Efficient Symbolic Execution of Large Quantifications in a Process Algebra

Benoît Fraikin and Marc Frappier

Département d'informatique, Université de Sherbrooke, Québec, Canada
{Benoit.Fraikin,Marc.Frappier}@Usherbrooke.CA

Abstract. This paper describes three optimization techniques for a process algebra interpreter called EB^{3}PAI. This interpreter supports the EB^3 method, which was developed for the purpose of automating the development of information systems using *efficient symbolic execution* of abstract specifications. The proposed optimizations allow an interpreter to execute actions on a quantified choice in constant time and on a quantified parallel composition in logarithmic time with respect to the number of entities in a quantified entity type. This time complexity is comparable to that of programmer-derived implementation of process expressions and significantly better than the time complexity of common process algebra simulators, which execute quantifications by computing their expansion into binary expressions.

1 Introduction

Process algebras have now been recognized has excellent modelling notations for specifying system behavior. They are used in various areas such as telecom systems, control systems, business processes and web services. In the APIS project [1,2], they are used to specify information systems (IS). The APIS platform supports the EB^3 method [3], which was designed for the specification of IS. This platform includes a symbolic interpreter, called EB^{3}PAI, to efficiently execute process expressions of IS specifications. Its goal is to reach a level of efficiency comparable to hand-made implementations, thereby avoiding the implementation of these process expressions, which represents a significant increase in software development productivity.

In developing an efficient interpreter for IS process expressions, one has to deal with quantified (also called indexed or replicated) process expressions, mainly for choice quantification and parallel composition quantification. For example, the process expression

$$||| \, x \in 1..m \, : \, \mathbf{P}\,(\,x\,)$$

denotes the expanded process expression

$$\mathbf{P}\,(\,1\,) \; ||| \; \ldots \; ||| \; \mathbf{P}\,(\,m\,).$$

Existing process algebras simulators like PROBE [4] and CIA [5] for CSP, the simulator in the μCRL tool set [6], and CADP's OCIS [7] for LOTOS, are executing quantifications (or an equivalent feature) by expanding them. Code generators like JCircus [8,9] translates a Circus [10] specification into a Java program

M. Butler, M. Hinchey, M.M. Larrondo-Petrie (Eds.): ICFEM 2007, LNCS 4789, pp. 327–344, 2007.
© Springer-Verlag Berlin Heidelberg 2007

using JCSP [11]. A quantification is also expanded and each interleaved process is implemented by a separate thread.

Expansion of quantifications is not acceptable for the execution of IS process specifications, because the size of the quantification set is typically huge (e.g., $m \geq 10^{10}$). Constant m denotes the maximum value of a key of an entity in an IS (e.g., a book id in a library system).

In this paper we propose techniques, called κ-optimization, to efficiently execute quantifications in a process expression. These techniques apply to several recurrent patterns of IS specifications which are found in EB^3 specifications and defined in [3]. Let n denote the number of entities in an entity type (e.g., the number of books in a library system). When the sufficient conditions are met, our algorithms can execute choice quantifications in constant time and quantified parallel composition in $\mathcal{O}(log(n))$ or in constant time, depending on the implementation used for a map (B-tree or hash table). A programmer derived (i.e., hand-made) implementation of these process expressions has a comparable time complexity, although it is more efficient since there is less overhead than for symbolic execution. Our algorithms are more efficient than existing process algebra simulators, since they expand quantifications, which means that their time complexity is linear ($\mathcal{O}(m)$) for both quantified choice and quantified parallel composition. Note that usually m is quite greater than n, since m denotes the upper bound for the value of an entity key, hence it denotes the maximum number of entities, whereas n denotes the number of entities currently existing in the system. Our algorithms are also more efficient than code generation in JCircus, since it requires m threads to implement a quantification.

The proposed algorithms are defined for the EB^3 process algebra, but they could probably be adapted for other process algebras like CSP, μCRL, FSP and Circus, which include quantified operators. LOTOS does not include quantification; it must be simulated using recursion.

Our algorithms are suitable for process algebra *simulators*, but not for *model checkers*. A simulator executes actions as requested by the environment. It explores only the execution path that the environment commands during execution. Simulators are typically used for specification animation and validation with users. The objective of EB^{3}PAI is to increase the efficiency of simulators to use them as an implementation of a specification.

Model checkers are addressing another issue for which our algorithms are not relevant. They are used to verify that a process expression satisfy a given property. The property is checked by exploring the entire transition system of the process expression; hence expansion of quantification is necessary since each individual process may have to be checked. Typical examples of model checkers include FDR2 [12] for CSP, the Concurrency Workbench [13] for CCS, ProB [14] for a combination of CSP and B, the model checking tools in the μCRL tool set [6], LTSA [15] for FSP, and CADP's EVALUATOR [7] for LOTOS.

This paper is structured as follows. Section 2 provides a brief overview of the EB^3 process algebra method and describes the general idea of symbolic execution with EB^{3}PAI. Section 3 is the main part of this paper. It describes the

optimization of large quantified expressions. Finally, Section 4 analyzes the space and time complexity of the symbolic execution algorithm and provides some experimental results showing the actual performance of its implementation, EB^{3}PAI. Section 5 concludes with some remarks and future work on improvements to EB^{3}PAI.

2 The EB^3 Process Algebra and Symbolic Execution

The EB^3 process algebra is inspired from regular expressions, CSP [16], CCS [17], ACP [18] and LOTOS [19]. Its syntax has been simplified in order to streamline IS specification. The reader may consult [3,20,21] for additional details and a thorough comparison with these process algebras.

2.1 Syntax

A process expression is defined over a set of symbols Σ, 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 Σ_e is the set of ground actions from Σ, i.e., those with no variable; it is called the *input event set*. Set Σ_l denotes the set of labels of actions in Σ.

The process expressions over Σ are defined recursively as follows. Elements of $\Sigma \cup \{\lambda\}$ represent *elementary* process expressions over Σ. The symbol $\boxdot$, called "box", is an elementary process expression denoting successful completion. Let E, E_1, and E_2 be process expressions over Σ, $n \in \mathbb{N}$, $\Delta \subseteq \Sigma_l$ and Φ be a formula. The expressions E^*, E^+, E^n, $E_1 \centerdot E_2$, $E_1 \mid E_2$, $E_1|[\Delta]|\,E_2$, $E_1 \parallel E_2$, $E_1 \mid\mid\mid E_2$, $\Phi \Longrightarrow E$, and $(\![x := t_1, \ldots, x := t_n]\!)E$ are process expressions over Σ. Operations *, $^+$, ^{n}, and $\centerdot$ denote the usual Kleene closure, positive closure, and concatenation of regular expressions. Operation $\mid$ is a choice between E_1 and E_2; it is drawn from regular expressions and CSP [16]. Operation $|[\Delta]|$ is the parameterized parallel composition of E_1 and E_2 with synchronization on actions whose labels belong to Δ; 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 Δ must be executed by both E_1 and E_2. Actions in Δ are said to be synchronized. Operations $\mid\mid\mid$ and $\parallel$ are the interleave and parallel composition of CSP [16], respectively; they are special variants of $|[\;]|$: $E_1 \mid\mid\mid E_2$ 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.*, $E_1|[\alpha(E_1) \cap \alpha(E_2)]|\,E_2$, where the operator α denotes the alphabet (set of labels) of a process expression. The operator α is defined recursively on the structure and returns the set of all the action labels occurring in a process expression but λ. The process expression $\Phi \Longrightarrow E$ is the guard of E by Φ: it means that E can execute an action if and only if Φ is true. The process expression $(\![x_1 := t_1, \ldots, x_n := t_n]\!)E$ is called an environment and it denotes the simultaneous substitution of $x_1, \ldots, x_n$ by $t_1, \ldots, t_n$ in E. The special symbol λ denotes an internal action that a process may execute without requiring input from the system's environment. It plays a role similar that of the empty word ϵ in regular expressions or the unobservable action τ in

CCS and `i` in LOTOS. The EB^3 process algebra also allows *quantification* (also called *indexing* or *replication* in CSP) over operators $|, |[\Delta]|, |||$. For instance, the process expression $|\, x \in 1..n : \mathbf{P}(x)$ denotes $\mathbf{P}(1) \mid \mathbf{P}(2) \mid \cdots \mid \mathbf{P}(n)$. Quantifications are restricted to finite sets.

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

2.2 An Example

To illustrate our optimizations, consider the specification provided by Figure 1. It shows the process **main** and auxiliary process definitions for a simple library system. The rest of the EB^3 specification is omitted, since it is not relevant to illustrate our optimization techniques. Figure 2 provides the entity-relationship diagram of this specification.

There are two entity types (book and member) and one association (loan), each modelled by a process expression. A book must be acquired in order to be used in the library, and a member must join it. Books and members are identified by a number (bId and mId). A member can borrow a book, renew it as many times as he wants and, finally, return it to the library. While a member is borrowing a book, no other member can borrow it. Other usual properties of loans are represented by this specification. The behavior of each single entity or

---

main ( )=
  ($|||\, bId \in$ BOOKID : $\mathbf{book}(bId)^*$)
  $||$
  ($|||\, mId \in$ MEMBERID : $\mathbf{member}(mId)^*$)

book (bId : BOOKID) =
  Acquire(bId)$\centerdot$
  ($|\; mId \in$ MEMBERID : $\mathbf{loan}(mId, bId)$)$^*\centerdot$
  Sell(bId) ;

loan (mId : MEMBERID, bId : BOOKID) =
  Lend(mId, bId)$\centerdot$
  Renew(bId)$^*\centerdot$
  Return(bId) ;

member (mId : MEMBERID) =
  Join(mId)$\centerdot$
  ($|||\, bId \in$ BOOKID : $\mathbf{loan}(mId, bId)$)$^*\centerdot$
  Leave(mId) ;

Process expression **main** and process definitions

Fig. 1. EB^3 specification example

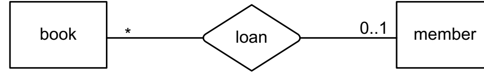

Fig. 2. Entity-relationship diagram of the library system

association (a book, a member, or a loan) is defined by the corresponding process definition. The system is defined using quantifications that, on the one hand, allow for multiple entities (quantified interleave), and on the other hand, model the cardinality of the association (quantified choice for `0..1` and a quantified interleave for `*`). In the following, an *entity* is an instance of a process that models an IS entity like **book** or **member**. An *association* is an instance of a process that models an IS association like **loan**. An entity type is represented in EB^3 by a quantification over all possible entities. For example,

$$|||\, mId \in \text{MEMBERID} \, : \, \textbf{member}\,(\, mId\,)^{*}$$

represents the entity type member. Process **member** (1) represents the member entity with $mId = 1$. Additional explanations and a more complex example of a library can be found in [22].

It is important to note that quantification is a crucial operator in IS specification. This constitutes a major difference from other problem domains where process algebras are typically used (protocol specification for example). Since the main aim of EB^3 is to provide an executable specification, the specification style used to achieve this goal is also different.

2.3 Symbolic Execution of EB^3 Process Expressions

EB^{3}PAI is a symbolic interpreter. It executes the inference rules of an operational semantics defined in the CCS style [17]. The original semantics of the EB^3 process algebra has been defined in [3]. A new operational semantics, optimized for symbolic execution, has been defined in [20,21]. EB^{3}PAI is based on this semantics. We provide below an outlook of the symbolic execution strategy. For more details, the reader may consult [20,21].

Given a process expression P and an action σ, one can compute the possible transitions and resulting process expressions (PEs) 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 $\frac{E_2 \xrightarrow{\sigma} E_2'}{E_1 \xrightarrow{\sigma} E_1'}$. When a match is found, the rule's premiss, which are themselves transitions (e.g., $E_2 \xrightarrow{\sigma} E_2'$), induce a recursive search. Ultimately, the search reaches a rule which doesn't have a transition in its premiss. Then, the resulting process expression Q is constructed by backtracking over the inference rules through termination of recursive search calls.

In summary, EB^{3}PAI executes a specification by simply evaluating the inference rules. We do not generate code per se; EB^{3}PAI can be considered as a virtual

machine and each specification becomes a high-level program. The implementation is the combination of EB^{3}PAI and the specification.

3 Optimizations for Symbolic Execution

3.1 Optimizing Quantification Execution Time: Direct κ-Optimization

The Problem of Large Quantification. The EB^3 language allows the use of quantification operators. For example,

$$|\, x \in 1..10^{10} \,:\, |||\, y \in 1..10^{10} \,:\, \mathsf{a}(x, y) \,.\, \mathsf{b}(y, x) \tag{1}$$

A basic approach to executing quantification operators is to iterate over the values of the quantification set to determine whether a transition is feasible. It is clear that such a linear search through a large set is too ineffective to be acceptable. Moreover, the execution of a quantified interleave generates large interleave expressions. For instance, if the process defined by (1) has executed $\mathsf{a}(2, 1), \ldots, \mathsf{a}(2, 10^9)$, the resulting PE is

$$\begin{array}{l} \quad (\![y := 1, x := 2]\!)\mathsf{b}(y, x) \\ \quad ||| \\ \quad \ldots \\ \quad ||| \\ \quad (\![y := 10^9, x := 2]\!)\mathsf{b}(y, x) \\ ||| \\ \quad \big(|||\, y \in 1..10^{10} - 1..10^9 \,:\, \mathsf{a}(x, y) \,.\, \mathsf{b}(y, x)\big). \end{array}$$

When action $b(10^9, 2)$ must be executed, another linear search must be done over the interleave composition, which is also too inefficient.

To optimize these executions, we determine by static analysis of each quantified expression which value of the quantified set must be selected based on the parameters of the action to execute. We call these values **κ-values**, the positions of the values in the action parameters **κ-positions**, and this method **direct κ-optimization.**

For instance, in process expression (1), we can determine that when

$$|\, x \in 1..10^{10} \,:\, \ldots$$

must execute $\mathsf{a}(t_1, t_2)$, the only execution feasible is with $x = t_1$; similarly, the only execution feasible for

$$|||\, y \in 1..10^{10} \,:\, \ldots$$

is with $y = t_2$. Hence, whenever possible, we determine a map $\Pi : \Sigma \to T$ for each quantified expression $\Phi x \in T : E$ such that $(\![x := \Pi(\sigma)]\!)E$ is the only candidate to execute a transition with σ. Specifically, we determine the position of the quantification variable within the parameters of each action.

These positions, called κ-positions, are determined by static analysis before the execution of a specification. Let $\kappa(\chi, E)$ be the κ-position of χ-labelled actions in E, where E is a quantified interleave operator or a quantified choice operator. Then, if we need to optimize the quantification E for any event σ,

$$\Pi(\sigma) \triangleq \pi_{\kappa(\alpha(\sigma),E)}(param(\sigma))$$

where $\pi_i((x_1, \ldots, x_i, \ldots, x_n)) = x_i$, $\alpha(\sigma)$ is the label of σ and $param(\sigma)$ is the tuple of the parameters of σ.

This approach is sufficient to optimize a choice quantification, since the quantification disappears after the transition. In the case of a quantified interleave, the quantification remains in the result process expression, since it can spawn one interleave process for each value in the quantification set. For example, the execution of $\mathsf{a}(1)$ from the process expression

$$|\, x \in [1..3] \,:\, \mathsf{a}(x) \,.\, \mathsf{b}(x)$$

returns $([x := 1])\mathsf{b}(x)$. The execution of the same event on

$$|||\, x \in [1..3] \,:\, \mathsf{a}(x) \,.\, \mathsf{b}(x)$$

returns

$$\big(([x := 1])\mathsf{b}(x)\big) \;|||\; \big(|||\, x \in [2..3] \,:\, \mathsf{a}(x) \,.\, \mathsf{b}(x)\big).$$

The interleave of the instantiated process expressions (i.e.,, $P(t_1) \,|||\, \ldots \,|||\, P(t_n)$) is represented by a function $K : T \to \mathcal{PE}$ such that $K(\Pi(\sigma))$ is the only process expression that can execute σ.

Figure 3 describes the function $find_\kappa(E, x, \text{EP})$, which determines a κ-position for a variable x in a process E. The parameter EP is used to keep track of the process definitions that have been parsed so far over recursive calls; it is set to $\emptyset$ on the initial call. The function $find_\kappa$ returns a relation between action labels and $\mathbb{N} \cup \{\bot\}$. The symbol $\dashv$ is used as a marker to detect overlapping quantifications on the same variable; the algorithm returns $\bot$ for actions within the scope of these overlapping quantifications. When an action does not contain x in its parameters, $\bot$ is also returned. A quantification is κ-optimizable if the result of $find_\kappa$ is a deterministic relation and does not include $\bot$ in its codomain (i.e., x occurs in the same position for each occurrence of an action in E). A choice quantification is also partially κ-optimizable when the image set of the action to execute is a singleton; for other actions which include either $\bot$ or several κ-positions, κ-optimization is not applicable.

This algorithm, which is part of a static analysis of the specification prior to its execution, is applied on every quantified process expression. Its algorithmic complexity is $\mathcal{O}(N_E)$, where N_E is the number of nodes in the syntax tree of E. This number is usually small ($N_E < 100$). It does not depend on the number of entities involved in E. At runtime, the algorithmic complexity of retrieving the instantiated interleave process depends on the implementation chosen for map K; databases usually offer either hash tables or B-trees, which means constant or logarithmic access time. As for space complexity, a process

$find_\kappa(E, x, \text{EP}) \triangleq$ `match` E `with`

$(\![z_1 := y_1 \dots z_n := y_n]\!)E_0$ `->` `if` $\exists_i (y_i = x)$ `then` $t := z_i$ `else` $t := x$ `endif;`
  `return` $find_\kappa(E_0, t, \text{EP})$,

λ `->` `return` $\emptyset$,

$\mathsf{a}(y_1, \dots, y_n)$ `->` `if` $\exists_i (y_i = x)$ `then` $j = i$ `else` $j = \perp$ `endif;`
  `return` $\{ (\mathsf{a}, j) \}$,

$E_1 \Phi E_2$ `->` `return` $find_\kappa(E_1, x, \text{EP}) \cup find_\kappa(E_2, x, \text{EP})$,

$\varphi \Longrightarrow E_0$ `->` `return` $find_\kappa(E_0, x, \text{EP})$,

$\Phi(E_0)$ `->` `if` Φ `is a quantification on` x
  `then` $t := \dashv$ `else` $t := x$ `endif;`
  `return` $find_\kappa(E_0, t, \text{EP})$,

$Q(y_1, \dots, y_n)$ `->` `if` $Q \in \text{EP}$ `then return` $\emptyset$ `else`
  `let` E_0 `be the definition of` $Q(x_1, \dots, x_n)$ `;`
  `return` $find_\kappa((\![x_1 := y_1 \dots x_n := y_n]\!)E_0, x, \text{EP} \cup \{Q\})$
  `endif.`

Fig. 3. An algorithm that computes κ-positions

expression $|||x \in T : E$ requires a map and only one instance of E for all map entries, because the environment $(\![x := \Pi(\sigma)]\!)$ is represented by a map entry. Each process expression E' reachable from E by transition execution is also instantiated only once, which is very efficient.

Example. Consider the library specification in Figure 1. The algorithm of Figure 3 has to be applied on four quantified process expressions:

1. $find_\kappa(\mathbf{book}\,(bId)^*, bId, \emptyset)$ in the **main** process definition;
2. $find_\kappa(\mathbf{loan}\,(mId, bId)^*, mId, \emptyset)$ in the **book** process definition;
3. $find_\kappa(\mathbf{loan}\,(mId, bId)^*, bId, \emptyset)$ in the **member** process definition;
4. and $find_\kappa(\mathbf{member}\,(mId)^*, mId, \emptyset)$ in the **main** process definition.

Once the computation is done, we obtain the following results:

1. $\{ (\mathsf{Acquire}, 1), (\mathsf{Lend}, 2), (\mathsf{Renew}, 1),$
   $(\mathsf{Return}, 1), (\mathsf{Sell}, 1) \}$
2. $\{ (\mathsf{Lend}, 1), (\mathsf{Renew}, \perp), (\mathsf{Return}, \perp) \}$
3. $\{ (\mathsf{Lend}, 2), (\mathsf{Renew}, 1), (\mathsf{Return}, 1) \}$
4. $\{ (\mathsf{Join}, 1), (\mathsf{Lend}, 1), (\mathsf{Renew}, \perp),$
   $(\mathsf{Return}, \perp), (\mathsf{Leave}, 1) \}$

The first and third quantifications are κ-optimizable. Therefore, to execute the action $\mathsf{Lend}(1, 2)$ from $|||\, bId \in \text{BOOKID} : \mathbf{loan}\,(mId, bId)$, we can directly try to execute $\mathsf{Lend}(1, 2)$ from the process expression $(\![bId := 2]\!)\mathbf{loan}\,(mId, bId)$, instead of trying every value of BOOKID for bId until $bId = 2$ is found. The fourth

quantification is not κ-optimizable since it is an interleave and the codomain of the result contains two occurrences of $\perp$ (for Renew and Return), because mId is not a parameter of these actions. The second quantification is a choice. It is partially κ-optimizable: κ-optimization can be used for a Lend, but not for a Renew or a Return. Hence, the κ-optimization is not totally satisfactory. The next section addresses this issue.

3.2 Extending Quantification Optimization: Indirect κ-Optimization

We have found conditions under which a quantification can be optimized when the algorithm in Figure 3 fails to find a single κ-position for each action. These conditions cover a large number of IS specification patterns described in [3]. Hence, our interpreter can optimize the execution of quantified interleaves in most common IS specifications. Let us start by providing the intuition behind this second optimization.

Example. Consider the example in Figure 1 and the action Renew, which cannot be optimized by the algorithm in Figure 3. Intuitively, one can see that when a loan is initiated, the action (Lend(mId, bId)) binds book bId to member mId. Since a book can only be borrowed by one member at a time, and since a renew can only occur after a book is borrowed, bId is sufficient to deduce mId; hence, actions Renew and Return do not need to include mId as a parameter, because of this binding between a borrowed book and its borrower. In entity-relationship data modeling, we say that loan is a one-to-many relationship between members and books: a member can borrow several books concurrently, but a book is borrowed by at most one member at any given time. Hence, there is a functional dependency from entity type book to entity type member.

The first question that must be raised is how exactly one can deduce, solely by static analysis of the process expression, that there exists a functional dependency between book and member. Next, we have to determine under what conditions such a dependency can be found.

In our example, one can deduce the functional dependency between a book and a member from the position of the choice quantification in the process **book**:

book (bId : BOOKID) =
  Acquire(bId) .
  (| $mId \in$ MEMBERID : **loan** (mId, bId))* .
  Sell(bId)

Indeed, the choice quantifier implies that one book (with the number bId) can be borrowed by only one member at a time. To be lent to another member, the execution of process **loan** has to be completed. If we closely examine the **loan** process expression,

loan (mId : MEMBERID, bId : BOOKID) =
  Lend(mId, bId). Renew(bId)*. Return(bId)

we see that it is made of three parts : a producer (Lend), a modifier (Renew) and a consumer (Return). This is a classical IS pattern described in [3]. The producer is the action that binds mId and bId. The consumer is the action that tells us that the bond is no longer active. Since the interleave quantification to optimize for action Renew and Return is synchronized over actions of **loan** with this quantified choice expression, we know which mId can execute Renew and Return from the process expression **loan** (mId, bId).

Using this example, we can summarize the general idea of indirect κ-optimization, as follows.

During static analysis:

1. Find the quantified choice operators to deduce the possible functional dependencies (below we refer to choice quantified variables occurring in the scope of other quantifications (interleave or choice) as the *dependent variables* and the enclosing quantified variables as the *keys*).
2. For all actions not optimized with the algorithm in Figure 3, identify the producer that binds the keys (bId in the example) to the dependent variable (mId in the example) under the condition that the choice and interleave quantifications to optimize are synchronized on these actions.

At runtime:

1. When a producer is executed, store the value of the functional dependency between the set of keys and the dependent variable.
2. Store the value of the new process expression for the operand of the quantified interleave in a mapping K, as for direct κ-optimization.
3. When a consumer is executed, delete the stored value of the functional dependency between the keys and the dependent variable.
4. Accept or reject an action using the value of the functional dependency and mapping K. If the value of the functional dependency is not initialized, then reject the modifier or the consumer; if it is initialized, then check if the corresponding process expression in mapping K can execute the action.

Functional Dependencies. The first part of the optimization process is a search for the functional dependencies. In a recursive search on the structure of the process expression of each entity type, the algorithm stores the *candidate* functional dependencies: a function from $\mathcal{P}(\mathcal{V}ar)$ to $\mathcal{V}ar$, where $\mathcal{V}ar$ is the set of all variables used in the entity types and $\mathcal{P}(\mathcal{V}ar)$ is the set of all subsets of $\mathcal{V}ar$. We say candidates, because a functional dependency will be selected only when it is required to optimize an interleave.

Algorithm in Figure 4 computes the functional dependencies for a process expression E. The function $FD(E, \textsc{ks}, wait?, \textsc{ep})$ is called initially with $\textsc{ks} = \emptyset$, $\textsc{ep} = \emptyset$ and $wait? = false$, since the parameter $\textsc{ks}$ represents the set of keys that will be mapped to the dependent variable and $\textsc{ep}$ represents the process definitions already parsed. The variable $wait?$ is used to avoid the creation of the next dependency.

```
FD(E, KS, wait?, EP) ≜ match E with
        Γ E0 -> return FD(E0, KS, wait?, EP),
           λ -> return ∅,
         a() -> return ∅,
 a(y1, ..., yn) -> return ∅,
     E1 . E2 -> return FD(E1, KS, wait?, EP) ∪ FD(E2, KS, wait?, EP),
     E1 | E2 -> return FD(E1, KS, wait?, EP) ∪ FD(E2, KS, wait?, EP),
    E1 || E2 -> return FD(E1, KS, wait?, EP) ∪ FD(E2, KS, wait?, EP),
   E1 ||| E2 -> return FD(E1, KS, true, EP) ∪ FD(E2, KS, true, EP),
  E1 |[Δ]| E2 -> return FD(E1, KS, true, EP) ∪ FD(E2, KS, true, EP),
     φ ⟹ E0 -> return FD(E0, KS, wait?, EP),
         E0* -> return FD(E0, KS, wait?, EP),
  | x ∈ S : E0 -> if wait? then add := ∅ else add := {(KS, x)} endif;
                return add ∪ FD(E0, KS ∪ {x}, false, EP)
  Φ x ∈ S : E0 -> where Φ  either a quantification ||| or |[]| on x
                   then KS' := KS ∪ {x} else KS' := KS endif;
                return FD(E0, KS', false, EP),
  Q(y1, ..., yn) -> if Q ∈ EP then return ∅ else
                   let E0 be the definition of Q(z1, ..., zn) ;
                   return FD(E0, KS, wait?, EP ∪ {Q})
                endif.
```

Fig. 4. An algorithm that computes functional dependencies

For example, if we analyze the process expression

$$\begin{array}{l} \quad |||\, x \in \mathrm{X} \,:\, \big((\,|\, y \in \mathrm{Y} \,:\, E'\,) \,|||\, (\,|\, y \in \mathrm{Y} \,:\, E'\,) \big) \\ \| \\ \quad |||\, y \in \mathrm{Y} \,:\, |||\, x \in \mathrm{X} \,:\, E' \end{array} \tag{2}$$

we don't know whether it associates one x to one or two y, because of the combination of two choice quantifications of $|\ y$ with $|||$. Since these cases can associate more than one value of y to a value of x, the existence of a functional dependency is not guaranteed. Even after excluding these cases, we can still successfully identify the functional dependencies for the process expressions that fit the patterns in [3]. The function call $FD(E,\ \mathrm{KS},\ wait?,\ \mathrm{EP})$ returns a map that is associated to E.

$$(\cdots E \cdots) \,\|\, (\cdots E' \cdots) \quad \text{where} \quad E = |||x \cdots |||\, \overrightarrow{y} \cdots \mathbf{A}\,(\,\overrightarrow{y}, x\,) \cdots$$
$$\text{and} \quad E' = |||\,\overrightarrow{y} \cdots |\ x \cdots \mathbf{A}\,(\,\overrightarrow{y}, x\,) \cdots \ .$$

Fig. 5. Example of structure for the κ-optimization

The Complete κ-optimization of an Action. Consider the example of a general process expression structure illustrated in Figure 5. The process definition **A** can be a simple pattern (Figure 6), or there could also be several producers and consumers with an arbitrary combination of $\mathsf{b}_i(\overrightarrow{y})$, which we denote by $\Xi_i \mathsf{b}_i(\overrightarrow{y})$ (Figure 7).

$$\mathbf{A}(\overrightarrow{y} : \overrightarrow{T}, x : T') = (\mathsf{a}(\overrightarrow{y}, x) \,.\, \mathsf{b}(\overrightarrow{y})^* \,.\, \mathsf{c}(\overrightarrow{y}))^*$$

Fig. 6. First pattern for the κ-optimization

The execution of an action $\mathsf{b}(\overrightarrow{y})$ for a quantifier $|||\, x$ in the entity E can be optimized under the following conditions:

1. Entity E is synchronized with an entity E' over a binding process expression **A** such that there is a functional dependency from $\overrightarrow{y}$ to x in E'. A binding expression is a process expression under the quantified choice scope that defined the functional dependency.
2. The binding process expression **A** enforces the following ordering constraints:
   (a) An event $\mathsf{b}(\overrightarrow{y})$ can only occur between a producer $\mathsf{a}(\overrightarrow{y}, x)$ and a consumer $\mathsf{c}(\overrightarrow{y})$
   (b) A consumer $\mathsf{c}(\overrightarrow{y})$ must be preceded by one producer $\mathsf{a}(\overrightarrow{y}, _)$ (i.e., the producer occurs before the consumer).
   (c) A consumer $\mathsf{c}(\overrightarrow{y})$ must occur between each pair of producers $\mathsf{a}(\overrightarrow{y}, _)$.
   (d) A producer $\mathsf{a}(\overrightarrow{y}, _)$ must occur between each pair of consumers $\mathsf{c}(\overrightarrow{y})$.
   Hence, a trace of $\mathbf{A}(\overrightarrow{y}, x)$ is of the form

$$\begin{array}{l} \mathsf{a}(\overrightarrow{y}, x) \,.\, \cdots \,.\, \mathsf{b}(\overrightarrow{y}) \,.\, \cdots \,. \\ \mathsf{c}(\overrightarrow{y}) \,.\, \mathsf{a}(\overrightarrow{y}, x') \,.\, \cdots \,. \\ \mathsf{b}(\overrightarrow{y}) \,.\, \cdots \,.\, \mathsf{c}(\overrightarrow{y}) \,.\, \cdots \end{array}$$

A consumer c can also be optimized under these conditions. Condition 1 is satisfied by the general structure of the expression in Figure 5. The patterns for **A** in Figures 6 and 7 satisfy condition 2 above.

$$\begin{array}{l} \mathbf{A}(\overrightarrow{y} : \overrightarrow{T}, x : T') = \\ \quad (\\ \qquad (\mathsf{a}_1(\overrightarrow{y}, x) \mid \cdots \mid \mathsf{a}_n(\overrightarrow{y}, x)) \,. \\ \qquad (\Xi_i \mathsf{b}_i(\overrightarrow{y}))^* \,. \\ \qquad (\mathsf{c}_1(\overrightarrow{y}) \mid \cdots \mid \mathsf{c}_m(\overrightarrow{y})) \\ \quad)^* \end{array}$$

Fig. 7. Second pattern for the κ-optimization

When a producer $\mathsf{a}(\overrightarrow{y}, x)$ is executed, the pair $\overrightarrow{y} \mapsto x$ is stored in a map f which represents the functional dependency $\overrightarrow{y} \rightarrow x$. When action $\mathsf{b}(\overrightarrow{y})$ must be executed, the only value of x in $|||x$ of E (i.e., quantification to optimize) that can execute $\mathsf{b}(\overrightarrow{y})$ is $f(\overrightarrow{y})$. This can be proved by contradiction. Suppose there are two values of x, v_1 and v_2 that can execute $\mathsf{b}(\overrightarrow{w})$. By condition 2a, each execution of $\mathsf{b}(\overrightarrow{y})$ is preceded by a producer in $\mathbf{A}$, which means that $|||x$ of E has spawned two interleaved processes, one for v_1 and one for v_2; each one has executed a producer, $\mathsf{a}(\overrightarrow{w}, v_1)$ and $\mathsf{a}(\overrightarrow{w}, v_2)$, respectively, and no consumer yet, by condition 2a. But since E and E' are synchronized over $\mathbf{A}$ by condition 1, the $|||\overrightarrow{y}$ of E' has spawned a single process for $\overrightarrow{w}$, and this process has executed two producers, $\mathsf{a}(\overrightarrow{w}, v_1)$ and $\mathsf{a}(\overrightarrow{w}, v_2)$, without a consumer in between, which contradicts condition 2c above.

The only candidate to execute a consumer $\mathsf{c}(\overrightarrow{w})$ is also the spawned process for $x = f(\overrightarrow{w})$ in $|||x$ of E. This can be proved as follows. Suppose there are two values of x, v_1 and v_2 that can execute $\mathsf{c}(\overrightarrow{w})$. By condition 2b, $|||x : T$ of E has spawned two interleaved processes, one for v_1 and one for v_2; each one has executed a producer, $\mathsf{a}(\overrightarrow{w}, v_1)$ and $\mathsf{a}(\overrightarrow{w}, v_2)$, respectively. Consider the last occurrences of these two actions. Since E and E' are synchronized over $\mathbf{A}$ by condition 1, the $|||\overrightarrow{y}$ of E' has spawned a single process for $\overrightarrow{w}$, and this process has now executed two producers, $\mathsf{a}(\overrightarrow{w}, v_1)$ and $\mathsf{a}(\overrightarrow{w}, v_2)$. By condition 2c, exactly one consumer $\mathsf{c}(\overrightarrow{w})$ must have been executed in between. Hence, only the last producer can execute $\mathsf{c}(\overrightarrow{w})$.

3.3 Generality of κ-optimization

Complete κ-optimization is not effective for all specifications that can be written. Actually, it is not effective for all IS specifications. However, our aim is to optimize all specifications written with the patterns described in [3]. There are seven patterns:

1. the producer-modifier-consumer pattern;
2. the one-to-many association pattern;
3. the multiple association pattern;
4. the n-ary association pattern;
5. the weak entity type pattern;
6. the recursive association pattern;
7. the inheritance association pattern.

For the sake of conciseness, the complete description of these patterns is omitted. It is straightforward to check that the first four patterns satisfy the conditions for κ-optimization. The first pattern describes the structure of an entity type. The producers of the association in the second pattern are exactly the producers of the functional dependency needed. These producers contain all the key variables, and the dependent variables, since they build an instance of the association. Each association also has consumer actions. The second pattern is same as the one used to illustrate indirect κ-optimization (i.e., a one-to-many association). The third and fourth patterns can use either direct or indirect κ-optimization,

depending on the cardinality of the association. An important point is that when an entity participates in several associations, these associations are combined with a parallel operator ($\|$). Therefore the problem of the process expression 2 (page 337) does not occur. This point justifies the use of the *wait*? predicate in the algorithm of Figure 4: if these patterns are used, there is no loss of generality. The last three patterns also fit our conditions for κ-optimization. They are special cases of the first four: a weak entity is just an instance of a simple association for our purpose; a recursive association is just an association between the entities of the same entity type; an inheritance association is decomposed into many process definitions, but it still has a behavior similar to that of simple entity types.

4 Implementation and Performance

EB^{3}PAI is implemented with Java 1.4 and an OODB ObjectStore PSE Pro for Java 6.0. The parser was built with ANTLR 2.7.1. Indirect κ-optimization is not implemented, but direct κ-optimization is. The performance for indirect κ-optimization should be very close to that of direct κ-optimization, because it uses the same data structures plus an additional hash table to store the functional dependencies.

4.1 Complexity Analysis

Let E_i, $1 \leq i \leq m_E$, denote an entity type and A_j, $1 \leq j \leq m_A$, denote an association. An association links two or more entity type E_i. The size $|E_i|$ of an entity types E_i is the maximal number of entities in the entity type. The size $|A_j|$ of an association A_j is a product of all $|E_k|$, where E_k is an entity type involved in A_j. Let n denote the sum of all $|X|$, where X is either an entity type or an association of an EB^3 specification. Let s denote the number of nodes in the tree representing a process expression, excluding the nodes of a κ-optimized quantification (they will be computed with n). Note that for most ISs, s is usually smaller, whereas n can be quite larger. Therefore, the number of nodes s can be considered negligible with respect to the number of entities and associations n. The search for a proof using the inference rules requires inspection (in the worst case) of all the nodes (i.e., s nodes). The space complexity is $\mathcal{O}(s + n)$, which corresponds to the size of a process expression, including instances of quantified process expressions. But, since s is negligible with respect to n, the space complexity is $\mathcal{O}(n)$. Without κ-optimization, the algorithmic complexity is impractical since the number of nodes s is multiplied by the number n of entities and associations involved. So a transition computation has a complexity of $\mathcal{O}(s\, n)$. With the κ-optimization, only one node is inspected for quantified expressions. The execution of a κ-optimized quantification depends on the implementation chosen for a map K. ObjectStore offers hash tables, which yield constant time in an average case, or B-trees, which yield logarithmic time. Hence, the algorithmic complexity of a transition computation is $\mathcal{O}(s + log(n))$

on average. The space complexity is still $\mathcal{O}(n)$. For indirect κ-optimization, we also need to store the functional dependencies (*cf.* 3.2). The total size of the tables needed for these optimizations is bounded by the number of quantifications involved multiplied by the number of actions involved. Theorically, this number could be an overwhelming difficulty to tackle. However, practicably, it is still negligible with respect to n. Therefore, the space complexity is the same as that of direct κ-optimization. The algorithmic complexity is also the same because the small tables needed can be implemented with hash tables which yield constant time. Typically, the algorithmic complexity of a manual implementation of an IS specification is $\mathcal{O}(log(n))$, since it will access several records from the database, each access usually being backed by an index which yields $log(n)$ access time using B-trees; its space complexity is $\mathcal{O}(n)$ on average. All of these complexities are summarized in Figure 8, under the hypothesis of IS domain. Thus, for κ-optimizable specifications, EB^{3}PAI has an overhead of $\mathcal{O}(s)$

	Algorithmic complexity	Space complexity
EB^{3}PAI with no optimization	$\mathcal{O}(s.n)$	$\mathcal{O}(n)$
EB^{3}PAI with direct κ-optimization	$\mathcal{O}(log(n) + s)$	$\mathcal{O}(n)$
EB^{3}PAI with indirect κ-optimization	$\mathcal{O}(log(n) + s)$	$\mathcal{O}(n)$
manual implementation	$\mathcal{O}(log(n))$	$\mathcal{O}(n)$

Fig. 8. Algorithmic and space complexity of EB^{3}PAI for IS specifications

compared with manual implementation of an IS. With no κ-optimization, the difference is substantial and EB^{3}PAI becomes impractical as a tool, but it can still be useful for specification animation for validation purposes.

4.2 Performance for Direct κ-optimization

Performance tests were conducted with a specification of a library management system on an Intel Core Duo 1.66GHz with 1GB of DDR2 SDRAM, running Mac OSX 10.4. Indirect κ-optimization has not been implemented yet; only direct κ-optimization. Figure 9 provides some statistics on these experiments. The column titled "Without DB" corresponds to the execution time without the use of a database; the column titled "With DB" corresponds to the time of executions with the use of an Object Store PSEPro as database. The experiment consists of the execution of actions creating 9,000 books and 9,000 members, followed by the execution of 30,000 actions which were randomly generated; 9,899 of these actions were valid and 20,101 were invalid. Figure 9 only shows information for valid actions. Invalid actions (actions which must be rejected by the interpreter) are less expensive in time than valid actions: they require approximately half the time of a valid action. We also manually implemented the

	Without DB	With DB
Time	1 m 52 s	8 m 27 s
Mean	4 ms	81 ms
Median	1 ms	10 ms

Fig. 9. EB^{3}PAI execution times for the library system

library specification in Java using an Oracle database. The average transaction processing time is 10 ms, which is 8 times faster than EB^{3}PAI with a database. Nevertheless, 80 ms is still acceptable for many IS systems where the transaction rate is low (e.g., a library management system). The results are good, but the median is quite low in comparison to the mean time. This is because some executions are rather slow (more than 2 s). These executions occur periodically. We are currently investigating the reason for these anomalies in order to correct this behavior. We also intend to implement indirect κ-optimization and validate its real performance.

5 Conclusion

In this paper, we have presented two optimization techniques to efficiently execute quantified process expressions in the EB^3 process algebra. Their space and algorithmic complexities are comparable to those of a manual implementation for a large number of IS specifications which are determined by a set of classical specification patterns. Direct κ-optimization was implemented in the EB^{3}PAI interpreter. It performs 8 times slower than a manual implementation of the specification for the library system, but its average response time is acceptable for a large class of IS with low transaction rates, which demonstrates that symbolic execution is a viable way of implementing IS.

The performance of EB^{3}PAI is largely dependent on the OO database used to the store the object representation of the specification. It seems quite feasible to implement a dedicated persistence manager for EB^{3}PAI to reduce the number of disk IOs.

We are currently looking at other optimizations for EB^{3}PAI. Tail-recursive deterministic process expressions can be represented by an extended labelled-transition system (ELTS) [23], which basically takes less space and avoids the computation of a proof at each transition. Preliminary experimentation has shown us that ELTS coupled with κ-optimization could cut computation time by as much as 40%. We are working on a complete definition of ELTS and the algorithms to translate an EB^3 process expression into an ELTS.

Future work also includes techniques for issuing meaningful error messages when an action is not executed. For instance, if a $\mathsf{Lend}(bId, mId)$ is rejected by the interpreter, we must tell the user why; it could be that the book or the member does not exist, the book is on loan to another member, or the member has reached his loan limit. This problem is similar to the determination of error

messages by a compiler. Finally, we wish to investigate how parallelism could be used for symbolic execution.

References

1. Frappier, M., Fraikin, B., Laleau, R., Richard, M.: Automatic Production of Information Systems. In: AAAI Symposium on Logic-Based Program Synthesis, p. 7. AAAI, Stanford University, Stanford, CA (2002)
2. Fraikin, B., Gervais, F., Frappier, M., Laleau, R., Richard, M.: Synthesizing Information Systems: the APIS Project. In: Rolland, C., Pastor, O., Cavarero, J.L. (eds.) First International Conference on Research Challenges in Information Science (RCIS), Ouarzazate, Morocco, p. 12 (2007)
3. Frappier, M., St-Denis, R.: EB^3: an Entity-Based Black-Box Specification Method for Information Systems. Software and Systems Modeling 2, 134–149 (2003)
4. Formal Systems (Europe) Ltd.: Process Behaviour Explorer (ProBE) User Manual (2003)
5. Leuschel, M.: Design and Implementation of the High-Level Specification Language CSP(LP) in Prolog. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 14–28. Springer, Heidelberg (2001)
6. Wooters, A.G.: Manual for the CRL tool set (version 2.8.2). Report SEN-R0130, CWI, Amsterdam, the Netherlands (2001)
7. Garavel, H., Lang, F., Mateescu, R.: An Overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter 4, 13–24 (2002)
8. Freitas, A., Cavalcanti, A.: Automatic Translation from *Circus* to Java. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 115–130. Springer, Heidelberg (2006)
9. Freitas, A.: From Circus to Java: Implementation and Verification of a Translation Strategy. Master's thesis, Department of Computer Science, The University of York (2005)
10. Oliveira, M.: Formal Derivation of State-Rich Reactive Programs using Circus. PhD thesis, Department of Computer Science - University of York, UK (2005)
11. Welch, P.H.: Process oriented design for java: Concurrency for all. In: Priss, U., Corbett, D.R., Angelova, G. (eds.) ICCS 2002. LNCS (LNAI), vol. 2393, Springer, Heidelberg (2002)
12. Formal Systems (Europe) Ltd.: Failures-Divergence Refinement (FDR2) User Manual (2005)
13. Moller, F., Stevens, P.: (Edinburgh Concurrency Workbench user manual (version 7.1))
14. Butler, M., Leuschel, M.: Combining CSP and B for Specification and Property Verification. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
15. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs. Wiley, Chichester (2006)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs, NJ (1985)
17. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
18. Bergstra, J.A., Klop, J.W.: Process Algebra for Synchronous Communication. Information and Control 60, 109–137 (1984)

19. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14, 25–59 (1987)
20. Fraikin, B.: Interprétation efficace d'expression de processus EB^3. PhD thesis, Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada (2006)
21. Fraikin, B., Frappier, M.: Efficient Execution of Process Expressions Using Symbolic Interpretation. Technical Report 8, Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada (2005)
22. Fraikin, B., Frappier, M., Laleau, R.: State-Based versus Event-Based Specifications for Information System Specification: a comparison of B and EB^3. Software and System Modeling 4, 236–257 (2005)
23. Frappier, M., Laleau, R.: Verifying Event Ordering Properties for Information Systems. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 421–436. Springer, Heidelberg (2003)