

zJ

1(0)

J

*nil*

| *¸¸¸¸¸*

*¸¸¸*z,

1(0)

J, *s*

1

J

⟨*m*⟩

J

*nil*

Fig. 10. lDAG *GS* for the box *β*(*x* : Γ)*βh*(*y* : Δ)[(*νz*)(*x*(*a*)*.z*(*a*)*.nil* | *y*(*b*)*.b*⟨*m*⟩*.nil*)].

# Conclusions

We proved the decidability of the structural congruence used in [[3](#_bookmark27)] to define the stochastic semantics of Beta-binders. The proof is constructive so that we have suggestions for possible implementations of the calculus.

# References

1. K.S. Booth and C.J. Colbourn. Problems polynomially equivalent to graph isomorphism. Technical Report CS-77-04, Department of Computer Science, University of Waterloo, Ontario, Canada, 1977.
2. L. Cardelli. Brane calculi. In *Proc. of Computational Methods in Systems Biology*, volume 3082 of

*LNCS*, pages 257–278, 2005.

1. P. Degano, D. Prandi, C. Priami, and P. Quaglia. Beta-binders for biological quantitative experiments. In *4rd Int. Workshop on Quantitative Aspects of Programming Languages (QAPL 06)*, 2006. to appear.
2. J. Engelfriet. A multiset semantics for the pi-calculus with replication. *Theor. Comput. Sci.*, 153(1&2):65–94, 1996.
3. J. Engelfriet and T.E. Gelsema. Multisets and structural congruence of the pi-calculus with replication.

*Theor. Comput. Sci.*, 211(1-2):311–337, 1999.

1. J. Engelfriet and T.E. Gelsema. Structural congruence in the pi-calculus with potential replication. Technical Report 00-02, Leiden Institute of Advanced Computer Science, Leiden University, The Nederlands, 2000.
2. J. Engelfriet and T.E. Gelsema. The decidability of structural congruence for replication restricted pi-calculus processes. Technical Report 04-07, Leiden Institute of Advanced Computer Science, Leiden University, The Nederlands, 2004.
3. J. Engelfriet and T.E. Gelsema. A new natural structural congruence in the pi-calculus with replication.

*Acta Inf.*, 40(6-7):385–430, 2004.

1. D.T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Phys. Chem.*, 22:403–434, 1976.
2. J. K¨obler, U. Sch¨oning, and J. Tor´an. *The Graph Isomorphism Problem: its structural complexity*. Birkhauser, 1993.
3. R. Milner. The polyadic pi-calculus: a tutorial. Technical Report ECSLFCS-91-180, Computer Science Department, University of Edinburgh, UK, 1991.
4. C. Priami and P. Quaglia. Beta binders for biological interactions. In *Proc. of Computational Methods in Systems Biology*, volume 3082 of *LNCS*, pages 20–33, 2005.
5. C. Priami and P. Quaglia. Operational patterns in beta-binders. *T. Comp. Sys. Biology*, 1:50–65, 2005.
6. C. Priami, A. Regev, E. Shapiro, and W. Silvermann. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.
7. A. Regev, E.M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.
8. A. Regev and E. Shapiro. Cells as computation. *Nature*, 2002.