Efficient symbolic computation of process expressions

Benoît Fraikin *, Marc Frappier

GRIL, Département d’Informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada, J1K 2R1

A R T I C L E   I N F O

Article history:
Received 27 February 2006
Received in revised form 13 February 2009
Accepted 14 February 2009
Available online 27 February 2009

Keywords:
Trace-based specifications
Black-box specifications
Process algebra
Information systems
Symbolic computation
Interpreter

A B S T R A C T

This paper describes three optimization techniques for the EB 3 process algebra. The optimizations are expressed in a new deterministic operational semantics which is shown to be trace-equivalent to a traditional non-deterministic operational semantics. Internal action transitions are eliminated by an efficient preruntime analysis of the structure of a process expression. Execution environments are used to optimize variable instantiation using lazy evaluation. Non-determinism is eliminated by returning a choice between possible transitions. This new operational semantics is implemented in the EB 3 PAI process algebra interpreter to support the EB 3 method. The goal of this method is to automate the development of information systems using, among other mechanisms, efficient symbolic computation of process expressions.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

The project EB 3 PAI (which stands for EB 3 Process Algebra Interpreter) is part of the APIS research project [1]. The objective of APIS is to support the rapid development of information systems (IS) from formal specifications by using code generation and efficient specification execution. APIS is based on the EB 3 (Entity-Based Black Box) method [2], which was specifically designed for IS specification.

In our viewpoint, an IS is a software system that helps an organization to collect and manipulate all its relevant data. IS are used in almost all areas of human activities where information must be stored, exploited and analyzed. Typical examples include management IS (e.g. accounting, human resource and production) which are used to support the business process of an organization.

An information system is generally characterized by large persistent data structures which are modified or queried by several users in concurrency. The distinctive characteristics of IS consist in managing complex relationships between data structures, of calculations involving several data structures, of processing large volume of data, and of preserving data integrity through concurrent updates. IS typically have little hard real-time constraints. Modern database management systems provide concurrency control mechanisms which simplify IS development.

An EB 3 specification consists essentially of two parts: (i) a process expression, called main, which defines the valid input traces of the IS, (ii) input–output (I–O) rules which assign an output to each input trace. The semantics of an EB 3 specification is given by a relation R defined on I + × O, where I + denotes the set of non-empty traces defined over input set I and O denotes an output set. Hence, process expression main defines the domain of R; in EB 1, a process algebra is used solely to define the inputs.

The EB 3 process algebra is inspired from regular expressions, CSP [3], CCS [4], ACP [5] and LOTOS [6]. For instance, the process expression a . (b | c), where | and . correspond to the well-known regular expression operators choice and
concatenation, denotes the input traces \( \{a, ab, ac\} \). Given some input–output rules (omitted here), a relation \( R = \{(a \rightarrow o_1), (ab \rightarrow o_2), (ac \rightarrow o_3)\} \) is associated to this specification. This specification means that the IS, from its initial state, must accept user input \( a \) and provide output \( o_1 \); if some other input is submitted by the user, it must be rejected and the user must be informed by an appropriate error message [7]. After accepting \( a \), the IS must accept user input \( b \) and produce output \( o_2 \), or accept \( c \) and produce output \( o_3 \). A system is said to be correct with respect to a specification \( R \) if it can accept all input traces \( t \) in the domain of \( R \) (i.e. a trace of \( \text{main} \)) and produce, for each \( t \), an output \( o \) such that \( (t \rightarrow o) \in R \).

We are currently working on effective tools to support \( \text{EB}^3 \). The tool \( \text{EB}^3 \text{PAI} \) is an interpreter for \( \text{EB}^3 \) process expressions. \( \text{EB}^3 \text{PAI} \) relies on several optimization techniques to handle non-deterministic process expressions, internal actions and quantified operators like choice and parallel composition with synchronization.

\( \text{EB}^3 \text{PAI} \) executes an action by applying the transition rules of an operational semantics (in the Plotkin style used for CCS [4]) defined for the \( \text{EB}^3 \) process algebra. Basically, \( \text{EB}^3 \text{PAI} \) efficiently computes on the fly a proof of a transition \( E \xrightarrow{a} E' \) to determine whether process expression \( E \) can accept user input \( \sigma \). If \( \sigma \) can be accepted, then \( E' \) becomes the resulting process expression on which the next input is applied; otherwise \( \sigma \) is discarded and the current process expression does not change. Hence, \( \text{EB}^3 \text{PAI} \) does not generate executable code to execute a process expression; rather, it is itself an abstract machine that executes a process expression. The state of the abstract machine is the abstract syntax tree (AST) of \( E \).

The original operational semantics of the \( \text{EB}^3 \) process algebra, proposed in [2], is not adequate for an efficient symbolic computation. This paper proposes a new set of transition rules on which \( \text{EB}^3 \text{PAI} \) is based. The transition rules of [2] suffer from three main problems.

First, they allow non-determinism, which means that an action can sometimes be executed by several transitions, leading to different process expressions. Since the \( \text{EB}^3 \) process expression \( \text{main} \) defines the traces that must be accepted by the IS, an interpreter must find the appropriate execution path to accept a given trace. A naive interpreter based on the rules of [2] must sometimes backtrack and try other execution paths for past (accepted) actions, in order to accept a new one. Note that we are dealing here with process expression non-determinism, which is distinct from \( \text{I–O rules non-determinism} \). \text{I–O} rules allows for the specification of several outputs for a given input trace, which is sometimes desirable for IS specification. For instance, in a travel agency, the choice of the ordering for a list of flights which match a set of criteria maybe non-deterministic.

Second, internal actions, which are not visible to the environment, can also require the interpreter to backtrack, or they can induce infinite loops (divergence) when trying to execute an action.

Third, the rules of [2] use syntactic substitution on the AST, which means that every occurrence of a variable is replaced by its substituted term. This can lead to significant overhead in transition computation and high memory usage for large interleave quantifications, because each interleaved process differs from the others only in the substituted text (cf. Fig. 4 in Section 4.2).

The proposed set of rules is proved to be trace-equivalent to the one defined in [2]. These new rules are more complex, because they are meant to be used for efficient execution. They have been implemented in \( \text{EB}^3 \text{PAI} \) in order to evaluate their efficiency from an experimental standpoint, taking into account practical implementation issues like persistency of large ASTs, large quantification sets, and memory usage in order to minimize redundancy.

For various patterns of IS (Section 5) which are derived from the structure of the business model (entity-relationship model), \( \text{EB}^3 \text{PAI} \) can execute an action in linear time with respect to the size of the specification (i.e. the number of terms and operators in the process expression) and logarithmic time with respect to the number of entities of an entity type in the business model. The current version of \( \text{EB}^3 \text{PAI} \) is implemented in Java; it uses the OODBMS ObjectStore PSE PRO (which is also implemented in Java) to handle the persistency of its internal state (i.e. an AST) and large collections of objects.

A companion paper [8] proposes algorithms to efficiently execute large interleave quantifications, which are fundamental components of an IS. Large interleave quantifications are used to model the entities (i.e. instances or objects) of an entity type (i.e. class) and the relationships between entity types. An entity type in an IS can easily contain thousands of entities.

The API framework supports the \( \text{EB}^3 \) method; it includes \( \text{EB}^3 \text{PAI} \) and other components which are illustrated in Fig. 1. A complete \( \text{EB}^3 \) specification includes five elements represented in the upper part of the figure. The user interacts with the IS through a web interface generated by DCI-WEB [9] from a formal specification of the user interface interaction. The web interface calls \( \text{EB}^3 \text{PAI} \) to determine if the user input is valid. \( \text{EB}^3 \text{PAI} \) tries to execute this input event on the process expression. If it succeeds, it calls an update program which has been generated by \( \text{EB}^3 \text{TG} \) [10] to update a relational database that contains the value of IS entity attributes and then calls a query program to compute the output associated to this input event; if \( \text{EB}^3 \text{PAI} \) fails to accept the input event, it reports an informative message to explain the error to the user. Entity attributes are formally specified by recursive functions on the set of traces accepted by \( \text{main} \). Entity types are defined by an entity-relationship (ER) diagram. Component \( \text{EB}^3 \text{IO} \) is under development.

The \( \text{EB}^3 \) process algebra differs in a number of aspects from traditional process algebras, in order to streamline the specification of IS. The first important distinction, as illustrated in Fig. 1, is that outputs are not specified using the process algebra, but from recursive functions defined on input traces. A process algebra provides operators to define ordering constraints on actions that can communicate with the environment; it does not include state variables like those found in a state-machine language like B [11] or Z [12]. It has been recognized by several authors that data management is hard to specify using solely a process algebraic approach. A number of proposals were made to combine a process algebra with a state-machine specification language to manage data. The key idea is that process algebra operators define the ordering of actions; state variables and state-machine operations manage the data. The CSP \( \Box B \) [13] approach combines a CSP specification with a B specification [11]. CSP actions are matched with B operations; when a CSP action is executed,
the corresponding B specification is also executed to update the B state variables. The csp2B [14] approach offers a similar combination, but the CSP specification is automatically translated into a B specification to form a single B specification. Circus [15] integrates Z with CSP in a single language. Z schemas provide definitions of state variables. Z operation schemas can be used as actions in a CSP process expression. CSP-OZ [16] offers a combination of CSP and Object-Z. For a thorough comparison of these approaches, see [17,8].

The design choice in EB3 of using recursive functions on the system trace, instead of a state-machine specification language like B or Z, is driven by IS characteristics. IS are highly data-oriented. It is important to easily understand how the values of entity attributes evolve. A recursive function provides a central, encapsulated definition of the value of an attribute in terms of input events received. This style is orthogonal to the state-machine-oriented style, where operations are defined by modifying state variables. Hence, to understand how the value of an entity attribute evolves over time, one must look at all operations of a state machine where a variable is modified. EB3 recursive functions can be translated into a B machine specification to benefit from both styles [18].

On the syntactic level, the EB3 process algebra differs from CSP [3] on several aspects. As in ACP [5] and μCRL [19], an action (e.g., a) constitutes an elementary process expression in EB3. In CSP, actions are not elementary process expressions; they must be combined with the action prefix operator to form a process expression (e.g., a → STOP). EB3 offers a smaller set of operations than CSP. EB3 includes a single sequence operator and a single choice operator |, like those used in regular expressions, ACP and μCRL. CSP offers two operators for sequential composition (action prefix → and sequential composition “;”), which is used in combination with the process SKIP; it offers three operators for choice: |, which applies only to action prefix expressions, \, an external choice, and ⊓, a non-deterministic (internal) choice. EB3 includes parallel operators (drawn from Hoare’s original definition of CSP [3]). Roscoe’s version of CSP [20] also includes (\Δ) (called the generalized parallel and denoted \)). Finally, EB3 offers the Kleene closure operator * from regular expressions, which is very useful for writing concise IS specification, but not offered in CSP. EB3 offers quantified versions of | and (\Δ), as does CSP (called indexing or replication). EB3 leaves out several other features of CSP, in particular elementary processes SKIP and STOP, successful termination event \, event hiding operator “\", but includes an internal action λ, which corresponds to \ in CCS and to ε in regular expressions. Successful termination is denoted by the special process || in EB3. These syntactic choices were made in order to foster conciseness in specification writing. For instance, the simple EB3 process expression a \cdot (b | c) \cdot d has the same traces as the following CSP expression:

(a → (b → SKIP | c → SKIP)); (d → STOP)

The next example illustrates the simplicity of Kleene closure. The EB3 process expression a \cdot (b | c)\cdot d has the same traces as the following CSP expression:

(a → (\mu P . ((b → P | c → P) \cdot SKIP))); (d → STOP)

There exist two classes of tools for process algebras: simulators (also called animators) and model checkers. Simulators allow users to execute a process expression for specification validation purposes (e.g., a walkthrough to explore the behavior for typical use cases). Classical examples include PROBE [21] and CIA [22] for CSP, the simulator in the μCRL tool set [23], and CADP’s OCIS [24] for LOTOS. They are usually based on an operational semantics of the process algebra, which enables one to compute the possible transitions of a process expression. Model checkers verify that a process expression satisfies a
given property by exploring its entire transition system, something that a simulator does not do, since the user is orienting the execution by exploring a particular execution path. Examples are FDR2 [25] for CSP, the Concurrency Workbench [26] for CCS, ProB [27] for a combination of CSP and B, the model checking tools in the μCRL tool set [23], LTSA [28] for FSP, and CADP’s EVALUATOR [24] for LOTOS.

EB^3PAI is more in the class of simulators. However, it must handle additional requirements since we want to use it to implement specifications. A simulator will typically execute a specification in a step by step manner. For instance, the process expression \( (a \cdot b) \mid (a \cdot c^* \cdot d) \) is non-deterministic when considering a traditional operational semantics [2,20]. This means that a simulator will offer the user to execute \( a \) using either the left or the right operand of \( \cdot \). If the user picks the right operand, then the simulator will refuse to execute \( b \) in the next step. Moreover, after executing \( a \) on the right operand, the simulator will offer to execute internal action \( \lambda \) (to exit the Kleene closure) or to execute \( c \); it does not offer to execute \( d \) at this point. An implementation of this EB^3 specification should not do that. It must be able to execute actions \( a \), \( d \) without asking the user to pick the appropriate branch of execution or to trigger internal actions. These are the problems that this paper is addressing: automatic handling of non-determinism and automatic execution of internal actions. EB^3PAI does not (and cannot) generate the entire transition system of a specification, because it is huge even for the most simple IS specifications.

Compilation approaches, which automatically translate a process expression into executable code in a high-level language, are orthogonal to the symbolic computation strategy of EB^3PAI. For instance, JCircus [29,30] translates a Circus [31] specification into a Java program using J CSP [32]. J CSP is a library that aims to provide an efficient framework to implement CSP process expressions via Java. Therefore JCircus clearly addresses the same executability objective that EB^3PAI aspires to. However, JCircus has certain limitations which prevent us from using a similar approach in our IS synthesis context. In JCircus, a quantified (replicated) interleave over a large set is not optimized. JCircus proceeds in a straightforward manner by converting the quantification into a large composition of binary interleave expressions with each process represented by a thread. For example \( \forall x \in 1..k : P(x) \) is translated to \( P(1) \mid \ldots \mid P(k) \), which is clearly inefficient when \( k \) is large (e.g. \( k = 10^9 \), which is very common for information systems). Moreover, JCircus implements CSP’s traditional semantics, which is not appropriate for us when dealing with non-determinism and internal actions.

JACK [33,34] is another library intended to implement CSP process expressions in Java. However, the same weakness can be pointed to, as a quantification is translated by threads as needed. This is unpractical when the quantification set is large as it can be in the IS domain.

In the industry, IS are typically specified using informal and semi-formal methods. In the 1980s, Structured Analysis [35] and Jackson System Development [36,37] were among the first methods proposed. Nowadays, UML [38] is more widely used [39].

The current technology for IS development essentially offers clerical support for defining abstract database models and class diagrams and translating them into concrete database schemes and class definitions. The bulk of the design, programming and testing is done manually by humans. These three activities consume up to 70% of the development effort [40]. The key to reducing development costs and increasing quality clearly resides in eliminating or mechanizing these three tasks.

This paper is organized as follows. Section 2 describes the EB^3 process algebra. Section 3 briefly explains the symbolic computation of process expressions using an operational semantics. Section 4 describes a syntactic simplification, three optimization techniques and their specification with a new operational semantics. The equivalence of this semantics with the original semantics of the EB^3 process algebra is presented in Section 5. Finally, we conclude with some remarks and future work in Section 6.

2. The EB^3 process algebra

2.1. Syntax

A process expression is defined over a set of symbols \( \Sigma \), called the action set, whose elements are denoted by \( a(t_1, \ldots, t_n) \), where \( a \) is an action label and \( t_i \) denotes a constant or a variable. Set \( \Sigma_0 \) is the set of ground actions from \( \Sigma \), i.e. those with no variable; it is called the input event set. Set \( \Sigma_1 \) denotes the set of labels of actions in \( \Sigma \). The process expressions over \( \Sigma \) are defined recursively as follows. Elements of \( \Sigma \cup \{ \lambda \} \), with \( \lambda \notin \Sigma \), represent elementary process expressions over \( \Sigma \). The symbol \( \square \), called "box", is an elementary process expression denoting successful completion. Let \( E, E_1 \), and \( E_2 \) be process expressions over \( \Sigma \), \( n \in \mathbb{N} \), \( \Delta \subseteq \Sigma_1 \) and \( \Phi \) be a formula. The expressions \( E^*, E^+, E_1 \cdot E_2, E_1 | E_2, E_1[\Delta] | E_2, E_1 \parallel E_2, E_1 || E_2 \) and \( \Phi \implies E \) are process expressions over \( \Sigma \). Operations \( \cdot, +, |, \parallel, || \) and \( \Phi \implies E \) denote the usual Kleene closure [41], positive closure, and concatenation of regular expressions. Operation \( | \) is a choice between \( E_1 \) and \( E_2 \); it is drawn from regular expressions and CSP [3]. Operation \( [\Delta] \) is the parameterized parallel composition of \( E_1 \) and \( E_2 \) with synchronization on actions whose labels belong to \( \Delta \); it is drawn from LOTOS. Intuitively, the composition \( E_1[\Delta] | E_2 \) is a process that can execute actions of either \( E_1 \) or \( E_2 \) without constraint, but actions in \( \Delta \) must be executed by both \( E_1 \) and \( E_2 \). Actions in \( \Delta \) are the interleave and parallel composition of CSP [3], respectively; they are special variants of \( | \): \( E_1 || E_2 \) is equivalent to \( E_1[\emptyset] | E_2 \) and \( E_1 \parallel E_2 \) is a synchronized composition of \( E_1 \) and \( E_2 \) on shared actions of \( E_1 \) and \( E_2 \), i.e. \( E_1 \parallel E_2 \).
the definition of the function \( \alpha \). The process expression \( \Phi \Rightarrow E \) is the guard of \( E \) by \( \Phi \): it means that \( E \) can execute an action if and only if \( \Phi \) is true. The special symbol \( \lambda \) denotes an internal action that a process may execute without requiring input from the environment. It plays a role similar to that of the empty word \( \epsilon \) in regular expressions or the unobservable action \( \tau \) in CCS and \( \uparrow \) in LOTOS. The \( \varepsilon \mathbf{b}^3 \) process algebra also allows quantification (also called indexing or replication in CSP) over operators \(|, ||[\Delta]|, |||\). For instance, the process expression \( |x \in 1..n : P(x)\) denotes \( P(1) \mid P(2) \mid \cdots \mid P(n) \). Quantifications are restricted to finite sets. Finally, a process may be declared using a name \( P \), a vector \( \overrightarrow{v} \), and a body, which is a process expression \( E \). Its syntactical form is \( P(\overrightarrow{v}) \triangleq E \).

For the sake of readability, we sometimes write instead of \( \alpha(\cdot) \). We use the following precedence of operators from highest to lowest, enclosing between \( ( \) and \( ) \) operators with the same precedence: \( +, \times \), \( |, ||[|], ||| \) as binary operators), \( (||[|], |||, || \) as quantified operators).

**Definition 2.1.** The set \( PE \) is the set of process expressions over \( \Sigma \).

Among the set of process expressions, we distinguish those which are defined by the user.

**Definition 2.2.** The set \( PE^{init} \) is called the *set of initial process expressions* and is defined as the set of process expressions over \( \Sigma \) that do not contain \( \emptyset \).

Process expressions containing \( \emptyset \) result only from executing an action.

### 2.2. Operational semantics

The \( \varepsilon \mathbf{b}^3 \) process algebra has an operational semantics in the spirit of CCS [4] defined by a set of transition rules, shown in Figs. 13 and 14 in pages 50 and 51, drawn from [2]. They define two transition relations: \( \rightarrow \) and \( \sim \). The first relation is used for atomic transitions: if \( E \sim E' \) then the process denoted by the process expression \( E \) can execute an action \( \sigma \) and become a process denoted by the process expression \( E' \). When there is no transition possible for \( E \), then \( E \) is equivalent to a deadlock. The second relation is used for trace transitions: if \( E \sim E' \) then the process denoted by \( E \) can execute the sequence of events (trace) \( s \) and become the process denoted by \( E' \). Expression \( s_1 \sim s_2 \) denotes the concatenation of sequences \( s_1 \) and \( s_2 \).

Rules are used to determine the actions that a process can execute. For example, rule FSD-1 says that a process expression \( \sigma \) can execute the action \( \sigma \) and become the process expression \( \emptyset \), which denotes successful completion. Rules FSD-7 and FSD-8 describe the semantics of the Kleene closure, FSD-3 and FSD-4 the semantics of the sequential composition of two process expressions, and, FSD-5 and FSD-6 the semantics of the choice operator. Rules FSD-14 to FSD-16 define quantified operations. \( x \in s[x := a] \) denotes the substitution of \( x \) by \( a \) in \( x \in s \). Symbol \( a \) denotes the value of \( x \) which must be chosen to execute a transition. For the sake of simplicity, we assume available a set of types and definedness conditions for operations on these types. The next section will illustrate the application of these rules. Some operators are defined from the others as they are only syntactic sugars: Fig. 14 provides the definition of positive closure \( (E^+) \), interleave \( (E_1 \parallel E_2) \), parallel composition \( (E_1 \parallel E_2) \) and quantification of interleave.

### 2.3. A small example

The following expression is a typical process expression that describes the expected behavior of bank accounts:

```plaintext
main \triangleq \emptyset \mid n \in AID : Account(n);
Account(n : AID) \triangleq
  open(n) .
  (open(n), _)
  (| \{y \in 0..maxint : (balance(trace, n) \geq y) \Rightarrow withdraw(n, y))
  | get_balance(n))
)\^*. close(n)
```

---

**Fig. 2.** Definition of the function \( \alpha \).
In this example, $n$ is the account number and $\text{AID}$ is the type of $n$ (i.e. the set of all possible values for $n$). Types used in signatures of actions and process definitions must be non-empty and finite. An action denotes a service of the IS that the user can invoke. An account $n$ must be first opened. Then, one can deposit any amount to this account without restriction. One can also withdraw an amount that is less than the balance of this account. Function $\text{balance}$ denotes an entity attribute which is defined on the system $\text{trace}$; the definition of $\text{balance}$ is omitted. The system $\text{trace}$ is the sequence of all valid events that has been executed by the system.

At any time, one can ask for this balance by submitting input $\text{get\_balance}(n)$. An input–output rule (also omitted) would define the output of this event as the value $\text{balance}(\text{trace}, n)$. Eventually, the account can be closed in order to end its activities. The wildcard symbol “_” in $\text{deposit}(n, \_)$ denotes that any value is accepted at execution time. It is an alias for a quantified choice. Hence, we have:

$$\text{deposit}(n, \_ ) = | y \in 0 .. \text{maxint} : \text{deposit}(n, y)\).$$

The wildcard symbol is not used with $\text{withdraw}(n, y)$ because we need to name the amount to use for $\text{balance}(\text{trace}, n) \geq y$.

3. Symbolic computation of transitions

This section presents the basic technique for the symbolic computation of an EB$^3$ process expression transition. The transition rules of Fig. 13 allow us to prove that a process expression $P$ can execute an action $\sigma$ and be transformed into a process expression $Q$, which we denote by the transition $P \xrightarrow{\sigma} Q$. Given a process expression $P$ and an action $\sigma$, one can compute the possible transitions and resulting process expressions using the inference rules. This involves a proof search that determines which inference rules are applicable, by matching the structure of $P$ with $E_1$ in an inference rule of the form $E_1 \xrightarrow{\sigma} E_2'$. When a match is found, the rule's premise which are themselves transitions (e.g. $E_2 \xrightarrow{\sigma} E_2'$), induce a recursive search. Ultimately, the search reaches a rule which does not have a transition in its premise (e.g. rule FSD-1). Then, the resulting process expression $Q$ is incrementally constructed over the inference rules through termination of recursive search calls. In summary, we do not generate code. EB$^3$PAI can be considered as a virtual machine and each specification becomes a high-level program. This algorithm is implemented in Java. In the CSP interpreter CIA [22], logic programing is used, which simplifies the implementation task at the expense of space and time complexity.

To illustrate the notion of a proof of a transition, consider the following example. Let $P \triangleq a \cdot (b \mid c) \cdot d$. According to the FSD rule system, the process $P$ can execute $a$, then either $b$ or $c$, then $d$. The sequence of transitions is as follows:

$$P \xrightarrow{a} \mathbb{I} \cdot (b \mid c) \cdot d \xrightarrow{b} \mathbb{I} \cdot d \xrightarrow{\lambda} \mathbb{I}$$

For illustration purposes, we will present the proof of the transition on $b$. Other proofs are omitted, as they are very similar.

| FSD-1 | $b \in \Sigma_r \cup \{\lambda\}$
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$b \xrightarrow{b}$</td>
</tr>
<tr>
<td>FSD-3</td>
<td>$\mathbb{I} \cdot (b \mid c) \cdot d \xrightarrow{b} \mathbb{I} \cdot d$</td>
</tr>
<tr>
<td>FSD-4</td>
<td>$\mathbb{I} \cdot (b \mid c) \cdot d \xrightarrow{\lambda} \mathbb{I} \cdot d$</td>
</tr>
</tbody>
</table>

To illustrate quantification, consider the following example. Let

$$P \triangleq | x \in 1..3 : Q(x)\),$$

where

$$Q(x) \triangleq a(x) \cdot b(x) \mid c(x) \cdot d(x)$$

Fig. 3 shows the labelled transition system (LTS) of $Q(x)$. It describes precisely the behavior of the interleaving operator $\mid$. Since $Q(x)$ is quantified by a choice $|$, on the interval $1..3$, six actions can be executed from $P$: $a(1)$, $c(1)$, $a(2)$, $c(2)$, $a(3)$ and $c(3)$. The first action executed determines the value of $x$. Here is a valid sequence of transitions from $P$ to $\mathbb{I}$:

$$P \xrightarrow{a(2)} a(2) \cdot b(2) \mid c(2) \cdot d(2)$$
$$\xrightarrow{b(2)} \mathbb{I} \cdot b(2) \mid c(2) \cdot d(2)$$
$$\xrightarrow{\lambda} \mathbb{I} \cdot d(2)$$

When a process expression contains an unguarded recursive call (i.e., a call which does not occur in at least one concatenation as the second operand), the proof search diverges.
4. Optimized rules

The FSD rule system is not adequate for achieving an efficient execution of an action on an EB³ process expression with the simple execution strategy of Section 3. Specifically, there are three problems that have to be tackled: first, the EB³ substitution operator lacks efficiency for execution purposes; second, the need to execute the internal action λ can induce a divergence during execution and drastically increase execution time; third, if a process expression is non-deterministic, backtracking on the FSD rule system requires a large amount of memory. This section addresses the solution of these problems by proposing a new set of rules on which EB³ pai is based. Furthermore we introduce a syntactic simplification which streamlines the execution of sequential composition. As a result, this section presents four modifications of the FSD rule system, in this order:

1. a syntactic simplification to replace ⊕ . E by E;
2. use of environments rather than the substitution operator;
3. elimination of internal transitions (λ-transitions);
4. management of non-determinism.

In Section 5, we show that the proposed set of rules is equivalent to the FSD system.

Each modification entails a new rule system. We denote the rule system obtained after making the first modification by $M^1$, the second (see Figs. 15 and 16) by $M^2$, the third (see Figs. 17 and 18) by $M^3$, and the last one, which is the final system (see Figs. 19–21) by PAI. Fig. 22 shows the relation between rules of the five systems. This allows us to conduct an incremental proof of equivalence between FSD and PAI. We denote rule x of system R by R-x.

4.1. Syntactic simplification

The first modification is a syntactic simplification. It allows users to have a shorter resulting process expression after a transition. From rules FSD-3 and FSD-4 (Fig. 13), it seems obvious that we can replace all ⊕ . E by E for any process expression E. To achieve this, it is enough to consider the resulting process expression of the transition for these two rules, and return $E_2$ if $E'_1 = ⊕$; $E'_1 , E_2$ otherwise. Thus rule FSD-3 is modified by adding the hypothesis $E'_1 \neq ⊕$ and becomes rule $M^1$-3, and rule FSD-4 is replaced by rule $M^1$-4, as follows:

\[
M^1-3 \quad E_1 \xrightarrow{\sigma} E'_1 , E'_1 \neq ⊕ \quad \text{by} \quad E_1 . E_2 \xrightarrow{\sigma_{|_u}} E'_1 . E_2 \\
M^1-4 \quad E_1 \xrightarrow{\sigma} E'_1 \quad \text{by} \quad E_1 . E_2 \xrightarrow{\sigma_{|_u}} E'_1 . E_2
\]

The rule FSD-8 is also replaced by

\[
M^1-8 \quad E \xrightarrow{\sigma} E', E' \neq ⊕ \quad \text{by} \quad E^* \xrightarrow{\sigma_{|_u}} E', E^*
\]

Finally, we also need a new rule for the Kleene closure operator, in order to handle the case where the operand of the closure operator results in ⊕ after executing an action:

\[
M^1-7' \quad E \xrightarrow{\sigma} E^*, E^* \neq ⊕ \quad \text{by} \quad E^* \xrightarrow{\sigma_{|_u}} E^*, E^*
\]

This rule is a new one. Rule $M^1$-x where x is not one of 3, 4 or 8 is the same as rule FSD-x.
4.2. Environment vs. substitution

Our second modification aims to enhance the behavior of substitution in \( E^3 \). We use environments in order to postpone the application of substitution until it is absolutely necessary (i.e., a kind of lazy evaluation). In the \( FSD \) system, substitution is immediately applied to the entire process expression, even when it is not needed, as in the example in Fig. 4.

Indeed, the proof of the first transition in the example is the following:

\[
\begin{array}{c}
\text{M}^1-14 \quad \frac{a \cdot b(0) \mid c \cdot d(0) \xrightarrow{\sigma} b(0)}{P(0) \xrightarrow{\sigma} b(0)}
\end{array}
\]

The process expression

\[
a \cdot b(0) \mid c \cdot d(0)
\]

is obtained from

\[
(a \cdot b(x) \mid c \cdot d(x))[x := 0]
\]

by applying substitution as it is defined in the \( FSD \) rules and, therefore, in the \( M^1 \) rules. Hence, substitution is applied to \( d(x) \) even if the right part of the choice will not be involved in a transition after executing \( a \). On a large process expression this could lead to a waste of time. Moreover, in large interleave quantifications, each interleave process expression differs from the others only in the substituted text, which is a waste of memory space. For example, \( ||x \in 1..10^9 : E|| \) is expanded as \( E[x := 1] || \ldots || E[x := 10^9] \), where \( E[x := a] \) is a new instantiation (i.e., a copy) of \( E \) with \( x \) replaced by \( a \) for each \( a \) in \( 1..10^9 \).

**Definition 4.1.** An environment is defined as a list \( \langle v_1 := t_1, \ldots, v_n := t_n \rangle \), where \( v_i \) is a variable and \( t_i \) is a term.

An environment \( \Gamma \) plays two roles. First, it is an \( E^3 \) process algebra operator (but not available to the user for specification construction; it is for internal use only). Hence the process expression \( \Gamma P \) denotes the application of environment \( \Gamma \) to process expression \( P \). We then say that \( \Gamma \) is a process environment. Second, it is a substitution: the expression \( u[\langle v_1 := t_1, \ldots, v_n := t_n \rangle] \) does the simultaneous substitution of \( v_1, \ldots, v_n \) by \( t_1, \ldots, t_n \) in \( u \).

**Definition 4.2.** The symbol \( \circ \) is a composition operator on environments such that \( t[\Gamma_1 \circ \Gamma_2] = (t[\Gamma_1])[\Gamma_2] \).

The inference rules of \( M^1 \) have been rewritten to use environments instead of direct substitution application. The new system is called \( M^2 \). Semantically, systems \( M^1 \) and \( M^2 \) are trace-equivalent. This is proved in [42] and analyzed in Section 5. An execution environment is added to the transition relation \( \rightarrow_{\omega^2} \) along with the action which is executed. For example \( E \xrightarrow{\sigma, \Gamma} E' \) means that the process expression \( E \) can execute the action \( \sigma \) in the environment \( \Gamma \) and become \( E' \).

Environments are introduced in trace rules. To be consistent with the \( FSD \) system, the trace transition relation is redefined to call atomic transitions with an empty environment. Environments evolve when quantified operations and process calls are used. Rules \( M^2-1 \) and \( M^2-2 \) in Fig. 16 introduce an empty environment when trying to execute a sequence of events.

Rule \( M^2-1 \) needs to be modified to apply the execution environment \( \Gamma \) as a substitution on \( \sigma \). It becomes rule \( M^2-1 \):

\[
\begin{array}{c}
\text{M}^2-1 \quad \frac{\sigma[\Gamma] = \sigma' \land \sigma'' \in \Sigma_e \cup \{\lambda\}}{\sigma[\sigma', \Gamma] \rightarrow_{\omega^2}}
\end{array}
\]

Rules \( M^2-18 \) and \( M^2-19 \) in Fig. 16 handle the environment. Rule \( M^2-19 \) discards the process environment \( \Gamma \) when the result is \( \square \). Indeed, since \( \square \) denotes termination, the process environment becomes useless and can even become a problem for other rules. Rule \( M^2-18 \) inserts a process environment \( \Gamma \) in the execution environment to compute the result, then wraps this result with \( \Gamma \).

In Lotos [6] and CSP [20], a different approach is used. For example, in Lotos if \( P[a, b, c](x) \triangleq a!x; b!x; \text{stop} \) then \( P[a, b, c](1) \) can execute \( a!1 \), with the Lotos rules [6] provided in Fig. 5. Here is a part of the proof:

\[
\begin{array}{c}
\text{LOTOS}_1 \quad \frac{a!x; b!x; \text{stop}}{a!x; b!x; \text{stop}}
\end{array}
\]

\[
\begin{array}{c}
\text{LOTOS}_3 \quad \frac{(a!x; b!x; \text{stop})[x := 1] \xrightarrow{a!1} (b!x; \text{stop})[x := 1]}{P[a, b, c](1) \xrightarrow{a!1} (b!x; \text{stop})[x := 1]}
\end{array}
\]
Each substitution can easily be seen as an environment and this execution is equivalent to the following in EB\(^3\) using \(M^2\):

\[
\begin{array}{c}
\text{M}^2-18 \\
\qquad a(x) \cdot b(x) \\
\quad \text{[(x := 1)b(x)]} \\
\end{array}
\]

\[
\begin{array}{c}
\text{M}^2-14 \\
\quad (x := 1) a(x) \cdot b(x) \quad \text{[(x := 1)b(x)]} \\
\end{array}
\]

where \(P'\) is the EB\(^3\) equivalent process expression of \(P\), i.e. \(a(x) \cdot b(x)\). While the two approaches look similar, they are actually opposite. The Lotos approach finds the formal parameter \(x\) from substitution \([x := 1]\) and replaces it in the action. In EB\(^3\), the substitution is added to the execution environment and evaluated only in the leaves of the proofs. Moreover, the action to execute is the same in each proof step, whereas in Lotos, the action to execute changes when the substitution is dealt with during the proof. For this example, the algorithmic complexity of the proof computation is the same for both approaches. But if \(P[a, b, c](x, y)\) is defined by \(alx; bly; \text{stop}\) (and, respectively, \(P'\) is defined by \(a(x) \cdot b(y)\)) the EB\(^3\) approach is more efficient. In Lotos, we have the following proof:

\[
\begin{array}{c}
\text{(1)} \quad alx; bly; \text{stop} \\
\quad \text{[(x := 1)b(y)]} \quad \text{[(x := 1)b(y)]} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Lotos}_1 \\
\text{Lotos}_3 \\
\end{array}
\]

There are two ways to instantiate rule \(\text{Lotos}_3\); the first is with the pair \((g, g') = (x, 1)\) and the second, with the pair \((g, g') = (y, 1)\). In EB\(^3\), there is only one way to apply rule \(M^2-18\), which handles substitution, thanks to the use of execution environments. Hence, the EB\(^3\) proof search space is smaller. Here is the corresponding proof in EB\(^3\):

\[
\begin{array}{c}
\text{M}^2-18 \\
\qquad a(x) \cdot b(y) \\
\quad \text{[(x := 1)b(y)]} \\
\end{array}
\]

\[
\begin{array}{c}
\text{M}^2-14 \\
\quad (x := 1) a(x) \cdot b(y) \\
\end{array}
\]

\[
\begin{array}{c}
\text{[(x := 1)b(y)]} \\
\end{array}
\]

The Lotos approach is more elegant since the substitution operator is “removed” by application of the rule (i.e. it does not occur in the process expression of the premise of rule \(\text{Lotos}_3\)), which is consistent with the style used for the other operators in the process algebra. The EB\(^3\) approach is more efficient because it does not need to cover the entire process expression syntax tree to find a correct instantiation of rule \(M^2-18\). This is why we chose this approach.
Fig. 6. Rules for λ-modulo reduction.

4.3. Execution of λ-action

The eb² specification language uses an internal action $\lambda$ which can play a similar role to that of $\epsilon$ in regular expressions. In order to reduce the number of rules in the system, some process expressions need to execute a $\lambda$ to become $m^2 - 10$ and $m^2 - 7$). A user never asks to execute $\lambda$ and has no control on its execution. It must be automatically executed by the interpreter. For example, in the transitions $a^* \cdot b \xrightarrow{\lambda} m^2 \triangleright \lambda$ $b \xrightarrow{m^2} \triangleright$, in the system $m^2$, the interpreter must automatically execute the $\lambda$-transition in order to execute $b$ on $a^* \cdot b$.

Therefore, to determine whether an action $\sigma$ can be executed on a process, it is sometimes necessary to try executing an undetermined number of $\lambda$ (internal transitions) before executing $\sigma$. For example, the process $\lambda^* \cdot a$ can execute any sequence of $\lambda$ (including none) and then an $a$. Trying to execute all sequences of $\lambda$ introduces a divergence. Indeed, $\lambda^* \cdot a$ is a fixed point in the system $m^2$ (in the system $fsd$ the corresponding fixed point is $\triangleright \lambda^* \cdot a$):

$$\lambda^* \cdot a \xrightarrow{\lambda} m^2 \triangleright \lambda^* \cdot a.$$

To tackle this problem, we define $E_1 \geq_r E_2$, a reduction relation that determines whether a process expression $E_1$ can result in $E_2$ after executing some $\lambda$-transitions in environment $\Gamma'$ (or whether $E_1$ is $E_2$). In that case, we say that $E_1$ can be reduced to $E_2$. If $E_1 \geq_r \triangleright$, then a process expression $E_1 \cdot E_2$ can execute an action from $E_1$ or from $E_2$ in $\Gamma'$.

Our specific concern is with the possibility of reducing a process expression to $\triangleright$. We want to find an efficient way to compute such a reduction, without executing $\lambda$. To achieve this, we define a new relation $E \geq_r \triangleright$ which intuitively means that process expression $E$ can be replaced by $\triangleright$ in environment $\Gamma'$ for transition calculation purposes. Relation $\geq_r \triangleright$ is recursively defined on the structure of a process expression, whereas $\geq$ is defined using transitions and the rule system. We proved in [42] that there is an equivalence between the relations $\geq \triangleright$ and $\geq_r \triangleright$: we have $E \geq_r \triangleright$ if and only if there exists a sequence of $\lambda$-transitions from $E$ that results in $\triangleright$. Hence, the interpreter does not need to compute $\lambda$-transitions, which eliminates $\lambda$-induced divergence problems and the computation of long chains of $\lambda$-transitions.

Fig. 6 shows the rules defining the computation of $\geq_r \triangleright$ for all operators. The positive closure and $n$-iteration operators can be derived from the sequence and the Kleene closure operators and the corresponding rules are, therefore, omitted. The parallel composition and interleave rules are also omitted, for the same reason. In this definition, $\Lambda_p(\Gamma')$ is the necessary and sufficient condition (formula) such that $\Gamma' E$ can be reduced to $\triangleright$. It is called the dependency formula. Note that it is safe to reduce $E^*$ to $\triangleright$ since we are using a trace semantics (and not a more discriminant semantics like bisimulation). The computation of the formula $\Lambda_p$ is given in Section 4.3.1.

The computation of $\geq_r \triangleright$ for process calls is carried out in two steps: prior to execution, a preruntime analysis is performed by iteration over process definitions (in order to deal with recursive processes); during execution of a transition, a formula must be evaluated. Since the preruntime analysis is conducted before execution, it does not influence execution time. This part yields three possible results:

- the process call can always be reduced to $\triangleright$;
- the process call can never be reduced to $\triangleright$;
- the process call can be reduced to $\triangleright$ iff $\Lambda_p(\Gamma')$, where $\Gamma'$ represents the environment of $P$. 
The last possibility is only due to the presence of a guard in the process definition. In this case, the condition must be tested for, during execution to determine whether the process call can be reduced to \( \Box \). In theory, the runtime part could be expensive to evaluate, since it may require verification of some existential quantifier on a large set. However, this is not the case in practice. A discussion is provided at the end of this section (Section 4.3.2).

4.3.1. Preruntime computation of dependency formula

We now show how to compute \( \Lambda_p(I) \), the dependency formula, which determines if a process call can be reduced to \( \Box \). This computation is done at preruntime. Intuitively, a process call can be reduced to \( \Box \) if the corresponding process body can be reduced to \( \Box \). We have to deal with mutually recursive process definitions, which can induce cycles in the process calls. To detect and remove these cycles, we define a set of equations from the set of process definitions. We then rewrite this set of equations until cycles are detected and removed. For each process definition \( P(x_1, \ldots, x_n) \triangleq E \), we define the equation

\[
\Lambda_p(I) = E^r
\]

where \( E^r \) is recursively defined on the structure of \( E \) by:

- \( \pi^r = \text{false} \) for every \( \sigma \in \Sigma_e \);
- \( \chi^r = \text{true} \);
- \( E_0^r = \text{true} \);
- \( E_1 \lor E_2 = E_1^r \lor E_2^r \);
- \( E_1 \land E_2 = E_1^r \land E_2^r \);
- \( \Phi \rightarrow E = \Phi[I'] \land E^r \);
- \( \exists x \in s : E^r = \exists x \in E^r \);
- \( [\Delta] x \in s : E^r = s \neq \emptyset \land \forall x \in E^r \);
- \( Q(a_1, \ldots, a_m)^r = \Lambda_Q((y_1, \ldots, y_m := a_1, \ldots, a_m) \circ I) \), given the process definition \( Q(y_1, \ldots, y_m) \triangleq E' \).

We can solve this set of equations by rewriting the right-hand side (RHS) of equations using simple predicate calculus laws and the equation themselves when a dependency formula does not refer to another formula.

(1) repeat

(a) rewrite a dependency formula with one of the applicable rewriting rules of Fig. 7.
(b) if there are two equations

(i) \( \Lambda_p(I) = \psi \), where \( \psi \) contains no reference to a dependency formula
(ii) \( \Lambda_p(I) = \ldots \Lambda_p(I') \ldots \)

then rewrite \( \Lambda_Q(I) \) as \( \Lambda_Q(I') = \ldots \psi[I' := I'] \ldots \)

until no more rewriting occurs

Let \( \#(\Lambda_p(I)) \) denote the number of references to a dependency formula in the RHS of equation \( \Lambda_p(I) \). Let \( s(\Lambda_p(I)) \) be the size of the RHS of equation \( \Lambda_p(I) \). Finally let \( M(\Lambda_p(I)) \) defined by \( \#(\Lambda_p(I)), s(\Lambda_p(I)) \). It is now easy to define a well-founded ordering on \( M \): let \( M_D = M(\Lambda_p(I)) \) and \( M_Q = M(\Lambda_Q(I)) \) then \( M_D <_M M_Q \) iff

\[
\#(\Lambda_p(I)) < \#(\Lambda_Q(I)) \lor (\#(\Lambda_p(I)) = \#(\Lambda_Q(I)) \land s(\Lambda_p(I)) < s(\Lambda_Q(I)))
\]

This algorithm terminates because: (i) the rewriting rules of Fig. 7 reduce the size of the formula through step (a); (ii) when an equation is used as a rewriting rule in step (b), it reduces the number of references to dependency formulas. Therefore, the measure \( M \) decreases at each iteration of step (1).

Step (a) eliminates process call cycles whenever possible. At the end of this algorithm, we obtain a set of equations where there may still be references between dependency formulas. These denote process call cycles which could not be eliminated; in that case, such process calls cannot be reduced to \( \Box \). Hence, each remaining reference to a dependency formula is replaced by \( \text{false} \), which takes into account divergent recursive calls. We can further simplify the dependency formulas using the predicate calculus rules. Finally, the resulting dependency formulas are either \( \text{true} \) (i.e. a process call reducible to \( \Box \)), \( \text{false} \) (i.e. a process call not reducible to \( \Box \)), or a non-trivial formula, induced by a guard operator. Non-trivial formulas are evaluated at runtime, which will provide the value of \( I' \).
For example, let \( P() \), \( Q() \), \( R() \) and \( S() \) be the following process definitions:

\[
P() \triangleq \text{init} \cdot Q(0) \mid \lambda \\
Q(x) \triangleq \text{add}^* \cdot R(x) \\
R(x) \triangleq Q(x + 1) \mid \text{remove}(x) \\
S() \triangleq P() \parallel \parallel P()
\]

The corresponding dependency formulas are the following:

\[
\Lambda_P(\Gamma) = (\text{false} \land \Lambda_Q([x := 0] \prec \Gamma)) \lor \text{true} \\
\Lambda_Q(\Gamma) = \text{true} \land \Lambda_R([x := x] \prec \Gamma) \\
\Lambda_R(\Gamma) = \Lambda_Q([x := x + 1] \prec \Gamma) \lor \text{false} \\
\Lambda_S(\Gamma) = (\Lambda_P([\parallel] \prec \Gamma) \land \Lambda_P([\parallel] \prec \Gamma))
\]

After the rewriting of step (a), the dependency formulas become

\[
\Lambda_P(\Gamma) = \text{true} \\
\Lambda_Q(\Gamma) = \text{true} \land \Lambda_R([x := x] \prec \Gamma) \\
\Lambda_R(\Gamma) = \Lambda_Q([x := x + 1] \prec \Gamma) \lor \text{false} \\
\Lambda_S(\Gamma) = (\Lambda_P([\parallel] \prec \Gamma) \land \Lambda_P([\parallel] \prec \Gamma))
\]

Since the formula \( \Lambda_P(\Gamma) \) does not refer anymore to another dependency formula, we can use it to rewrite \( \Lambda_S(\Gamma) \) to \( \text{true} \land \text{true} \) by step (b). Iterating on step (a), we obtain the following dependency formulas.

\[
\Lambda_P(\Gamma) = \text{true} \\
\Lambda_Q(\Gamma) = \Lambda_R([x := x] \prec \Gamma) \\
\Lambda_R(\Gamma) = \Lambda_Q([x := x + 1] \prec \Gamma) \\
\Lambda_S(\Gamma) = \text{true}
\]

No more rewriting can occur. Each process call of \( P \) and \( S \) can be considered reducible to \( \parallel \). Since both \( \Lambda_Q(\Gamma) \) and \( \Lambda_R(\Gamma) \) contain a reference to a dependency formula, process calls to \( Q \) and \( R \) are never reducible to \( \parallel \). Hence, their dependency formula is rewritten to \( \text{false} \).

Since none of the process definitions \( P, Q, R \) and \( S \) have guards (\( \implies \)), their reducibility to \( \parallel \) is determined at preruntime. However, the existence of a guard operator in a specification may produce a dependency formula that cannot be evaluated to \( \text{false} \) or \( \text{true} \) at preruntime; it may depend on the guard predicate \( \Phi \) and hence can only be evaluated at runtime.

### 4.3.2 Runtime evaluation of dependency formulas

During execution, to check whether a process call \( P(a_1, \ldots, a_n) \) can be reduced to \( \parallel \) in the environment \( \Gamma \), the interpreter has to verify whether the formula

\[
\Lambda_P([x_1, \ldots, x_n := a_1, \ldots, a_n] \prec \Gamma)
\]

is true. If the formula is either \( \text{true} \) or \( \text{false} \), there are no more steps. But if it is a non-trivial formula (neither \( \text{true} \) nor \( \text{false} \)), an evaluation of this formula must be done to know whether the reduction is possible.
The expressions which are potentially computationally expensive to evaluate at runtime are quantifications over an expression containing a guard. Indeed, the process definition:

\[ P(x) \triangleq \Phi \implies (a(x) \mid \lambda) \]

has the following dependency formula:

\[ \Lambda_P(\Gamma) = \Phi[\Gamma] \]

Consequently, if we have to check whether the process expression

\[ |x \in 1..10^9 : P(x) \]

can be reduced to \( \Box \), this implies verifying whether

\[ \exists_{x \in 1..10^9} \Phi \]

An evaluation of this formula can be quite expensive depending on \( \Phi \). In the general case, it requires an iteration over the values of the interval \( 1..10^9 \) if \( \Phi \) depends on \( x \), which is too expensive to compute. In any case, the potentially expensive expressions can be identified at the preruntime analysis stage and reported to the user to warn him. Practically speaking, however, this rarely happens in IS specifications. In \[2\], Frappier and St.-Denis have defined patterns to specify IS. None of these patterns leads to such a problem. In practice, quantified expressions containing guards are usually enclosed in a Kleene closure or the guarded expression is itself enclosed in a Kleene closure. Hence the guard does not have to be evaluated for all values of \( x \). So the evaluation of \( \exists \) during transition computation is unusual and can hopefully be considered exceptional for IS. The same argument is also applicable to interleave quantification.

When formula evaluation occurs, the interpreter may not necessarily be worse than a programmer derived implementation of the specification. Indeed, one has to iterate over the quantification set to find the value that sets the guard to true. However, the programmer may take advantage of some constraints. For instance, consider the following specification where status is some attribute of entity \( x \):

\[ |x \in 1..10^9 : \text{status}(x) = \text{borrowed} \implies (a(x) \mid \lambda) \]

A programmer would implement this by using a SELECT statement that uses an index on attribute status in the database; this is indeed much faster than \( \text{EB}^3\text{PAI} \) which has to iterate on all values in \( 1..10^9 \). However, this specification could also be written as follows, where borrowed would be an externally defined function that returns the set \( \{x \mid x \in [1..10^9] \land \text{status}(x) = \text{borrowed}\} \):

\[ |x \in \text{borrowed} : (a(x) \mid \lambda) \]

In that case, \( \text{EB}^3\text{PAI} \) would be as efficient as the programmer derived implementation. This transformation is comparable to the predicate subtyping technique in HOL \[43\].

### 4.3.3. Modification in the rule system

In order to allow the execution of \( E_2 \) in \( E_1 \cdot E_2 \) when \( E_1 \) can be reduced to \( \Box \), we add rule \( \text{M}^3\cdot5 \):

\[
\text{M}^3\cdot5 \quad \frac{E_2 \xrightarrow{E_1 \cdot E_2 \xrightarrow{(\sigma, \Gamma) \rightarrow E_2'} \E_1 \性 \Box} \Gamma}{E_1 \cdot E_2 \xrightarrow{(\sigma, \Gamma) \rightarrow E_2'}}
\]

We also need to remove rules that introduce a \( \lambda \)-transition (e.g. \( \text{M}^2\cdot7 \) and \( \text{M}^2\cdot10 \)), and to restrict rule \( \text{M}^2\cdot1 \) to disallow \( \lambda \) execution since there is no longer any \( \lambda \)-transition. Rule \( \text{M}^2\cdot1 \) therefore becomes \( \text{M}^3\cdot1 \):

\[
\text{M}^3\cdot1 \quad \frac{\sigma = \sigma' \xrightarrow{\Gamma} \sigma' \in \Sigma_e}{\sigma \xrightarrow{(\sigma, \Gamma) \rightarrow \Box}}
\]

Thus, with the new set of rules, \( \lambda^* \cdot a \) can execute \( a \) since \( \lambda^* \xrightarrow{\Box} \Box \) and process expression \( a \) can execute action \( a \). Here is the complete transition proof:

\[
\text{M}^3\cdot4 \quad \frac{a = a \xrightarrow{\Box} \lambda^* \xrightarrow{\Box}}{\lambda^* \cdot a \xrightarrow{(a, \Box) \rightarrow \Box}}
\]
4.4. Handling non-determinism

A trace semantics is used in EB³ (cf. Section 2.2), which means that the process definition

\[ E \triangleq (\sigma \cdot a) \mid (\sigma \cdot b) \]

must be able to execute action \( \sigma \) followed by action b. This is due to the fact that a process expression defines the set of valid input traces of an IS in EB³. In other words, the process expression defines the language of valid executions of the IS. Hence, EB³PAI must accept, action by action, any trace which can be accepted by the process expression. This is not as straightforward to achieve as it looks, since the inference rules of Figs. 17 and 18 allow non-determinism; i.e. there may be several possible transitions for a given action.

For example, in our case, we have \( E \rightarrow^* a \) (using rule \( M^3-6 \)) and \( E \rightarrow^* b \) (using rule \( M^3-7 \)). If rule \( M^3-6 \) is selected, then the resulting process expression a cannot execute b. However, the trace semantics imposes the obligation for the interpreter to accept b at this point. Backtracking to select rule \( M^3-7 \) instead of rule \( M^3-6 \) for the execution of a is not practical in general, because it can generate an unbounded stack of execution choices. For instance, consider the following example:

\[ P(\cdot) \triangleq (a \cdot Q(\cdot) \mid a^* \cdot b) \quad Q(\cdot) \triangleq (P(\cdot) \mid c). \]

One can see that the execution of the action a on \( P(\cdot) \) can return two different processes: \( P(\cdot) \rightarrow (a, Q(\cdot) \mid a^* \cdot b) \) or \( P(\cdot) \rightarrow (a, Q(\cdot) \mid a^* \cdot b) \). Both \( Q(\cdot) \mid c \) and \( a^* \cdot b \) can execute an unbounded sequence of a. However, only the first can execute \( c \) and the second b. If an interpreter maintains a stack of execution choices in order to backtrack to execute b or c, a memory overflow can occur, because the number of choices is unbounded due to the unbounded number of a executions. Indeed, the stack can never be emptied until b or c is reached, since there are ambiguities due to non-determinism.

Hence we need a technique that not only keeps the execution options in memory, but also minimizes the state space by merging equivalent states. So we modify the set of transition rules such that the execution relation becomes deterministic, while preserving trace-equivalence. Intuitively, if there exist two transitions for \( \sigma \) from \( E \), i.e.

\[ E \xrightarrow{\sigma \cdot \Gamma} E_1 \quad \text{and} \quad E \xrightarrow{\sigma \cdot \Gamma} E_2, \]

then we represent them as a single transition with a choice between the two processes, i.e.

\[ E \xrightarrow{\sigma \cdot \Gamma} E_1 | E_2. \]

A similar approach is taken in [44]. When \( E_1 \) is trace-equivalent to \( E_2 \) for the system PAI, we simply return either \( E_1 \) or \( E_2 \), since the traces of both processes are the same. Therefore, in our previous example, we want to obtain this transition:

\[ P(\cdot) \rightarrow (a, Q(\cdot) \mid a^* \cdot b). \]

In practice, it is difficult to implement an efficient trace-equivalence relation. So the relation currently used in EB³PAI is syntactic equality, at the expense of slower execution in some unusual cases. This reduction (i.e. \( E \rightarrow E' \)) is needed because there are some process expressions that could grow indefinitely otherwise. For example, consider the process expression \( a^* \cdot a^* \). The result of executing a on this process expression is \( a^* \cdot a^* \mid a^* \). The left part of the choice operator results from the execution on the first \( a^* \) of the initial process expression; the right part is the result of an a-transition on the second \( a^* \) of the initial process expression, since the first part of the sequence in this process expression \( (a^*) \) can be reduced to \( \emptyset \). Therefore the sequence can execute transition on its second part. Here is part of the transition computation:

\[
\begin{array}{c}
\text{PAI-5} \\
\text{a^*} \rightarrow (a, Q(\cdot)) \quad a^* \rightarrow (a, Q(\cdot)) \\
\text{a^*} \rightarrow (a, Q(\cdot)) \quad a^* \rightarrow (a, Q(\cdot)) \\
\text{a^*} \rightarrow (a, Q(\cdot)) \quad a^* \rightarrow (a, Q(\cdot)) \\
\end{array}
\]

Rule PAI-5 is provided in Fig. 19. Clearly, every execution of a will create a new \( a^* \) in the resulting process expression.

The system \( M^3 \) is modified to take non-determinism into account. The resulting system is called PAI and is presented in Figs. 19-21.

For a choice expression \( E_1 \mid E_2 \), we need one more rule (PAI-12) to handle the case where both \( E_1 \) and \( E_2 \) can execute \( \sigma \). We must also modify the premise of rules \( M^3-7 \) and \( M^3-6 \) to ensure that only one of the two operands can execute \( \sigma \). The expression \( E \vdash (\sigma \cdot \Gamma) \) denotes that \( E \) cannot execute \( \sigma \). It is computed as true when no inference rule applies to \( E \); false otherwise.

For a sequence expression \( E_1 \cdot E_2 \), there are several cases to consider, depending on the following aspects: the result obtained after executing \( E_1 \) and the ability of \( E_2 \) to execute \( \sigma \) when \( E_1 \) is reducible to \( \emptyset \). Given these two aspects, we obtain five rules (PAI-3 to PAI-7), because there are five possible outcomes after executing \( \sigma: E_1 \cdot E_2, E_1, E_2, E_1 \mid E_2, E_2 \mid E_1, E_2, E_2 \).

For a parameterized parallel composition \( E_1[\Delta] E_2 \), there are four possible outcomes, depending on the ability of \( E_1 \) and \( E_2 \) to execute \( \sigma \) and the need to synchronize on \( \sigma \). Rules PAI-15 to PAI-17 deal with these cases.
The rules do not handle the reduction of choice expressions using trace-equivalence. $E^3\text{PAI}$ reduces choice expressions simply by using associativity, commutativity and idempotence of choice ($E \ | \ E = E$).

$E^3\text{PAI}$ can execute both deterministic and non-deterministic specifications; however, it can execute deterministic specifications more efficiently, since there is no need to track and compare several possible results. If the performance is too much affected by non-determinism, it can usually be manually removed by rewriting the specification into an equivalent deterministic one. For example, $a \cdot b \mid a \cdot c$ can be rewritten as $a \cdot (b \mid c)$, which is deterministic.

5. A proof of trace-equivalence between FSD and PAI

The last goal of this work is to prove that the rule system FSD and the rule system PAI are trace-equivalent. We give an outline of this proof below. The proof is decomposed into four parts, since there are four main modifications from the previous sections (Sections 4.1–4.4). For each modification, a new set of rules has been defined. We prove that each system is trace-equivalent to its predecessor. All these proofs follow the same pattern. In the first subsection, we present this pattern, which consists of a main theorem and a lemma. Then we provide a brief analysis of each proof step. A complete demonstration is available in [42].

5.1. Definitions

For each rule system, a set of valid process expressions can be defined. It is the set of all process expressions that can be reached from a process expression in $PE^{init}$ with the rule system.

**Definition 5.1.** Let $R$ be an $E^3$ rule system. $PE_R$ is called the set of valid process expressions for $R$ and is the smallest set that satisfies

1. $PE^{init} \subseteq PE_R$;
2. $(\beta_1 \in PE_R \land \exists \gamma \in \Sigma \cup \{\lambda\} (\beta_1 \xrightarrow{\gamma} \beta_2)) \Rightarrow \beta_2 \in PE_R$.

Roughly speaking, we can say that $PE_R$ is the closure of $PE^{init}$ with respect to relation $\rightarrow_R$. One can remark that the set obtained from the closure of $PE^{init}$ by the relation $\sim_R$ is also $PE_R$.

Now, we need to formally define the semantics of a system: the semantics of a system is the set of all sequences of actions that can be executed from the set of valid process expressions of the system.

**Definition 5.2.** Let $R$ be an $E^3$ rule system, and let $E$ be a process expression of $PE_R$. Then $E^R$ denotes the $R$-semantics of $E$, which is defined by

$$E^R = \{ \sigma : \sigma \in \Sigma_+ \land \exists E \in PE_R (E \xrightarrow{\sigma} E') \}.$$  

Finally, we can define the equivalence relation that will be considered below the trace-equivalence.

**Definition 5.3.** Two $E^3$ rule systems $R_1$ and $R_2$ are trace-equivalent, which is written as $R_1 \simeq_R R_2$,

if and only if, for all process expressions $E$ of $PE^{init}$,

$$E^{R_1} = E^{R_2}.$$  

5.2. Proof pattern

To prove that system $R_1$ is trace-equivalent to system $R_2$, we prove that, for each process expression $E$ in $PE^{init}$, $E^{R_1} \subseteq E^{R_2}$ and $E^{R_2} \subseteq E^{R_1}$. We consider only initial process expressions ($PE^{init}$), since only these process expressions are used to initiate a transition. Obviously, $PE_{R_1} \cap PE_{R_2} \subseteq PE^{init}$ by Definition 5.1.

The proof of trace-equivalence is decomposed into two theorems of the following form.

**Corollary 1.** For each process expression $E$ of $PE^{init}$, and for each event sequence $\sigma$, if there is a process expression $E'$ such that $E \xrightarrow{\sigma} E'$, then there exists $E''$ such that $E \xrightarrow{\sigma} E''$.

Obviously, from Definitions 5.2 and 5.3, Corollary 1 implies

$$E^{R_1} \subseteq E^{R_2}.$$  

This corollary is immediately deduced from the theorem pattern 2. In this theorem we want to prove that for all $E_1$ of $PE_{R_1}$ and $E_2$ of $PE_{R_2}$, if $E_1$ and $E_2$ have the same trace then an execution of one or several $\sigma$ will transform these process expressions into two trace-equivalent process expressions $E'_1$ and $E'_2$. However, we cannot directly use the predicate of trace-equivalence since it is the aim of our theorem to prove that the same process expression still has the same traces in $R_1$ and $R_2$. Therefore, we will use a predicate $\chi$ that links $E_1$ and $E_2$ with their structures. This predicate will depend on the rule systems we need to demonstrate their trace-equivalence. The predicate $\chi$ looks weaker than trace-equivalence initially. Nevertheless, the demonstration of theorem pattern 2 will prove that $\chi(E_1, E_2)$ implies that the trace of $E_1$ in $R_1$ is included in the trace of $E_2$ in $R_2$. 

The proof of trace-equivalence is decomposed into two theorems of the following form.
Theorem 2. For each process expression $E_1$ in $\text{PE}_{\text{R}_1}$, for each process expression $E_2$ in $\text{PE}_{\text{R}_2}$ and for each event sequence $s$, if $\chi(E_1, E_2)$ and if there is a process expression $E'_1$ such that $E_1 \xrightarrow{\sigma} E'_1$ then there exists $E'_2$ such that $E_2 \xrightarrow{\sigma} E'_2$ and $\chi(E'_1, E'_2)$.

This theorem is proved by induction on the length of $s$. The first two modifications ($M^1$ and $M^2$) are proved using an extended sequence which may contain some $\lambda$. Indeed, during a sequence execution, $\lambda$-transitions can be inserted before or after a non-$\lambda$-transition. Therefore, we do not consider the sequence $s$, which does not contain any $\lambda$, but the sequence needed to provide all transitions, including all the $\lambda$ needed. Since this is an artefact to enable induction on the length of the sequence $s$, we do not explicitly define the extended sequence. However, all details are presented in [42]. To demonstrate theorem pattern 2, we first need to establish a lemma following the pattern of 3.

Lemma 3. For each process expression $E_1$ in $\text{PE}_{\text{R}_1}$, for each process expression $E_2$ in $\text{PE}_{\text{R}_2}$ and for each $\sigma \in \Sigma_e \cup \{\lambda\}$, if $\chi(E_1, E_2)$ and if there exists a process expression $E'_1$ such that $E_1 \xrightarrow{\sigma} E'_1$ then there exists $E'_2$ such that $E_2 \xrightarrow{\sigma} E'_2$ and $\chi(E'_1, E'_2)$.

Given that $\chi$ is a structure correlation between two process expression structures, the lemma is proved recursively on the structures of $E_1$ and $E_2$. The demonstration of theorem pattern 2 with lemma pattern 3 is schematized in Fig. 8.

Theorem pattern 2 and the lemma pattern 3 can be slightly different from one proof to the other, but the scheme is still recognizable. For example, after environments are introduced, we need to take into account the current environment in $\chi$.

In the next section, we present a more precise description of the main step of each demonstration. We analyze corresponding lemmas, with the description of $\chi$ when necessary. Since the theorem and the lemma corresponding to patterns 2 and 3 can be slightly different from one proof to the other, we give them explicitly. On the other hand, the corollary corresponding to pattern 1 is always the same. Therefore, we do not mention it below and we stop the demonstration at the aforementioned theorem.

5.3. The proofs

5.3.1. Trace-equivalence between $\text{FSD}$ and $M^1$

5.3.1.1. First part of the equivalence. This part is straightforward to demonstrate. We just have to define $\chi(E_1, E_2)$ as $E_2 = \zeta(E_1)$ where $\zeta$ is, roughly speaking, a transformation of $\text{PE}_{\text{FSD}}$ that recursively converts every $\square, E$ to $\zeta(E)$. So $\zeta(E)$ for some $E$ in $\text{FSD}$ is the process expression $E$ minus all the “$\square$” occurrences. One can note that $\text{PE}_{M^1} = \text{PE}_{\text{init}}$.

Definition 5.4. Function $\zeta$ is defined from $\text{PE}_{\text{FSD}}$ to $\text{PE}_{M^1}$, for every $E_1$ and $E_2$ in $\text{PE}_{\text{FSD}}$, as follows:

- $\zeta(\square) = \square$;
- $\zeta(\sigma) = \sigma$ for all $\sigma \in \Sigma_e \cup \{\lambda\}$;
- $\zeta(E_1 \cdot E_2) = \zeta(E_1) \cdot \zeta(E_2)$ for all binary operators $\cdot$ in [[. | |.]];
- $\zeta(E_1 \cdot E_2) = \zeta(E_1) \cdot \zeta(E_2)$, if $\chi(E_1) \neq \square$;
- $\zeta(E_1 \cdot E_2) = \zeta(E_1) = \square$ if $\chi(E_1) = \square$;
- $\zeta(\square(E_1)) = \zeta(\square(E_1))$ in which $\square$ is either $\sigma$ or a quantified operator in [[. | |.]];
- $\zeta(\Phi) \rightarrow \zeta(E_1) = \Phi \rightarrow \zeta(E_1)$ for all well-formed formulas $\Phi$;
- $\zeta(P(\overrightarrow{A})) = P'(\overrightarrow{\zeta(A)})$, for all processes $P(\overrightarrow{X}) \triangleq E$ with $P'(\overrightarrow{A}) = \zeta(E)[\overrightarrow{X} = \overrightarrow{A}]$.

If we look at the differences between $\text{FSD}$ and $M^1$, we find that if a process expression $E$ can execute some $\sigma$ and become $E'$ in system $\text{FSD}$, then the result will be $\zeta(E')$ in system $M^1$. For example, if $E = (a \cdot a \cdot b)[[a]] (a \cdot a \cdot c)$ then, on the one hand,

$E \xrightarrow{\sigma_{\text{FSD}}} (a \cdot a \cdot b)[[a]][[a]] (a \cdot a \cdot c)$

and on the other hand

$E \xrightarrow{\sigma_{M^1}} (a \cdot a \cdot b)[[a]] (a \cdot c)$.

This leads to the following lemma.
Lemma 4. For all $E \in PE^{\text{init}}$, $E' \in PE_{\text{fsd}}$ and $\sigma \in \Sigma_e \cup \{\lambda\}$,

$$E \xrightarrow{\sigma_{\text{fsd}}} E' \Rightarrow E \xrightarrow{\sigma_{\lambda_1}} \zeta(E').$$

However, this lemma is not sufficient by itself to demonstrate Theorem 7, the instance of 2 for this part. Indeed, since the initial process expression is the same and the final process expressions are not, the induction step cannot be proved. So we need the following lemma.

Lemma 5. For all $E \in PE_{\text{fsd}}$, $E' \in PE_{\text{fsd}}$ and $\sigma \in \Sigma_e \cup \{\lambda\}$ such that

$$E \xrightarrow{\sigma_{\text{fsd}}} E' \Rightarrow \zeta(E) \xrightarrow{\sigma_{\text{fsd}}} E'' \wedge \zeta(E') = \zeta(E'').$$

For example, since

$$(\text{ } . a . b) || (\text{ } . a . c) \xrightarrow{a_{\text{fsd}}} (\text{ } . b) || (\text{ } . a . c),$$

then we have some $E''$ such that

$$(a . b) || (a . c) \xrightarrow{a_{\text{fsd}}} E''.$$ 

This is clearly true with $E'' = (\text{ } . b) || (a . c)$.

These two lemmas are proved by induction on the structure of $E$.

So, as a consequence of the two previous lemmas, we can derive the following lemma (the instance of 3 for this part).

Lemma 6. For all $E_1 \in PE_{\text{fsd}}$, $E_2 \in PE_{\lambda_1}$ and $\sigma \in \Sigma_e \cup \{\lambda\}$, if there exists $E'_1 \in PE_{\text{fsd}}$ such that

$$\chi(E_1, E_2) \wedge E_1 \xrightarrow{\sigma_{\text{fsd}}} E'_1,$$

then there exists $E'_2 \in PE_{\lambda_1}$ such that

$$\chi(E'_1, E'_2) \wedge E_2 \xrightarrow{\sigma_{\lambda_1}} E'_2.$$ 

Relation $\chi(E_1, E_2)$ is defined by $\zeta(E_1) = E_2$.

Proof of Lemma 6. Let $E_1$ be a process expression of $PE_{\text{fsd}}$ such that $E_1 \xrightarrow{\sigma_{\text{fsd}}} E'_1$. By Lemma 5, there exists $E''_1$ such that $\zeta(E'_1) = \zeta(E'')$ and $\zeta(E_1) \xrightarrow{\sigma_{\text{fsd}}} E''_1$. This last assertion is the premise of Lemma 4 and therefore we can assert that $\zeta(E_1) \xrightarrow{\sigma_{\lambda_1}} \zeta(E'')$. But since $\zeta(E'_1) = \zeta(E'')$ we have the conclusion of Lemma 6. Fig. 9 illustrates this demonstration.

Theorem 7. For all $E_1 \in PE_{\text{fsd}}$, $E_2 \in PE_{\lambda_1}$, $E' \in PE_{\text{fsd}}$ and $s \in \Sigma_e^+$,

$$\chi(E_1, E_2) \wedge E_1 \xrightarrow{s_{\text{fsd}}} E'_1 \Rightarrow E_2 \xrightarrow{s_{\lambda_1}} \zeta(E'_1).$$

This is the theorem sought (an instantiation of the theorem pattern 2) after. Consequently we have proved that, for all $E$ in $PE^{\text{init}}$, $E^{\text{fsd}} \subseteq E^{\lambda_1}$. 

![Fig. 9. Demonstration principle for Lemma 6.](image-url)
5.3.1.2. Second part of the equivalence. This part is quite similar to the previous part. We can easily prove the following two lemmas.

**Lemma 8.** For all $E \in \text{PE}_{m_1}$, $E' \in \text{PE}_{m_1}$ and $\sigma \in \Sigma_\epsilon \cup \{\lambda\}$, if

$$E \xrightarrow{\sigma}_{m_1} E'$$

then there is some $E'' \in \text{PE}_{\text{FSD}}$ such that

$$E \xrightarrow{\sigma}_{\text{FSD}} E'' \land \zeta(E'') = E'.$$

**Lemma 9.** For all $E \in \text{PE}_{\text{FSD}}$, $E' \in \text{PE}_{\text{FSD}}$ and $\sigma \in \Sigma_\epsilon \cup \{\lambda\}$ such that

$$\zeta(E) \xrightarrow{\sigma}_{m_1} E'$$

there is some $E'' \in \text{PE}_{\text{FSD}}$ such that

$$E \xrightarrow{\sigma}_{\text{FSD}} E'' \land \zeta(E'') = \zeta(E').$$

Clearly, these two lemmas are the duals of Lemmas 4 and 5. They lead to the following lemma, corresponding to 6 with $\chi(E_1, E_2) \Leftrightarrow \zeta(E_2) = E_1$ (an instance of lemma pattern 3).

**Lemma 10.** For all $E_1 \in \text{PE}_{m_1}$, $E_2 \in \text{PE}_{\text{FSD}}$ and $\sigma \in \Sigma_\epsilon \cup \{\lambda\}$, if there exists $E'_1 \in \text{PE}_{m_1}$ such that

$$\chi(E_1, E_2) \land E_1 \xrightarrow{\sigma}_{m_1} E'_1$$

then there exists $E'_2 \in \text{PE}_{\text{FSD}}$ such that

$$\chi(E'_1, E_2) \land E_2 \xrightarrow{\sigma}_{\text{FSD}} E'_2.$$  

Consequently we can demonstrate the main theorem of this part (an instance of theorem pattern 2).

**Theorem 11.** For all $E \in \text{PE}_{m_1}$, $E' \in \text{PE}_{m_1}$ and $s \in \Sigma_\epsilon^+$, if

$$E \xrightarrow{s}_{m_1} E'$$

then there is some $E'' \in \text{PE}_{\text{FSD}}$ such that

$$E \xrightarrow{s}_{\text{FSD}} E'' \land \zeta(E'') = E'.$$

This completes the proof that FSD is trace-equivalent to $m^1$.

**Corollary 12.** For all $E \in \text{PE}^{\text{init}}$, $E \xrightarrow{s}_{\text{init}} = E^{m_1}$.

5.3.2. Trace-equivalence between $m^1$ and $m^2$

5.3.2.1. First part of the equivalence. We define $\xi_2$, a transformation from $\text{PE}_{m_2}$ to $\text{PE}_{m_1}$, that converts a process expression $E$ with an environment $I'$ into a process expression in which $I'$ is substituted recursively on the structure of $E$. For example, if we have

$$E = (x := 1)(a(x) \cdot (y := 2)b(x, y))$$

then

$$\xi_2(E) = a(1) \cdot b(1, 2).$$

As for $\eta$ in the previous section, this transformation will be the structure correlation needed between a process expression of $m^1$ and a process expression of $m^2$. Thus, in this section, $\chi(E_1, E_2)$ is defined as $E_1 = \zeta_2(E_2)$.

**Definition 5.5.** Function $\xi_2$ is defined from $\text{PE}_{m_2}$ to $\text{PE}_{m_1}$, for all process expressions $E_1$ and $E_2$ in $\text{PE}_{m_2}$, as follows:

- $\xi_2(\emptyset) = \emptyset$;
- $\xi_2(\sigma) = \sigma$ for all $\sigma \in \Sigma_\epsilon \cup \{\lambda\}$;
- $\xi_2(E_1 \odot E_2) = \xi_2(E_1) \odot \xi_2(E_2)$ for all binary operators $\odot$ in $\{[A], \|, \|, \|\}$;
- $\xi_2(\Diamond(E_1)) = \Diamond(\xi_2(E_1))$ where $\Diamond$ is either $\ast$ or a quantified operator in $\{[A], \|, \|, \|\}$;
- $\xi_2(\Phi) \equiv E_1 \equiv \Phi \equiv \xi_2(E_1)$ for all well-formed formulas $\Phi$;
- $\xi_2(P(A)) = P'((A))$, for all $P((A)) \equiv E$, with

$$P'((A)) = \xi_2(E);$$
- $\xi_2(\Gamma E) = \xi_2(E)[\Gamma]$ for all environments $\Gamma$.  

The following two lemmas can be readily compared to the first two lemmas in the previous section. Suppose that \( P(x) = a(x) \cdot b(x) \), then we can deduce that
\[
P(1) \xrightarrow{a(1)} a(1), \ x \quad \text{and} \quad P(1) \xrightarrow{a(1)} b(1) \mid x := 1 \mid b(x).
\]
It is easy to generalize this example in the following lemma.

**Lemma 13.** For all \( E \in PE_{m^1}, E' \in PE_{m^1} \) and \( \sigma \in \Sigma_e \cup \{\lambda\} \), if
\[
E \xrightarrow{\sigma} m^1, \ E'
\]
then there exists \( E'' \in PE_{m^2} \) such that
\[
E \xrightarrow{(\sigma, \lambda)} m^2, \ E'' \wedge \zeta_2(E'') = E'.
\]

The next lemma binds each step of the induction, as described in the last section. Indeed, since \( E'' \neq E' \) a priori, Lemma 13 is not sufficient to establish the lemma corresponding to pattern 3.

**Lemma 14.** For all \( E \in PE_{m^2}, E' \in PE_{m^2} \) and \( \sigma \in \Sigma_e \cup \{\lambda\} \), if
\[
\zeta_2(E) \xrightarrow{(\sigma, \lambda)} m^2, \ E'
\]
then there exists \( E'' \in PE_{m^2} \) such that
\[
E \xrightarrow{(\sigma, \lambda)} m^2, \ E'' \wedge \zeta_2(E'') = \zeta_2(E').
\]

Now we can infer the following lemma which satisfies pattern 3.

**Lemma 15.** For all \( E_1 \in PE_{m^1}, E_2 \in PE_{m^2} \) and \( \sigma \in \Sigma_e \cup \{\lambda\} \), if there exists \( E'_1 \in PE_{m^1} \) such that
\[
\chi(E_1, E_2) \wedge E_1 \xrightarrow{\sigma} m^1, \ E'_1
\]
then there exists \( E'_2 \in PE_{m^2} \) such that
\[
\chi(E'_1, E'_2) \wedge E_2 \xrightarrow{\sigma} m^2, \ E'_2.
\]

This lemma is proved with the same kind of demonstration as for Lemma 6.

Therefore the following theorem (an instance of Theorem 2) can be proved with an induction on the length of the sequence of events \( s \) (cf. Fig. 8).

**Theorem 16.** For all \( E_1 \in PE_{m^1}, E_2 \in PE_{m^2} \) and \( s \in \Sigma_e^+ \), if there exists \( E'_1 \in PE_{m^1} \) such that
\[
E_1 = \zeta_2(E_1) \wedge E_1 \xrightarrow{\sigma} m^1, \ E'_1
\]
then there exists \( E'_2 \in PE_{m^2} \) such that
\[
E_2 \xrightarrow{\sigma} m^2, \ E'_2 \wedge E'_1 = \zeta_2(E'_2).
\]

5.2.2 Second part of the equivalence. The steps for this demonstration are quite similar to the first part of the demonstration with dual lemmas. Here is the instance of pattern 3.

**Lemma 17.** For all \( E_1 \in PE_{m^2}, E'_1 \in PE_{m^2} \) and \( \sigma \in \Sigma_e \cup \{\lambda\} \), if
\[
E_1 \xrightarrow{(\sigma, \lambda)} m^2, \ E'_1
\]
then
\[
\zeta_2(E_1) \xrightarrow{\sigma} m^1, \ \zeta_2(E'_1),\ \text{where} \ \chi(E_1, E_2) \text{is defined as} \ E_2 = \zeta_2(E_1).
\]

Finally this lemma is used to prove the next theorem (an instance of 2).

**Theorem 18.** For all \( E_1 \in PE_{m^2}, E_2 \in PE_{m^1} \) and \( s \in \Sigma_e^+ \), if there exists \( E'_1 \in PE_{m^2} \) such that
\[
E_2 = \zeta_2(E_1) \wedge E_1 \xrightarrow{\sigma} m^1, \ E'_1
\]
then there exists \( E'_2 \in PE_{m^1} \) such that
\[
E_2 \xrightarrow{\sigma} m^1, \ E'_2 \wedge E'_2 = \zeta_2(E'_1).
\]

This allows us to prove that \( m^1 \) is trace-equivalent to \( m^2 \).

**Corollary 19.** For all \( E \in PE_{m^1} \), \( \overline{m^1} = \overline{m^2} \).
5.3.3. Trace-equivalence between $m^2$ and $m^3$

5.3.3.1. First part of the equivalence. This demonstration is the most interesting one since it clearly ensures that using the $\lambda$-modulo transition system is equivalent to executing $\lambda$-transitions when needed. We will first prove in Lemma 20 that $\trianglerighteq$ and $\trianglerighteq$ are equivalent.

To achieve this, we must establish an important equivalence between the syntactic reduction $\trianglerighteq$ and the semantic reduction $\trianglerighteq$. So, first, we must formally define the $\trianglerighteq$ relation.

**Definition 5.6.** If $E$ and $E'$ are $PE_{m^2}$ process expressions, and if $\Gamma'$ is an environment, then we say that $E$ can be reduced to $E'$ in $\Gamma'$, and we write

$$E \trianglerighteq_{\Gamma'} E',$$

if and only if

- $\Gamma'[E] = \Gamma'[E']$, or
- there exists a set \{ $E_i : 0 \leq i \leq n$ \} of process expressions such that

$$E = E_0 \land E' = E_n \land \forall 1 \leq i \leq n (E_{i-1} \xrightarrow{(\lambda, \Gamma')} m^2 E_i).$$

It is important to note that $\trianglerighteq$ uses the transition relation $\rightarrow_{m^2}$ since there is no longer any $\lambda$-transition in system $m^3$.

**Lemma 20.** Let $E$ be a process expression of $PE_{m^2}$ and let $\Gamma'$ be an environment of $\Sigma_v$; then

$$E \trianglerighteq_{\Gamma'} \iff E \trianglerighteq_{\Gamma').$$

The proof is made by induction on the structure of $E$ for the left implication $E \trianglerighteq_{\Gamma'} \iff E \trianglerighteq_{\Gamma'}$, and by induction on the number of $\lambda$-transitions for the right implication $E \trianglerighteq_{\Gamma'} \Rightarrow E \trianglerighteq_{\Gamma'}$. Therefore, $E \trianglerighteq_{\Gamma}$ means that there exists a sequence of $\lambda$-transitions from $E$ to $\Gamma$ in the system $\Sigma_v$; since the system $m^3$ is trace-equivalent to the system $\Sigma_v$ (Corollary 19).

It is easy now to demonstrate the following lemma.

**Lemma 21.** For all process expressions $E$ and $E'$ of $PE_{m^2}$, for all $\sigma \in \Sigma_e$, and for all environments $\Gamma'$, if

$$E \xrightarrow{(\sigma, \Gamma')} m^2 E'$$

then

$$E \xrightarrow{(\sigma, \Gamma')} m^3 E'.$$

Indeed, since $\sigma$ is not $\lambda$, the behavior of system $m^2$ is the same as the behavior of system $m^3$ for a single transition. However, the $\sim_{m^2}$ relation can involve $\lambda$-transitions (coming from rule $m^2 \rightarrow_{\lambda} m^2$ for example). For example, the execution of the sequence $b \rightarrow d$ from the process expression $a^* \cdot b \cdot c^* \cdot d$ needs two $\lambda$-transitions in the system $m^2$:

$$\xrightarrow{(\lambda, \Gamma')} a^* \cdot b \cdot c^* \cdot d \xrightarrow{(\lambda, \Gamma')} m^2 b \cdot c^* \cdot d \xrightarrow{(\lambda, \Gamma')} m^2 c^* \cdot d \xrightarrow{(\lambda, \Gamma')} m^2 d \xrightarrow{(\lambda, \Gamma')} m^2 \Sigma_v$$

In the system $m^3$, $\lambda$-transitions no longer exist:

$$\xrightarrow{(\lambda, \Gamma')} a^* \cdot b \cdot c^* \cdot d \xrightarrow{(\lambda, \Gamma')} m^3 c^* \cdot d \xrightarrow{(\lambda, \Gamma')} m^3 \Sigma_v$$

But one can easily note that since $b$ can be obtained by a $\lambda$-transition from $a^* \cdot b$ and since $b$ can execute $b$ in system $m^2$, then $a^* \cdot b$ can execute $b$ in system $m^3$. A generalization of this remark leads to the next lemma.

**Lemma 22.** For all process expressions $E_1$, $E'_1$ and $E_2$ of $PE_{m^2}$, for all $\sigma \in \Sigma_e$, and for all environments $\Gamma'$ of $\Sigma_v$, if

$$E_2 \trianglerighteq_{\Gamma'} E_1 \land E_1 \xrightarrow{(\sigma, \Gamma')} m^2 E'_1$$

then there is a process expression $E'_2$ of $PE_{m^2}$

$$E'_2 \trianglerighteq_{\Gamma'} E'_1 \land E_2 \xrightarrow{(\sigma, \Gamma')} m^3 E'_2.$$
This lemma corresponds to pattern 3 with some minor modifications: first, \( \chi \) embeds the environment \( \Gamma' \) which is needed to compute the relation \( \succeq \) and hence \( \chi(E, E', \Gamma) \) is defined as \( E' \succeq_{\Gamma} E \); second, \( \sigma \) can no longer be \( \lambda \). In comparison with the previous lemmas, \( \chi \) does not denote a structural link. In this lemma, it denotes a semantic one, because the relation \( \succeq \) is transition based. This means that if \( E \succeq_{\Gamma} E' \) then \( E^m \succeq_\Gamma E^m \). Consequently, Theorem 23 can be proved with Lemma 22.

**Theorem 23.** For all process expressions \( E \) and \( E' \) in \( \text{PE}_{m^2} \), for all process expressions \( E_2 \) in \( \text{PE}_{m^2} \), for all sequences \( s \) of \( \Sigma_e^* \), and for all environments \( \Gamma' \) of \( \mathcal{E} n \), if

\[
E_1 \overset{\sigma}{\rightarrow}_{m^2} E_1' \wedge E_2 \succeq_\Gamma E_1
\]

then there exists a process expression \( E_2' \) of \( \text{PE}_{m^2} \) such that

\[
E_2 \succeq_{\Gamma} E_2' \wedge E_2' \succeq_\Gamma E_1'.
\]

5.3.3.2 Second part of the equivalence. The demonstration of this part is different from the others. Indeed, we cannot follow patterns 2 and 3. Actually the demonstration is quite easy.

First we prove that if \( E_1 \) can execute \( \sigma \) and becomes \( E_1' \) in environment \( \Gamma' \) in the system \( m^3 \),

\[
E_1 \overset{(\sigma, \Gamma)}{\rightarrow}_{m^3} E_1'
\]

then either

\[
E_1 \overset{(\sigma, \Gamma)}{\rightarrow}_{m^2} E_1'
\]

or there exists a sequence of \( \lambda \)-transitions from \( E_1 \) that leads to a process \( E_2 \) which can execute \( \sigma \) and becomes \( E_1' \). This can be summarized as follows:

\[
E_1 \overset{(\lambda, \Gamma)}{\rightarrow}_{m^2} \ldots \overset{(\lambda, \Gamma)}{\rightarrow}_{m^2} E_2 \overset{(\sigma, \Gamma)}{\rightarrow}_{m^2} E_1'
\]

For example, let \( E_1 \) be \( (a^* || \boxempty) \cdot b \) and let \( E_1' \) be \( \boxempty \). In system \( m^3 \), \( b \) can be immediately executed since \( a^* \succeq_{\boxempty} \boxempty \) and hence \( (a^* || \boxempty) \succeq_{\boxempty} \boxempty \). In the system \( m^2 \), however two \( \lambda \)-transitions are needed before the execution of \( b \) to transit from \( (a^* || \boxempty) \cdot b \) to \( b \).

**Lemma 24.** For all process expressions \( E_1 \) and \( E_1' \) in \( \text{PE}_{m^3} \), for all actions \( \sigma \) in \( \Sigma_e \), and for all environments \( \Gamma' \) of \( \mathcal{E} n \), if

\[
E_1 \overset{(\sigma, \Gamma)}{\rightarrow}_{m^3} E_1'
\]

then there exists \( E_2 \) a process expression of \( \text{PE}_{m^2} \) such that

\[
E_1 \succeq_\Gamma E_2 \wedge E_2 \overset{(\sigma, \Gamma)}{\rightarrow}_{m^2} E_1'.
\]

This demonstration is easily achieved with the help of Lemma 20. It is done by induction on the structure of \( E_1 \).

Since Lemma 24 is proved, the next theorem can be proved.

**Theorem 25.** For all process expressions \( E \) and \( E' \) in \( \text{PE}_{m^3} \), for all sequences \( s \) of \( \Sigma_e^* \), and for all environments \( \Gamma' \) in \( \mathcal{E} n \), if

\[
E \overset{s}{\rightarrow}_{m^3} E'
\]

then

\[
E \overset{s}{\rightarrow}_{m^2} E'.
\]

The demonstration is made by induction on the length of \( s \). It is sufficient to see that a sequence of transitions similar to (1) in system \( m^3 \) proves the transition \( E_1 \overset{s}{\rightarrow}_{m^2} E_1' \) with rules \( m^2 \cdot 1 \) and \( m^2 \cdot 3 \) (page 53).

For example, the proofs of the transition of sequence \( b \) from process expression \( E = (a^* || \boxempty) \cdot b \) in the systems \( m^2 \) and \( m^3 \) are given in Figs. 10 and 11. In Fig. 10 the expression \( E' \) denotes \( (\boxempty || \boxempty) \cdot b \).

This allows us to prove that \( m^2 \) is trace-equivalent to \( m^3 \).

**Corollary 26.** For all \( E \in \text{PE}_{m^3} \), \( E^{m^2} = \overline{E^{m^3}} \).
5.3.4. Trace-equivalence between $m^3$ and PAI

5.3.4.1. First part of the equivalence. In this part, we deal with the non-determinism of process expressions. To prove that system PAI is trace-equivalent to system $m^3$, we use the notion of part of a process expression. A part of a process $E$ is a process that has the same structure tree but only some branches of some choice operators. We first provide the definition of $\chi$ and then the definition of the notion part of.

**Definition 5.7.** Let $E'$ be a process expression in $PE_{m^3}$ and let $E''$ be a process expression in $PE_{PAI}$; then $\chi(E', E'')$ if and only if one of the following holds

- $E' = E''$;
- $E' = I'E_1, E'' = I'E_2$ and $\chi(E_1, E_2)$ for some $E_1$ in $PE_{m^3}$ and some $E_2$ in $PE_{PAI}$;
- $E' = E_1 | E_2$ or $E'' = E_2 | E_1$ and $\chi(E', E_2)$ for some $E_1$ and $E_2$ in $PE_{PAI}$;
- $E' = E_2, E_1, E'' = E_3, E_1$ and $\chi(E_2, E_3)$ for some $E_1$ and $E_2$ in $PE_{m^3}$ and some $E_3$ in $PE_{PAI}$;
- $E' = E_2[[\Delta]] E_1, E'' = E_3[[\Delta]] E_1$ and $\chi(E_2, E_3)$ for some $E_1$ and $E_2$ in $PE_{m^3}$ and some $E_3$ in $PE_{PAI}$;
- $E' = E_1[[\Delta]] E_2, E'' = E_3[[\Delta]] E_4, \chi(E_1, E_2)$ and $\chi(E_2, E_4)$ for some $E_1$ and $E_2$ in $PE_{m^3}$ and some $E_3$ and $E_4$ in $PE_{PAI}$.

**Definition 5.8.** Let $E'$ be a process expression in $PE_{PAI}$. We say that $E$ is a part of $E'$ if and only if

$$E \in PE_{m^3} \land \chi(E, E').$$

For example, if $E_1 = a \parallel b$ then $a$ and $b$ are both parts of $E_1$. If $E_2 = c || (a \parallel b)$ then $c || a$ and $c || b$ are both parts of $E_2$. If $E_3 = (a \parallel b) \parallel c$ then $a$, $a \parallel b$ and $c$ are clearly parts of $E_2$. But $a \parallel c$ is not: a part of a choice is only a part of one of its sub-process expressions or itself. It is not a choice between parts of its sub-process expression. For the concatenation, only the first process expression can be partitioned. For example, if $E_4 = (a \parallel b) \cdot (c \parallel d)$ then $a \cdot (c \parallel d)$ is a part of $E_4$ but $(a \parallel b \cdot c$ is not. As one can see from Definition 5.7, the only part of a process call $P$ is itself. We are interested only in process expressions that can appear during an execution. If $E$ is a process expression of $PE_{PAI}$ and if $E'$ is the result of the execution of some $\sigma$ from $E$ in PAI, then the result $E''$ of the execution of $\sigma$ from $E$ in $m^3$ is a part of $E'$: $\chi(E'', E')$. We need a more powerful version of this claim, and in the other direction, from $m^3$ to PAI: if $E_1$ is a part of $E_2$ and if $E_1$ can execute $\sigma$ to become $E'_2$ in $m^3$, then there exists some $E'_2$ such that $E'_2$ is the result of the execution of $\sigma$ from $E_2$ in PAI. Here is the instance of pattern 3.
Lemma 27. For all process expressions $E_1$ and $E_1'$ in $\text{PE}_{\text{pat}}$, for all process expressions $E_2$ in $\text{PE}_{\text{pat}}$, for all actions $\sigma$ in $\Sigma_\alpha$, for all environments $\Gamma$ in $\text{env}$, if

$$E_1 \xrightarrow{(\sigma, \Gamma)} E_1' \wedge E_1 \text{ is a part of } E_2$$

then there exists a process expression $E_2'$ of $\text{PE}_{\text{pat}}$ such that

$$E_2 \xrightarrow{(\sigma, \Gamma)} E_2' \wedge E_1' \text{ is a part of } E_2'$$
**Elementary operations**

\[(m^2-1): \frac{\sigma[\Gamma] = \sigma', \sigma' \in \Sigma_v \cup \{\lambda\}}{\sigma[\Gamma] \rightarrow \square}\]

\[(m^2-2): \frac{\Gamma[\Phi]}{E \stackrel{(\sigma, \Gamma)}{\rightarrow} E'}\]

\[(m^2-3): \frac{E_1 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1', E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2'}{E_1 | E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1' | E_2'}\]

\[(m^2-4): \frac{E_1 \stackrel{(\sigma, \Gamma)}{\rightarrow} \square}{E_1, E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2}\]

\[(m^2-5): \frac{E_1 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1', E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2'}{E_1 | E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1' | E_2'}\]

\[(m^2-6): \frac{E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} \square}{E_1 | E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2'}\]

\[(m^2-7): \frac{E_1 \stackrel{(l, \Gamma)}{\rightarrow} E}{E \stackrel{(\sigma, \Gamma)}{\rightarrow} E}\]

\[(m^2-8): \frac{E \stackrel{(\sigma, \Gamma)}{\rightarrow} E'}{E \stackrel{(\sigma, \Gamma)}{\rightarrow} E', E^*}\]

\[(m^2-9): \frac{E \stackrel{(\sigma, \Gamma)}{\rightarrow} \square}{E \stackrel{(\sigma, \Gamma)}{\rightarrow} E^*}\]

\[(m^2-10): \frac{E \stackrel{(\sigma, \Gamma)}{\rightarrow} \square}{E \stackrel{(\sigma, \Gamma)}{\rightarrow} E^*}\]

\[(m^2-11): \frac{E_1 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1', E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2'}{E_1[\Gamma] E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1'[\Gamma] E_2'}\]

\[(m^2-12): \frac{E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2'}{E_1[\Gamma] E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1'[\Gamma] E_2'}\]

\[(m^2-13): \frac{E_1 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1', E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_2'}{E_1[\Gamma] E_2 \stackrel{(\sigma, \Gamma)}{\rightarrow} E_1'[\Gamma] E_2'}\]

\[(m^2-14): \frac{[x_1, \ldots, x_n := a_1, \ldots, a_n] E \stackrel{(\sigma, \Gamma)}{\rightarrow} E'}{P(x_1, \ldots, x_n) \stackrel{(\sigma, \Gamma)}{\rightarrow} E'}\]

**Fig. 15.** $m^2$ rule system — 1.

This lemma is proved by induction on the size of $\chi$. The size of $\chi$ is defined as follows.

**Definition 5.9.** Let $E_1$ and $E_2$ be two process expressions such that $\chi(E_1, E_2)$. The size of $\chi(E_1, E_2)$ is equal to:

- 1 if $E_1 = E_2$;
- 1 plus the maximum of the size of all relations $\chi$ satisfied by subterms of $E_1$ and $E_2$ with respect to Definition 5.7 otherwise.

With Lemma 27 it is easy to prove the main theorem of this part (an instance of pattern 2) by induction on the length of $s$.

**Theorem 28.** For all process expressions $E_1$ and $E'$ in $\text{PE}_{\text{pai}}$, for all process expressions $E_2$ in $\text{PE}_{\text{pai}}$, for all event sequences $s$ in $\Sigma_e^+$, and for all environments $\Gamma$ in $\mathcal{E}_{nv}$, if

$$E_1 \stackrel{s_{\text{pai}}}{\rightarrow} E' \land \chi(E_1, E_2)$$

then there exists $E''$ a process expression in $\text{PE}_{\text{pai}}$ such that

$$E_2 \stackrel{s_{\text{pai}}}{\rightarrow} E'' \land \chi(E', E'').$$

5.3.4.2. **Second part of the equivalence.** This part is a little bit more complex than the previous one, since we do not only need to consider one part but all the parts of a possible process expression in $\text{PE}_{\text{pai}}$. Indeed, in the previous part we considered a transition in $m^3$ that results in $E$ and we were able to show a process expression $E'$ in $\text{pai}$ that proceeded from the same transition such that $E \subseteq E'_{\text{pai}}$. In this part, we start from a transition in $\text{pai}$. We cannot take only one part of the resulting process expression of this transition and assume that all the future transitions possible in $\text{pai}$ are possible from this part in $m^3$. For example, let $E$ be $(a | b) \parallel (a | c)$ and consider the following transition:

$$E \stackrel{(a, (\emptyset))}{\rightarrow}_{\text{pai}} (\parallel (a | c) \parallel ((a | b) \parallel \emptyset)).$$
**Quantified operations**

\[(m^2\text{-}15): (x \in s) \{ x := a \} \Gamma \quad \xrightarrow{\langle \sigma, \Gamma \rangle} \quad E'\]
\[| x \in s : E \xrightarrow{\langle \sigma, \Gamma \rangle} E' \]

\[(m^2\text{-}16): (x \in s) \{ x := a \} \Gamma \wedge (s \setminus \{ a \} \neq \emptyset) \Gamma \]
\[\quad \xrightarrow{\langle \sigma, \Gamma \rangle} \quad E' \]

\[(m^2\text{-}17): (x \in s) \{ x := a \} \Gamma \wedge (s \setminus \{ a \} = \emptyset) \Gamma \]
\[\quad \xrightarrow{\langle \sigma, \Gamma \rangle} \quad E' \]

**Environment operations**

\[(m^2\text{-}18): \quad E \xrightarrow{\langle \sigma, \Gamma \rangle} E' \quad E' \neq E \]
\[\Gamma E \xrightarrow{\langle \sigma, \Gamma \rangle} \Gamma E'\]

\[(m^2\text{-}19): \quad E \xrightarrow{\langle \sigma, \Gamma \rangle} E'\]
\[\Gamma E \xrightarrow{\langle \sigma, \Gamma \rangle} \Gamma E'\]

**Traces**

\[(m^2\text{-}2): \quad E \xrightarrow{\langle \sigma, \xi \rangle} E' \quad \sigma \neq \lambda \]
\[E \xrightarrow{\xi} E'\]

\[(m^2\text{-}3): \quad E \xrightarrow{\langle \lambda, \xi \rangle} E' \quad \sigma \neq \lambda \]
\[E \xrightarrow{\xi} E'\]

\[(m^2\text{-}4): \quad E \xrightarrow{\langle \lambda, \xi \rangle} E' \quad E' \xrightarrow{\langle \sigma, \theta \rangle} E'' \quad \sigma \neq \lambda \]
\[E \xrightarrow{\xi} E''\]

**Fig. 16.** $m^2$ rule system – 2.

The process expression $E' = (a | b) || \boxcirc$ is a part of the previous transition result. Moreover, it is a result of the execution of a from $E$ in $m^3$. But it cannot execute $c$. The process expression $E'' = \boxcirc || (a | c)$ has the same problem: it cannot execute some actions the previous transition result could. So we need to consider all the possible results of the transition in $m^3$. Actually, this is how rule system $\mathsf{PAI}$ is created from system $m^3$. Therefore, we define two functions $\mathcal{E}(E, \sigma, \Gamma)$ and $\widetilde{\mathcal{E}}(E, s)$ that return the set of all possible executions of an action $\sigma$ from a process expression $E$ in an environment $\Gamma$ and all possible executions of a sequence of actions $s$ from a process expression $E$, respectively.

**Definition 5.10.** For all process expressions $E$ in $\mathsf{PE}_{m^3}$, for all actions $\sigma$ in $\Sigma_e$ and for all environments $\Gamma$ in $\mathcal{E}nv$, $\mathcal{E}(E, \sigma, \Gamma)$ is defined as the following set:

\[\mathcal{E}(E, \sigma, \Gamma) \triangleq \{ E' | E' \in \mathsf{PE}_{m^3} \wedge E \xrightarrow{\langle \sigma, \Gamma \rangle} m^3 E' \}.\]

**Definition 5.11.** For all event sequences $s$ in $\Sigma^*_e$, $\widetilde{\mathcal{E}}(E, s)$ is defined as the following set:

\[\widetilde{\mathcal{E}}(E, s) \triangleq \{ E' | E' \in \mathsf{PE}_{m^3} \wedge E \xrightarrow{\xi} m^3 E' \}.\]

Next, we define the notion of a partition of a process expression.

**Definition 5.12.** For all process expressions $E$ in $\mathsf{PE}_{\mathsf{PAI}}$, $\xi$ is said to be a partition of $E$ if and only if for all $x \in \xi$, $x$ is a part of $E$.

Finally, we need the concept of complete partition of a process expression.

**Definition 5.13.** For all process expressions $E$ in $\mathsf{PE}_{\mathsf{PAI}}$, $\xi$ is said to be a complete partition of $E$ if and only if $\check{\chi}(\xi, E)$, where $\check{\chi}(\xi, E)$ is defined by the following properties:

1. $\xi$ is a partition of $E$ and
2. either $\xi = \{ E \}$ or
   a) if $E = \Gamma E_0$ then the set $\xi'$, defined by
   \[\xi' = \{ E' | \exists E' \in \xi \quad E' = \Gamma E_0 \wedge \check{\chi}(E_0, E_0) \},\]
   is a complete partition of $E_0$:
**Elementary operations**

1. 

\[
\begin{align*}
\sigma | \Gamma & = (\sigma', \epsilon) \in \Sigma_c \\
\sigma' & \xrightarrow{(\sigma', \epsilon)} \square
\end{align*}
\]

2. 

\[
\begin{align*}
\Gamma | \Phi & \xrightarrow{(\sigma, \epsilon)} E' \\
\Phi & \xRightarrow{(\sigma, \epsilon)} E'
\end{align*}
\]

3. 

\[
\begin{align*}
E_1 & \xrightarrow{(\sigma, \epsilon)} E_1' \\
E_1 & \cdot E_2 \xrightarrow{(\sigma, \epsilon)} E_1' \cdot E_2
\end{align*}
\]

4. 

\[
\begin{align*}
E_1 & \xrightarrow{(\sigma, \epsilon)} E_1' \\
E_1 & \cdot E_2 \xrightarrow{(\sigma, \epsilon)} E_1' \cdot E_2
\end{align*}
\]

5. 

\[
\begin{align*}
E_1 & \xrightarrow{(\sigma, \epsilon)} E_1' \\
E_1 & \cdot \epsilon \xrightarrow{(\sigma, \epsilon)} E_1' \cdot E_2
\end{align*}
\]

6. 

\[
\begin{align*}
E_1 & \xrightarrow{(\sigma, \epsilon)} E_1' \\
E_1 & \cdot E_2 \xrightarrow{(\sigma, \epsilon)} E_1' \cdot E_2
\end{align*}
\]

7. 

\[
\begin{align*}
E & \xrightarrow{(\sigma, \epsilon)} E' \\
E & \xrightarrow{(\sigma, \epsilon)} E'
\end{align*}
\]

8. 

\[
\begin{align*}
E & \xrightarrow{(\sigma, \epsilon)} E' \\
E & \xrightarrow{(\sigma, \epsilon)} E'
\end{align*}
\]

9. 

\[
\begin{align*}
E_1 & \xrightarrow{(\sigma, \epsilon)} E_1' \alpha \epsilon \notin \Delta \\
E_1 & \cdot E_2 \xrightarrow{(\sigma, \epsilon)} E_1' \cdot E_2
\end{align*}
\]

10. 

\[
\begin{align*}
E_1 & \xrightarrow{(\sigma, \epsilon)} E_1' \alpha \epsilon \notin \Delta \\
E_1 & \cdot \epsilon \xrightarrow{(\sigma, \epsilon)} E_1' \cdot E_2
\end{align*}
\]

11. 

\[
\begin{align*}
\{x_1, \ldots, x_n := a_1, \ldots, a_n\} & \xrightarrow{(\sigma, \epsilon)} E' \\
P(a_1, \ldots, a_n) & \xrightarrow{(\sigma, \epsilon)} E'
\end{align*}
\]

Fig. 17. m³ rule system – 1.

(b) and if \( E = E_1 \cdot [\Delta] \cdot E_2 \) then the set \( \xi_1 \), defined by

\[
\xi_1 = \{ E_1' \mid \exists E \in \xi \subseteq E_1' \cdot \Delta \} \cdot E_2 \}
\]

is a complete partition of \( E_1 \), and the set \( \xi_2 \), defined by

\[
\xi_2 = \{ E_2' \mid \exists E \in \xi \subseteq E_2' \cdot \Delta \} \cdot E_1 \}
\]

is a complete partition of \( E_2 \).

(c) and if \( E = E_1 \cdot E_2 \) then the set \( \xi_1 \), defined by

\[
\xi_1 = \{ E_1' \mid \exists E \in \xi \subseteq E_1' \cdot E_2 \}
\]

is a complete partition of \( E_1 \);

(d) and if \( E = E_1 \cdot E_2 \) then there exist two sets \( \xi_1 \) and \( \xi_2 \) such that \( \xi_1 \cap \xi_2 = \emptyset \) and \( \xi_1 \cup \xi_2 = \xi \) and, \( \tilde{\chi} (\xi_1, E_1) \) and \( \tilde{\chi} (\xi_2, E_2) \).

We note that \( \xi \) is a complete partition of \( E \) with \( \tilde{\chi} (\xi, E) \) because \( \tilde{\chi} \) can be seen as an extension of the predicate \( \chi \) of this part of the demonstration. Intuitively, a set \( \xi \) is a complete partition of \( E \) if all the elements of \( \xi \) (which are parts of \( E \)) can be combined to form \( E \). The combination is done by collapsing the parts of the syntax tree for the elements of \( \xi \) that are the same and grouping the others with choice operators.

For example, let \( E \) be

\[
\begin{align*}
& (b \parallel (a \cdot c) \cdot a \cdot d) \parallel (a \cdot b \parallel (c \cdot d))
\end{align*}
\]

Then the set of the three process expressions \( E_1, E_2, E_3 \) with

\[
\begin{align*}
E_1 & = b \parallel (a \cdot c) \cdot a \cdot d \\
E_2 & = a \cdot b \parallel c \\
E_3 & = a \cdot b \parallel d
\end{align*}
\]

is a complete partition of \( E \). Indeed, \( \{ E_1 \} \) is a complete partition of the left part of the choice of \( E \), and \( \{ E_2, E_3 \} \) is a complete partition of the right \( (E') \) part of \( E \). The latter assertion is true because the left part of the synchronization operator is the same in both \( E_2 \) and \( E_3 \). Moreover, this is the same as the left part of the synchronization operator of \( E' \), and the right part of \( E' \) is a choice between the right parts of \( E_2 \) and \( E_3 \), Fig. 12 shows a graphical representation of this combination.

From the definition of \( \tilde{S}, \tilde{S} \) and \( \tilde{\chi} \) it is easy to claim the main lemma and the main theorem for this part. If we look again at the previous example, we see that \( E \) is the result of the execution of \( a \) from the process expression \( E_0 \) in the system \( \text{Pai} \), with

\[
E_0 = a \cdot b \parallel (a \cdot c) \cdot a \cdot d.
\]
Quantified operations

\[ (x \in s) [x := a] [\Gamma] \quad \{x := a\} E \xrightarrow{\sigma, \Gamma} E' \]

\[ | x \in s : E \xrightarrow{\sigma, \Gamma} E' \]

\[ (x \in s) [x := a] [\Gamma] \land (s \setminus \{a\} \neq \emptyset) [\Gamma] \]

\[ \{x := a\} E \| (|\Delta| \land (s \setminus \{a\} : E)) \xrightarrow{\sigma, \Gamma} E' \]

\[ |\Delta| \land (s \setminus \{a\} : E) \xrightarrow{\sigma, \Gamma} E' \]

Environment operations

\[ E \xrightarrow{\sigma, \Gamma, \Delta_0} E' \quad E' \neq \emptyset \]

\[ \Gamma E \xrightarrow{\sigma, \Gamma_0} \Gamma E' \]

Traces

\[ (m^3-1): \quad E \xrightarrow{\sigma, \emptyset} E' \]

\[ E \sim_\sigma E' \]

\[ (m^3-2): \quad E \xrightarrow{\sigma, \emptyset} E' \]

\[ E \sim_\sigma E'' \]

Fig. 18. \( m^3 \) rule system 2.

There are three possible executions of \( a \) from \( E_0 \) in system \( m^3 \), with \( E_1, E_2 \) and \( E_3 \) as results. So, we can say that

\[ \Xi(E_0, a, \emptyset) = \{ E_1, E_2, E_3 \} \]

and therefore \( \Xi(E_0, a, \emptyset) \) is a complete partition of \( E \). To be more general, if we have a complete partition of \( E_0 \), we want the results of all executions from all elements in this complete partition to form a complete partition of \( E \). Consequently, the lemma is stated as follows.

Lemma 29. For all process expressions \( E_1 \) and \( E'_1 \) in \( PE_{\text{PAI}} \), for all actions \( \sigma \) in \( \Sigma_e \), for all environments \( \Gamma \) in \( E_{nv} \), and for all complete partitions \( \xi \) of \( E_1 \) if

\[ E_1 \xrightarrow{\sigma, \Gamma} E'_1 \]

then there exists a complete partition \( \xi' \) of \( E'_1 \) such that

\[ \xi' = \bigcup_{E_2 \in \xi} \Xi(E_2, \sigma, \Gamma). \]

This lemma is proved by induction on the size (number of elements) of the complete partition.

Therefore, the main theorem can be proved.

Theorem 30. For all process expressions \( E_1 \) and \( E'_1 \) of \( PE_{\text{PAI}} \), for all sequence of events \( s \) of \( \Sigma_e^* \), and for all complete partitions \( \xi \) of \( E_1 \) if

\[ E_1 \xrightarrow{\Sigma_e, s} E'_1 \]

then there exists a complete partition \( \xi' \) of \( E'_1 \) such that

\[ \xi' = \bigcup_{E_2 \in \xi} \Xi(E_2, s). \]

This completes the proof that \( m^3 \) is trace-equivalent to \( \text{PAI} \).

Corollary 31. For all \( E \in PE_{\text{init}} \), \( E^{m^3} = E_{\text{PAI}} \).

Therefore we can conclude that \( \text{FSD} \) is trace-equivalent to \( \text{PAI} \).

Corollary 32. For all \( E \in PE_{\text{init}} \), \( E^{\text{FSD}} = E_{\text{PAI}} \).
Elementary operations (1)

(PM-1): \[ \sigma \left[ \Gamma \right] = \sigma' \wedge \sigma' \in \Sigma \quad \sigma \xrightarrow{\left( \sigma, \Gamma \right)} \sigma' \]

(PM-2): \[ \Gamma \left[ \Phi \right] = E \xrightarrow{\left( \Phi, \Gamma \right)} E' \quad \Psi \implies E \xrightarrow{\left( \Phi, \Gamma \right)} E' \]

(PM-3): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_1' \neq \| \wedge \left( E_2 \uparrow \left( \sigma, \Gamma \right) \wedge E_1 \| \right) \]

(PM-4): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} \| \wedge \left( E_2 \| \wedge E_1 \| \right) \]

(PM-5): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_2' \quad E_1' \neq \| \wedge \left( \exists F \| \wedge \left( E_2 \| \wedge E_1 \| \right) \right) \]

(PM-6): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_2' \quad E_1 \| \implies \quad E_2 \| \]

(PM-7): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_2' \quad E_1 \| \wedge E_2 \| \]

(PM-8): \[ E \xrightarrow{\left( \sigma, \Gamma \right)} E' \quad E \xrightarrow{\left( \sigma, \Gamma \right)} E'' \]

(PM-9): \[ E \xrightarrow{\left( \sigma, \Gamma \right)} E' \quad E' \neq \| \]

(PM-10): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_2' \]

(PM-11): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_2' \quad E_1 \| \wedge E_2 \| \]

Fig. 19. PAT rule system – 1.

Elementary operations (2)

(PM-12): \[ \left\{ x_1, \ldots, x_n \right\} \equiv \left\{ a_1, \ldots, a_n \right\} \quad E \xrightarrow{\left( \sigma, \Gamma \right)} E' \quad P(x_1, \ldots, x_n) \equiv E \]

(PM-13): \[ P(a_1, \ldots, a_n) \xrightarrow{\left( \sigma, \Gamma \right)} E' \]

(PM-14): \[ E_1 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \quad E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_2' \quad \alpha \sigma \notin \Delta \]

(PM-15): \[ E_1 \| \Delta \| E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \| \Delta \| E_2' \]

(PM-16): \[ E_1 \| \Delta \| E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \| \Delta \| E_2' \quad \alpha \sigma \in \Delta \]

(PM-17): \[ E_1 \| \Delta \| E_2 \xrightarrow{\left( \sigma, \Gamma \right)} E_1' \| \Delta \| E_2' \quad \alpha \sigma \notin \Delta \]

Fig. 20. PAT rule system – 2.
6. Conclusion

In this paper, we have presented a set of rules for the $\mathcal{EB}^3$ process algebra which is well adapted for efficient symbolic computation of process expressions. This new set of rules tackles three problems: non-determinism, internal actions, and substitution. We have shown that it is trace-equivalent to the original set of rules proposed in the semantics of $\mathcal{EB}^3$.

The $\mathcal{EB}^3\text{PAI}$ interpreter implements this new set of rules. Taking into account the additional optimization techniques presented in \cite{8}, the algorithmic complexity of $\mathcal{EB}^3\text{PAI}$ favorably compares with programmer-made implementations for all the patterns identified in \cite{2}. The overhead of $\mathcal{EB}^3\text{PAI}$ is linear in terms of the size of the specification text. In terms of actual response time, $\mathcal{EB}^3\text{PAI}$ is definitely slower than a programmer-made implementation, but for IS with low transaction rates, it could be acceptable.

The main cost factor for $\mathcal{EB}^3\text{PAI}$ is disk I/O. For the sake of implementation simplicity, $\mathcal{EB}^3\text{PAI}$ uses an OO database to store the process expression. Navigation in the process expression requires $O(s)$ disk reads, where $s$ is the size of the specification text, whereas the programmer-made implementation requires $O(k)$, where $k$ is the number of entities referenced by the transaction. Even for a simple specification, $k$ is significantly smaller than $s$, which explains the relatively poor response time of $\mathcal{EB}^3\text{PAI}$. A solution to this problem resides in the development of a custom persistence management system for $\mathcal{EB}^3$ ASTs, which would be more efficient than a general purpose OO database.

A solution is to develop an efficient persistency manager for process expressions based on the following observation. A process expression $E$ can be decomposed into two parts: the skeleton of $E$; and the quantification sets of $E$ which implement the quantified interleaves. Since the size of the skeleton is roughly constant (i.e. $O(s)$), it can be easily stored in a single record in a database and kept in main memory between transactions, to avoid disk reads. The second part, quantification sets, can be very large, because quantification sets denote entities (e.g. individual books, members, or loans). Each entity must be stored in a separate record to be efficiently accessed.

Another performance improvement is to transform process expressions into automata when the process expression induces a finite LTS. In that case, we hope to represent the LTS in a compact form, which we call an algebraic state transition diagram (ASTD) \cite{45}. An ASTD is an LTS augmented with hierarchical states to efficiently model parallel composition and avoid combinatorial explosion. It seems quite feasible to address deterministic, tail recursive process expressions in this way.

$\mathcal{EB}^3\text{PAI}$ is a powerful rapid development tool which can be used when users need hands-on experience with a real system to effectively determine a suitable set of requirements. It is an interesting alternative to agile development, which

### Quantified operations

\begin{align*}
\text{(PAI-18):} & \quad (x \in s) [x := a] [\Gamma] \land (\{ x \in s \backslash \{a\} : E \} \triangleright \{x := a\} E \xrightarrow{\{x := a\} \Gamma} E') \\
& \quad | x \in s : E \xrightarrow{\{x := a\} \Gamma} E' \quad | x \in s \backslash \{a\} : E \xrightarrow{\{x := a\} \Gamma} E' \\
\text{(PAI-19):} & \quad x \in s) [x := a] [\Gamma] \land (\{ x \in s \backslash \{a\} : E \} \triangleright \{x := a\} E \xrightarrow{\{x := a\} \Gamma} E) \\
& \quad | x \in s : E \xrightarrow{\{x := a\} \Gamma} E_1 \quad | x \in s \backslash \{a\} : E \xrightarrow{\{x := a\} \Gamma} E_2 \\
\text{(PAI-20):} & \quad (x \in s) [x := a] [\Gamma] \land (\{ x \in s \backslash \{a\} : E \} \triangleright \{x := a\} E \xrightarrow{\{x := a\} \Gamma} E') \\
& \quad | \{x := a\} E \xrightarrow{\{x := a\} \Gamma} E' \\
\text{(PAI-21):} & \quad (x \in s) [x := a] [\Gamma] \land (\{ x \in s \backslash \{a\} = \emptyset \} [\Gamma] \triangleright \{x := a\} E \xrightarrow{\{x := a\} \Gamma} E') \\
& \quad | \{x \in s \backslash \{a\} = \emptyset \} \xrightarrow{\{x := a\} \Gamma} E' \\
\end{align*}

### Environment operations

\begin{align*}
\text{(PAI-22):} & \quad E \xrightarrow{\{x := a\} \Gamma} E' \quad E' \neq \emptyset \\
& \quad \Gamma E \xrightarrow{\{x := a\} \Gamma} \Gamma E' \\
\text{(PAI-23):} & \quad E \xrightarrow{\{x := a\} \Gamma} E' \\
& \quad \Gamma E \xrightarrow{\{x := a\} \Gamma} \Gamma E' \\
\end{align*}

### Traces

\begin{align*}
\text{(PAI-1):} & \quad E \xrightarrow{\{x := a\} \Gamma} E' \\
& \quad E \xrightarrow{\{x := a\}} E' \\
\text{(PAI-2):} & \quad E \xrightarrow{\{x := a\} \Gamma} E' \\
& \quad E' \xrightarrow{\{x := a\}} E'' \\
\end{align*}

**Fig. 21.** $\text{PAI}$ rule system — 3.
![Table](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAIcAAAD2CAYAAAAzIFi1AAAABGdBAQABIGAAQAAAABGR0e30AAAtAAAAA49/fGQABfjAAAAH3RSTlMAQiQAAAABJRU5ErkJggg==)

The table shows the relations between rules of the five systems. The rules are denoted as follows:

- **FSD**: Free-Style Development
- **PAI**: Process Algebra Implementation

**Atomic rule**

<table>
<thead>
<tr>
<th>FSD - 1</th>
<th>FSD - 2</th>
<th>FSD - 3</th>
<th>FSD - 4</th>
<th>PAI - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
</tr>
</tbody>
</table>

**Process call**

<table>
<thead>
<tr>
<th>FSD - 13</th>
<th>FSD - 14</th>
<th>FSD - 15</th>
<th>FSD - 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
</tr>
</tbody>
</table>

**Quantif.**

<table>
<thead>
<tr>
<th>FSD - 17</th>
<th>FSD - 18</th>
<th>FSD - 19</th>
<th>FSD - 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
</tr>
</tbody>
</table>

**Environment**

<table>
<thead>
<tr>
<th>FSD - 21</th>
<th>FSD - 22</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢</td>
<td>⊢</td>
</tr>
</tbody>
</table>

**Trace**

<table>
<thead>
<tr>
<th>FSD - 1</th>
<th>FSD - 2</th>
<th>FSD - 3</th>
<th>FSD - 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
<td>⊢</td>
</tr>
</tbody>
</table>

**References**
