Synthesizing Information Systems: the APIS Project

Marc Frappier*, Benoît Fraikin*, Frédéric Gervais*†, Régine Laleau‡ and Mario Richard*§
*GRIL, Département d'Informatique, Université de Sherbrooke, Canada
Email: {benoit.fraikin, marc.frappier, frederic.gervais}@usherbrooke.ca
†CEDRIC, CNAM-IIE, France
‡LACL, Université Paris 12, Département Informatique, IUT Fontainebleau, France
Email: laleau@univ-paris12.fr
§IBM Canada, Business Consulting Services, Montreal, Québec, Canada

Abstract— **This article presents the main features of the APIS project that addresses the rapid development of information systems from formal specifications. Information systems are specified using EB^3, a trace-based formal language. The sequences of input events accepted by the system are described with a process algebra; they represent the valid traces of the information system. Entity types, associations and attributes are described using a class diagram and computed by means of recursive functions defined on the valid traces of the system. In the APIS framework, three tools have been developed. A first tool, called DCI-WEB, allows the generation of Web interfaces from GUI specifications. To query and/or to update the system, an end-user can trigger an event through the Web interface. This event is then analyzed by EB^{3}PAI, an interpreter for EB^3 process expressions. Finally, the tool EB^{3}TG generates, for each EB^3 action, a Java program that executes a relational database transaction. The synthesized transactions implement the specification of the information system's data structure and are used by the interpreter to update or query the database. The article also brings out the main future developments of the project.**

Keywords: **Information system modelling, formal notation, information system synthesis.**

I. Introduction

This article addresses the rapid development of information systems (IS) from specifications, a process which we call *information system synthesis*. The traditional process for developing an IS consists of the following steps: requirements analysis, specification, design, implementation, test and deployment. Following its initial deployment, an IS is subject to constant evolution, which consists in adapting the IS to respond to changes in its environments. Evolution is essentially iteration over the development steps.

The *main challenges* of this research are to reduce the effort required to develop an IS, to shorten the cycle time for development and evolution, and to provide reliable IS. This can be reached by using appropriate notations for specifying IS and by using algorithms for automatically generating an implementation from the specification. In turn, this reduces or eliminates the need for the design and implementation steps.

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 IS can be decomposed into three parts : the user interface, the business logic and the database. The database is definitely the most mastered part. Relational and object-oriented databases provide suitable services for IS development. From an abstraction power view-point, an equivalent technology does not exist for user interfaces and business logic. The notations that we intend to develop originate from the field of formal methods in software engineering. Formal methods were traditionally intended for safety critical software, in order to ensure correctness in a rigorous way. Traditional formal methods involve proofs of correctness, a task which requires a high-level expertise, and most often a manual derivation of an implementation. In contrast, we consider automatic derivation from the formal specifications. Not all software is amenable to automatic derivation, but a large number of elements of an IS are. This is the cornerstone of this article and its most important scientific and technological contribution.

The structure of the article is as follows: in Section 2, we summarize the current research and technology for developing IS. In our project, information systems are specified using EB^3, a trace-based formal language, presented in Section 3. The sequences of input events accepted by the system are described with a process algebra; they represent the valid traces of the IS. Entity types, associations and attributes are described using a class diagram and computed by means of recursive functions defined on the valid traces of the system. Section 4 described the three tools developed in the APIS project: DCI-WEB allows the generation of Web interfaces from GUI specifications, EB^{3}PAI is an interpreter for EB^3 process expressions and EB^{3}TG generates, for each EB^3 action, a Java program that executes a relational database transaction. The synthesized transactions implement the specification of the IS data structure. Section 5 considers possible future developments and Section 6 concludes by comparing our project with the approaches presented in Section 2.

II. State of the Art in IS

A. COTS

COTS (commercial off-the-shelf) IS have seized most of the IS market for standard business functions typically covered by an ERP (Enterprise Resource Planning) system. The COTS ERP global market was worth US$ 16,7 billion in 2005. It is dominated by five major suppliers which control 72 % of the market: SAP, Oracle, Sage Group, Microsoft's Business Solutions group, and SSA Global. Organizations moved in the 90's from custom built IS to COTS IS in order to reduce IS costs, to increase systems integration, and to focus their energy on core business functions. However, there are several drawbacks in using COTS. An organization must adapt its business process to fit the vanilla flavor proposed by the COTS IS; adaptation and customization of ERP are strongly discouraged, because they are expensive to maintain over new releases. For some business functions, this is in contradiction with the principles of agility and adaptability, since innovative business processes or services are often considered a key competitive advantage in a global market. Moreover, organizations are locked in with a single supplier, since switching from one ERP supplier to another is such a colossal task that few can seriously consider it. Customers must pay annual maintenance fees, on which they have little control, and must follow the software evolutions proposed by the supplier, which does not always match the evolution path of their needs.

B. Custom Development

UML is probably the most widely used notation for analyzing, specifying and designing IS in industry. UML has a well-defined syntax, but there is no consensus on a comprehensive formal semantics. Thus, UML is not an executable notation. Executable code can be generated from some of the diagrams, but it does not constitute a complete implementation.

The technology for implementing IS is very rich and evolves rapidly. It nicely supports distributed implementations on heterogeneous computing and programming platforms, organized into layers which can communicate through de facto standards like SOAP (Simple Object Access Protocol), XML, WSDL (Web Services Description Language), JDBC. The recent trend is to organize IS into services accessible from the Web, and to compose them to form new ones.

Industrial case tools supporting IS development are of clerical nature. They allow designers to *edit* user requirement models, data models, objects models, graphical user interface layout, and perform basic translation of these models into executable code. But the bulk of the design, programming and testing is manually done by humans. These three activities consume up to 70 % of the development effort. They are usually not hard to realize, but they are time consuming and error-prone.

The maintenance of IS remains as challenging as it was in the past. The main factors are still lack of proper documentation, personnel turnover, learning curve and the huge amount of technical details to deal with. For example, if someone has to make a small change in a typical web IS, he has to master the following aspects: the business logic of the system, a programming language like Java or C#, HTML, a web scripting language like Javascript, Struts, XML, SOAP, JDBC, a web application server like Tomcat or IBM Websphere, servlets, and a relational database like Postgres or Oracle. Properly mastering each of these technologies requires something between a few weeks and several months.

C. Service Oriented Architecture

COTS IS are usually decomposed into modules which can be bought separately. Still, these modules are tightly coupled and integrated; it remains very difficult to combine modules from different vendors. A recent trend in the industry is to decompose modules further, into smaller independent units called *services*. In a service oriented architecture (SOA), services can be called over a network using protocols like SOAP or, more abstractly, using an enterprise service bus. The interface of a service is described using a language like WSDL; there is currently no industrial standard to describe the functional behavior of a service. New services can be constructed from other services using languages like BPEL (Business Process Execution Language) [2] and WSCI (Web Service Choreography Interface).

D. Model Driven Architecture and Generative Programming

Model driven architecture (MDA) [23] is an initiative of the Object Management Group. It consists in building abstract models, typically using UML, and transforming them into executable systems. MDA fosters portability, interoperability and reusability through architectural separation of concerns. AndroMDA [1] is a typical example of MDA for IS. MDA is closely related to generative programming (GP), which aims at automating the creation of a system from a specification written in one or more textual or graphical domain-specific languages [7]. Domain specific languages can take various forms and are not restricted to UML, although UML is the most used. Pastor *et al* [24] are using notations very close to UML to model IS and generate code from them. They use class diagrams to describe the entities of the system, state machines to define the behavior of each object of a given class, and a collaboration diagram to show how objects cooperate. Updates to attributes are given by a simple functional description. The semantics of these specifications is described using the OASIS framework, also proposed by Pastor *et al*. OASIS uses a process algebra to translate state machines. MDA and generative programming are active research area, but they have not reached yet the IS industry.

E. Graphical User Interface Synthesis

Closely related to MDA, model-based interface development environments (MB-IDE) [25] address the issue of generating a graphical user interface (GUI) from abstract models. These models describe various aspects like the application functions, data, window contents, dialogue (interactions between the users and the systems), user tasks and platform. The presentation model describes which association and filtering conditions to use for an entity, and the graphical representation of entities. The dialogue model describes the system reaction to a user action. The user model defines the access privilege and preferences of a user (e.g., limited access to some entity types, attributes or entities, special-purpose representations of some data).

A knowledge base, algorithms and mapping rules, sometimes supplemented with human guidance, can generate a concrete GUI from the abstract models. Early approaches were able to generate a standard (single style) interface from a data model. More recent approaches have focused on providing more flexibility for specifying a wider ranger of GUI styles [28] to satisfy various user requirements. The main challenges facing MB-IDE are i) the composition of the various models in order to obtain a coherent GUI, and ii) management of model complexity in order to achieve greater flexibility in GUI style. Model complexity significantly increases when more flexibility is provided. MB-IDE tools are not yet available as commercial tools. Recently, there were attempts to define XML-based specifications integrating GUI abstract and concrete models and their relationships [20], [26].

F. Formal Methods for IS

Formal methods denote a variety of notations, techniques and tools which are based on *mathematical models* of software. Notations have a formal syntax and a formal semantics, which avoid the ambiguities related to semi-formal notations like UML. They are more abstract than programming languages and cover the phases from specification to implementation. Formal methods include a variety techniques to check that an implementation satisfies a specification (or a set of properties) through either a formal proof derived with a theorem prover or by model checking, which consists in exploring the state space of the system. They are increasingly used in medium to large scale industrial applications in transportation, telecommunication, aerospace and VLSI (e.g., Alstom, Clearsy, Gemplus, Schlumberger in France, BMW, Siemens in Germany, Praxis in the UK, Intel, Lucent, NASA, NSA in the USA).

Few researchers have addressed IS using formal methods. The work of Laleau *et al* is the closest to our work [21], [22]. They synthesize information systems using the B method. A UML specification is translated into a B specification, using translation rules that provide a formal semantics to a subset of UML. The specification can then automatically be refined to produce an executable system, which is semi-automatically proven correct with respect to the specification. This approach addresses the business logic and the database; the user interface is not taken into account.

Butler *et al* have studied the specification and implementation of long business transactions and compensation operators using B and StAC, a process algebra with compensation operators [4]. Compensation consists in specifying actions that cancel the effect of previous actions. Refinement into an implementation is done manually using the B method. StAC can also be used to provide a formal semantics to BPEL, which also includes compensation mechanisms.

III. SPECIFYING INFORMATION SYSTEMS WITH THE EB^3 METHOD

Over the past years, our research team has developed a specification method for IS, called EB^3 (Entity-Based Black Box), and a framework that supports the synthesis of an executable IS from an EB^3 specification, called APIS (Automatic Production of Information Systems), which is described in Section IV.

A. Overview of EB^3

The core of EB^3 [12] includes a process and a formal notation to describe a complete and precise specification of the input-output behaviour of an IS. An EB^3 specification is composed of five parts:

1) A diagram describes the entity types and associations of the IS, and their respective actions and attributes. It is based on entity-relationship (ER) model concepts [8] and uses a subset of the UML graphical notation for class diagrams. The terms *entity type* and *entity* are used instead of class and object, respectively. This diagram is called ER diagram in the remainder of the paper.
2) A process expression, denoted by *main*, defines the valid input traces of the system.
3) Input-output (I/O) rules assign an output to each valid input trace of the system.
4) Recursive functions, defined on the valid input traces of *main*, assign values to entity type and associations attributes.
5) A graphical user interface (GUI) specification describes the functionalities of Web interfaces used to interact with IS end-users.

EB^3 differs from the other process algebraic languages by the following characteristics. First of all, EB^3 has been specially created for IS specification. Hence, the specification of the inputs and of the outputs of the system is divided in two parts. The semantics of EB^3 is a trace-based semantics. Process expressions represent the valid input traces of the IS, while outputs are computed from valid EB^3 traces. The syntax of process expressions has been simplified and adapted to IS with respect to other process algebra languages like CSP [19]. In particular, EB^3 process expressions are close to regular expressions, with the use of the sequence operator and the Kleene closure (see Sect. III-B.2).

The denotational semantics of an EB^3 specification is given by a relation R defined on $\mathcal{T}(main) \times O$, where $\mathcal{T}(main)$

```
$\mathtt{trace} := [\,]$;
forever do
  receive input event $\sigma$;
  if $main$ can accept $\mathtt{trace} :: \sigma$ then
    $\mathtt{trace} := \mathtt{trace} :: \sigma$;
    send output event $o$ such that
    $(\mathtt{trace}, o) \in R$;
  else
    send error message;
```

Fig. 1. Operational semantics of EB^3

denotes the traces accepted by *main* and O is the set of output events. Let `trace` denote the system trace, which is a list of valid input events accepted so far in the execution of the system. Let $t :: \sigma$ denote the right append of an input event σ to trace t, and let `[ ]` denote the empty trace. The operational behaviour of an EB^3 specification is defined in Fig. 1.

B. Case Study

The case study is a library management system, which has to manage book loans by members. A book is acquired (action Acquire) by the library; it can be discarded (Discard), but only if it is not borrowed. Action Modify is used to change the title of the book, while action DisplayTitle outputs the title of the book. A member must join the library (Register) in order to borrow a book (Lend). A member can relinquish library membership (Unregister) only when all his loans are returned (Return). A book can be borrowed by only one member at once.

The main parts of an EB^3 specification are illustrated in the next paragraphs. Nevertheless, we do not deal with the GUI specification here, since this work [27] is described in Section IV-B.

1) ER Diagram: Figure 2 shows the ER diagram of the example.

The signature of EB^3 actions is the following:

$\mathsf{Acquire}(bId : bk\_Set, bTitle : T^{\perp}) : void$
$\mathsf{Discard}(bId : bk\_Set) : void$
$\mathsf{Modify}(bId : bk\_Set, nTitle : T^{\perp}) : void$
$\mathsf{DisplayTitle}(bId : bk\_Set) : (title : T^{\perp})$
$\mathsf{Register}(mId : mk\_Set) : void$
$\mathsf{Unregister}(mId : mk\_Set) : void$
$\mathsf{Lend}(bId : bk\_Set, mId : mk\_Set,$
$\quad type : \{Permanent, Classic\}) : void$
$\mathsf{Return}(bId : bk\_Set) : void$

The special type *void* is used to denote an action with no input-output rule. Some input parameters can be instantiated with a default value, *NULL*, that denotes undefinedness. The input type is then decorated with an exponent "$\perp$". In EB^3, each action has an implicit output which determines whether an event of this action is valid ("*ok*") or not ("*error*"). An event is considered as valid if it is accepted by the main process.

2) Process Expressions: An input event σ is an instantiation of an action a and of its input parameters. An instantiated action $\mathsf{a}(t_1, ..., t_n)$ constitutes an elementary process expression. The special symbol "_" may be used as an actual parameter of an action, to denote an arbitrary value of the corresponding type.

The EB^3 notation for process expressions is similar to Hoare's CSP [19]. Complex EB^3 process expressions can be constructed from elementary process expressions using the following operators: sequence (denoted by $\boldsymbol{.}$), choice ($|$), Kleene closure (*), interleaving ($|||$), parallel composition ($||$, i.e., CSP's synchronization on shared actions), guard ($\Longrightarrow$), process call, and quantification of choice ($|x : T : ...$) and of interleaving ($|||x : T : ...$). The complete syntax and semantics of EB^3 can be found in [12].

For instance, the EB^3 process expression for entity type *book* is of the following form:

$book(bId : bk\_Set) \triangleq$
$\mathsf{Acquire}(bId, \_)\boldsymbol{.}$
$($
$\qquad (\ |\ mId : mk\_Set : loan(mId, bId)\ )^*$
$\quad ||$
$\qquad \mathsf{Modify}(bId, \_)^*$
$\quad ||$
$\qquad \mathsf{DisplayTitle}(bId)^*$
$)\boldsymbol{.}$
$\mathsf{Discard}(bId)$

where $loan$ denotes the process expression for the homonymic association. The process *book* describes the life-cycle of each book entity of the system.

First, book entity bId is produced by action Acquire. Then, it can be borrowed by only one member entity mId at once (quantified choice " $|\ mId : mk\_Set : ...$"). Indeed, process expression *book* then calls process expression $loan$, that involves actions Lend and Return. The Kleene closure on $loan$ means that an arbitrary number of loans can be made on book entity bId. At any moment, actions Modify and DisplayTitle can be interleaved with the actions of $loan$. Finally, book entity bId is consumed by action Discard. The complete process expressions for the example are given in [15].

Contrary to other formal languages based on process algebras, the process expressions of an EB^3 specification focus on the valid input event traces of the system. This viewpoint simplifies the process specification, since the management of invalid events and the communication of results are not described in the processes, but in other parts of the EB^3 specification. For the implementation, the EB^3 method provides a tool which checks that each new event is valid before executing it.

3) Input-Output Rules: The system trace is usually accessed through recursive functions that extract relevant information from it. For instance, the following input-output rule is defined for action DisplayTitle:

RULE *R1*
INPUT $\mathsf{DisplayTitle}(bId)$

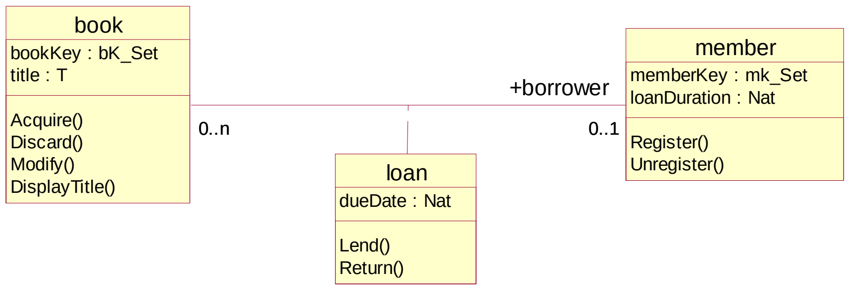


Fig. 2. ER diagram of the library management system

OUTPUT $title(trace, bId)$
END;

When action DisplayTitle is a valid input event, then the recursive function $title$ is called to compute the value of attribute $title$.

4) Recursive Functions: In EB^3, recursive functions defined on the valid traces are used to describe the data model of the IS. In this paper, we just provide an informal description of these functions. The syntax and semantics are detailed in [15].

The definition of an attribute in EB^3 is a recursive function on the valid traces. It outputs the attribute values that are valid for the state in which the system is, after having executed the input events in the trace. For instance, the next two attribute definitions are associated to entity type $book$:

```
$bookKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bk\_Set) \triangleq$
**match** $last(s)$ **with**
    $\perp$ : $\emptyset$,
    $\mathsf{Acquire}(bId)$ : $bookKey(front(s)) \cup \{bId\}$,
    $\mathsf{Discard}(bId)$ : $bookKey(front(s)) - \{bId\}$,
    $\_$ : $bookKey(front(s))$;

$dueDate(s : \mathcal{T}(\mathsf{main}), bId : bK\_Set) : DATE \triangleq$
**match** $last(s)$ **with**
    $\perp$ : $\perp$,
    $\mathsf{Lend}(bId, mId, type)$ : **if** $type = Permanent$
            **then** $CurrentDate + 365$
            **else** $CurrentDate + loanDuration(mId)$
            **end** ,
    $\mathsf{Return}(bId)$ : $\perp$,
    $\_$ : $dueDate(front(s), bId)$;
```

Function $bookKey$ represents the key of $book$; it has a unique input parameter, a valid trace s, and it outputs the set of key values of $book$. Function $dueDate$ corresponds to the homonymic non-key attribute. It also has a valid trace as input parameter. The function outputs the attribute value for the key given as input parameter.

The functions are total and defined in a CAML-like style (CAML is a functional language [6]). $\mathbb{F}(S)$ denotes the set of finite subsets of set S. Standard list operators are used, such as *last* and *front* which respectively return the last element and all but the last element of a list; they return the special value $\perp$ when the list is empty. The functions are always recursively called with $front(s)$. A recursive call refers to the current value of the attribute, i.e., its value before the update. Expressions of the form $input : expr$, like $\mathsf{Acquire}(bId) : bookKey(front(s)) \cup \{bId\}$ in $bookKey$, are called *input clauses*.

The functional style of attribute definitions points out the effects of actions on a particular attribute. For instance, the key of *book* is affected by actions Acquire and Discard. The latter has the effect of removing a book from the set of book entities that exist before the execution of the action, while the former consists in adding a new book. Special symbol '$\perp$' in an input clause pattern matches with the empty trace, while symbol '$\_$' is used to pattern match with any list element.

IV. The APIS Framework

In the APIS project, rather than using refinement techniques to implement the system like in state-based languages [21], the IS is obtained by efficient interpretation and code generation from the different components of an EB^3 specification.

Figure 3 represents the different components of the APIS framework. From the designer point of view, recall that an EB^3 specification of an IS consists of five elements (cf III-A) represented in the upper part of the figure. The IS execution environment is built on a process expression interpreter (EB^{3}PAI [11]). It checks that every new input event matches the behaviour of the EB^3 specification described by the process expression *main*. To interact with the interpreter, a end-user generates an event through a web interface. If the event is valid then the corresponding DB transaction is executed, otherwise, an error message is sent back to the user. The DCI-WEB tool allows us to generate web interfaces from GUI specifications. The tool EB^{3}TG generates, for each EB^3 action, a Java program that executes a relational database transaction. The synthesized transactions implement the specification of IS attributes and are used by the interpreter to update or query the database. The tool EB^{3}TG also generates Java programs that create and initialize the database. Several database management systems (DBMS), like Oracle, PostgreSQL and MySQL, are supported by EB^{3}TG.

A. *The process expression interpreter* EB^{3}PAI

EB^{3}PAI (EB^3 Process Algebra Interpreter), is the core of the APIS toolbox. It can efficiently execute all IS process

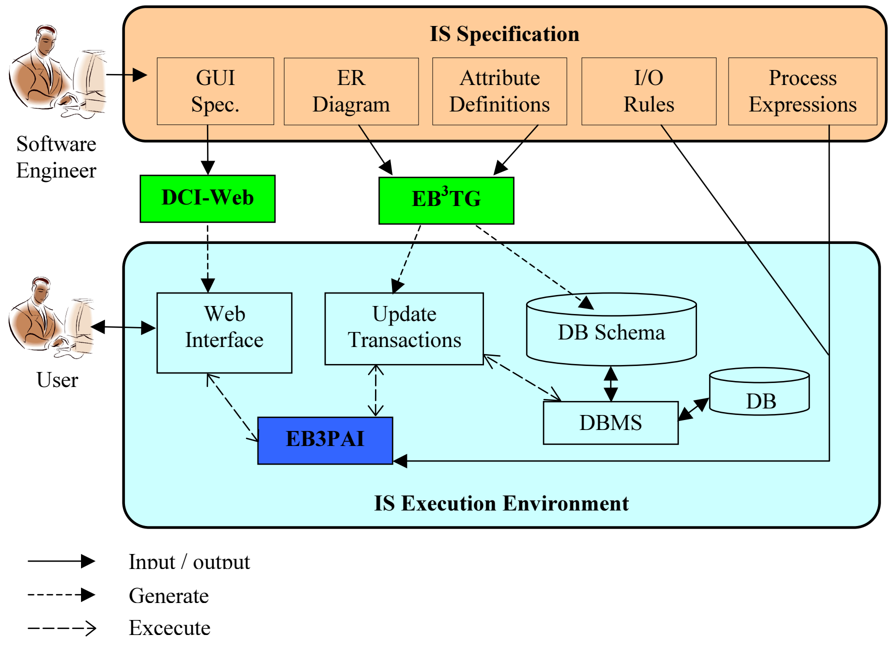


Fig. 3. Components of the APIS framework

$$\text{EB}^3{}_1\,\frac{\sigma \in \Sigma_e \cup \{\lambda\}}{\sigma \xrightarrow{\sigma} \boxdot} \qquad \text{EB}^3{}_2\,\frac{E_1 \xrightarrow{\sigma} E_1' \ \wedge\ E_1' \neq \boxdot}{E_1 \,.\, E_2 \xrightarrow{\sigma} E_1' \,.\, E_2}$$

$$\text{EB}^3{}_3\,\frac{E_1 \xrightarrow{\sigma} E_1'}{E_1 \mid E_2 \xrightarrow{\sigma} E_1'} \qquad \text{EB}^3{}_4\,\frac{E_1 \xrightarrow{\sigma} E_1' \ \wedge\ E_1' = \boxdot}{E_1 \,.\, E_2 \xrightarrow{\sigma} E_2}$$

Fig. 4. Examples of transition rules

expression patterns identified in [16] using the operational semantics of the EB^3 process algebra (see Fig. 1). Basically, this interpreter efficiently computes on-the-fly a proof of a transition $E \xrightarrow{\sigma} E'$ to determine if process expression E can accept user input σ. The transition $E \xrightarrow{\sigma} E'$ denotes that a process expression E can execute an action σ and be transformed into a process expression E'. Hence, we do not generate executable code to execute a process expression; rather, EB^{3}PAI is an abstract machine that executes a process expression. EB^{3}PAI is implemented in Java, for portability and simplicity.

Figure 4 shows four transition rules from the system of rules. Symbol $\boxdot$ denotes successful completion and Σ_e denotes the set of actions. For instance, rule EB$^3{}_1$ states that an action can execute itself and transform into $\boxdot$. Rule EB$^3{}_2$ states that a sequence $E_1.E_2$ can execute action σ if E_1 can execute σ. Moreover, if E_1' is the result of executing σ on E_1, then $E_1'.E_2$ is the result of executing σ on $E_1.E_2$. The symbol λ denotes an internal action that a process may execute without requiring input from the environment.

Given a process expression P and an action σ, one can compute the possible transitions and resulting process expressions using the set of transition 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 premises 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 premise. (e.g., rule EB$^3{}_1$). Therefore, the resulting process expression is incrementally constructed over the inference rules through termination of recursive search calls. For the sake of concision, the algorithm is omitted (see [11] for details). We only give an example. Let $P \triangleq \mathsf{a} \,.\, (\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d}$. According to the transition rules, the process P can execute a, then either b or c, then d. The sequence of transitions is as follows:

$$\begin{tikzcd} \mathsf{a.(b|c).d} \arrow[r, "\mathsf{a}"] & \mathsf{(b|c).d} \arrow[r, bend left, "\mathsf{b}"] \arrow[r, bend right, "\mathsf{c}"'] & \mathsf{d} \arrow[r, "\mathsf{d}"] & \boxdot \end{tikzcd}$$

For illustration purposes, we will present the proof of the transition $(\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d} \xrightarrow{\mathsf{b}} \mathsf{d}$, The following is a proof tree of this transition:

$$\text{EB}^3{}_4\,\cfrac{\text{EB}^3{}_3\,\cfrac{\text{EB}^3{}_1\,\cfrac{\mathsf{b} \in \Sigma_e \cup \{\lambda\}}{\mathsf{b} \xrightarrow{\mathsf{b}} \boxdot}}{(\mathsf{b} \mid \mathsf{c}) \xrightarrow{\mathsf{b}} \boxdot}}{(\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d} \xrightarrow{\mathsf{b}} \mathsf{d}}$$

The other transitions are proved in a similar manner. EB^{3}PAI executes a transition by computing its proof.

EB^{3}PAI solves three problems for the efficient execution of process expressions. First, it can automatically execute the internal action λ when necessary in order to accept a user input σ. Second, it can deal with nondeterministic process expressions and execute them like deterministic process expressions. This problem is similar to the transformation of a regular expression into a deterministic automaton, although it is solved in a different manner, since process expressions are more expressive than regular expressions. Third, it can execute quantified process expressions. For instance, the following process expression is executed in $O(\log n)$

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

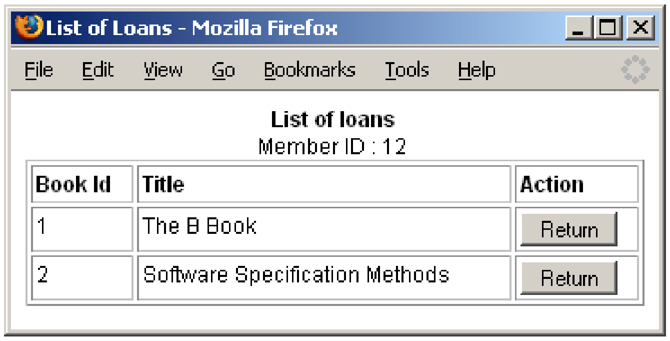
This is a significant improvement over existing process algebra interpreters. CADP [13] cannot execute quantified process expression; μCRL [18] supports only choice quantification and executes it in $O(n)$.

For IS patterns described in [12], the algorithmic complexity of EB^{3}PAI is $O(s+log(n))$ in time and $O(s+n)$ in space, where s is the size of the specification text and n is the number of entity instances [11]. This favorably compares with programmer-made implementations, which are in $O(log(n))$ in time and $O(n)$ in space; the overhead of EB^{3}PAI is $O(s)$. In terms of actual response time, EB^{3}PAI is definitely slower than a programmer-made implementation. For a library case study, on a Pentium III 800MHz with 384Mo of SDRAM, running GNU/Linux, EB^{3}PAI executes transactions with an average response time of 100 ms, whereas the programmer-made implementation executes transactions in 10 ms. 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. Note that for IS with low transactions rates, 100 ms is quite reasonable. The main overhead of EB^{3}PAI is the persistency manager, which is based on an OO database.

B. DCI-WEB

DCI-WEB is a user interface specification language supported by a tool that generates an implementation based on HTML, JSP and Struts. It can be connected to EB^{3}PAI or to any other system that can provide information for the dynamic content of a page; hence, DCI-WEB is independent of EB^3. The DCI-WEB tool generates from the specification the JSP pages, the Struts action forms and the servlet required to implement the web user interface. Struts is completely hidden from the specifier.

A DCI-WEB page is divided in three zones : hierarchical navigation bars at the top, which act like menus in a native graphical user interface; options, which appear on the left-hand side and whose content is determined by the items selected in a navigation bar; the core content of a page appears in the remaining area and is determined by the option selected. A DCI-WEB specification is made of four parts, separating format, content and control: i) a description of the hierarchical navigation bars and options of the application, ii) a description of the core content of each page (i.e., a list of fields) and the transitions between the web pages, iii) a description of the layout of the core content of each page, written in HTML and augmented with new tags that describe where fields are positioned and their display instructions, and iv) an interface specification of the IS that is called by the web pages to obtain page content.



List of loans
Member ID : 12

Book Id	Title	Action
1	The B Book	Return
2	Software Specification Methods	Return

Fig. 5. A DCI-WEB generated web page for listOfLoans

The following simple DCI-WEB specification contains two pages and a transition between them. Page selectMember contains two fields: an input field memberId and a button displayLoans to submit the request. It also contains a transition defined by the if statement; its condition refers to the button and its then part contains a call to display page listOfLoans. This page contains a list, denoted by operator *, and whose components are elements of a Cartesian product (i.e., a tuple). It also contains a transition to execute a return action for the corresponding book. Figure 5 shows the generated page for listOfLoans. The HTML specification of the page is omitted.

```
page selectMember
  {memberId : int ?; displayLoans : button;}

  if (displayLoans)
    { listOfLoans(
        memberId = memberId,
        loans =⟨Library.getLoans(memberId)⟩)
    }

page listOfLoans
  { memberId : int;
    loans : (bookId : int, title : string, return : button)*
  }

  if (loans.return)
    { ⟨Library.return(memberId, loans.selected.bookId)⟩;
      listOfLoans(loans =⟨Library.getLoans(memberId)⟩)
    }
```

Separating format, content and control makes the user interface description simpler to analyze with tools. The specification abstracts from implementation and programming details, making it more tractable for automated analysis.

C. EB^{3}TG

To avoid keeping track of the system trace, which would be difficult because of its increasing size, attribute definitions are implemented by a relational DB (RDB). A RDB transaction is generated by EB^{3}TG for each action in the ER diagram. Thus, every time an event is considered as valid by EB^{3}PAI, then the corresponding transaction is called to update the attributes affected by the action. EB^{3}TG supports the generation of the RDB schema from an XML representation of the ER diagram. It parses the EB^3 attribute definitions and generates a set of Java classes that implements the transactions using JDBC (Java Database Connectivity). It can be connected to various DB management systems (DBMS) like Oracle, PostgreSQL and MySQL, handling the subtle variations in the SQL syntax.

To generate a program that executes a RDB transaction associated with an action a, the input clauses of the attribute definitions are analyzed in order to determine: i) the set $Att(\mathsf{a})$ of attributes affected by the execution of action a, ii) the set $T(\mathsf{a})$ of tables of the DB affected by a, and iii) the effects of a on the attributes. The general algorithm is:

```
(1) translate the ER diagram into a RDB schema
(2) for each action a
(3)     analyze the input clauses
(4)         determine $Att(a)$
(5)         determine $T(a)$
(6)         for each table $t$ in $T(a)$
(7)             determine the key values to delete
(8)             determine the key values to insert
                    and/or to update
(9)     define the transaction for a
```

The different steps of the algorithm, which are summed up in this section, are detailed in [15], [16], [14].

Step (1): The DB is generated from the EB^3 specification. We use standard algorithms from [8] to translate the ER diagram into a RDB schema.

Steps (4)-(5): To execute an attribute definition, all the input clauses are analyzed, and the first input clause that matches with the last event of the trace is the one executed. An attribute is affected by action a if there exists at least an input clause of the form $\mathsf{a}(\overrightarrow{p}) : expr$ in its definition.

Steps (7)-(8): To execute an attribute definition, if an input clause matches with the last event of the trace, then an assignment of a value for each free variable in the input clause has been determined. For instance, let us suppose that $\mathsf{Lend}(b0, m1, \text{“}Classic\text{”})$ is the last valid event of the IS. In that case, to evaluate the function $dueDate$ (see Sect. III-B.4), input clause $\mathsf{Lend}(bId, mId, type)$ matches with the event and free variables bId and mId of the attribute definition are bound to values $b0$ and $m1$, respectively. When expression $expr$ in an input clause contains **if then else end** expressions, then the different conditions in the **if** predicates must be analyzed to determine the values of the key attributes that are not bound by the pattern matching. We use binary trees called *decision trees* to analyze such predicates; their construction and analysis are detailed in [15]. **SELECT** statements are then generated from the decision trees to characterize the key values to delete, update or insert. We have identified the most typical patterns of predicates and their corresponding **SELECT** statements in [15].

The key values to delete are in expressions $expr$ of the form $eKey(front(s)) - \{k\}$ in the input clauses of key definitions. A **DELETE** statement is then generated. The other key values correspond either to insertions or updates. We cannot distinguish key values to insert from key values to update because, in expressions $expr$ of the form $eKey(front(s)) \cup S$, sets $eKey(front(s))$ and S are not necessarily disjoint. To be consistent with respect to the EB^3 specification, we have chosen to implement such expressions by an update, followed by an insertion.

Step (9): The ordering of SQL statements in the generated transactions is the following.

1) list of the **SELECT** statements identified by the analysis of the input clauses,
2) list of **DELETE** statements,
3) list of SQL statements for insertions and/or updates.

We use a high-level pseudo-code to describe the synthesized transactions; this pseudo-code is translated into Java in EB^{3}TG. For instance, the transaction generated for action Lend is:

```
TRANSACTION Lend(bId : bK_Set, mId : mK_Set,
            typeOfLoan : Loan_Type)
VAR TEMP1 : INT
    SELECT CurrentDate + loanDuration INTO #TEMP1
    FROM member
    WHERE memberKey = #mId;
UPDATE loan SET borrower = #mId
WHERE bookKey = #bId;
IF SQL%NotFound
THEN
    INSERT INTO loan(bookKey,borrower)
    VALUES (#bId,#mId);
END;
IF typeOfLoan = “Permanent”
THEN
    UPDATE loan SET dueDate = CurrentDate + 365
    WHERE bookKey = #bId;
    IF SQL%NotFound
    THEN
        INSERT INTO loan(bookKey,dueDate)
        VALUES (#bId,CurrentDate + 365);
    END;
ELSE
    UPDATE loan SET dueDate = #TEMP1
    WHERE bookKey = #bId;
    IF SQL%NotFound
    THEN
        INSERT INTO loan(bookKey,dueDate)
        VALUES (#bId,#TEMP1);
    END;
END;
COMMIT;
```

where a program variable is prefixed by the symbol “#”, in order to distinguish it from attribute names.

TEMP1 is an example of temporary variable that is defined by EB^{3}TG in the host language, when a record to manipulate is determined by a predicate or a recursive call of attribute. It corresponds to the case where the loan is of type “Classic”. A temporary variable is defined when there is a unique record, while a temporary table is used to store several records. The generation of **SELECT** statements that correspond to these records is discussed in [15].

The variable "SQL%NotFound" contains a value returned by the DBMS to determine whether the update has modified a record in the table. Indeed, the analysis of attribute definitions is not sufficient to distinguish updates from insertions. In that case, tests are defined by EB^{3}TG to determine whether the key values already exist in the tables. These update/insert combinations can be simply optimized by analyzing the IS trace to determine whether the corresponding action is an entity producer or modifier.

The algorithmic complexity of the synthesized transactions is similar to that of typical equivalent programs written by a programmer. The tool EB^{3}TG has been implemented in Java. The code includes 50 classes, 625 methods and 20 KLOCs.

V. FUTURE DEVELOPMENT

We have identified six major themes. The first three themes deal with extensions to EB^{3}PAI and EB^3. The last three focus on specification maintenance, validation and reuse. Validation consists in checking properties about the specification, in order to make sure that it is appropriate. Reuse is about building a specification from existing ones, instantiating IS patterns, and interfacing with legacy systems. In the next sections, we describe our proposition for each theme.

A. *Improvements to* EB^{3}PAI

One of the performance bottlenecks in EB^{3}PAI is the OO database which is used to ensure persistence of the interpreter's state. It consumes 50 % of the transaction time. We could perform some fine tuning, in order to reduce disk IO. OODBMS offer several options which we have to investigate (like grouping sub-objects to retrieve them in one disk read, implementations of maps using hash tables or b-trees). Another option is to implement our own persistence manager using serialization of objects and a conventional relational database, taking advantage of the structure of the process expression.

Another optimization is to transform a process expression into an extended labelled transition system (ELTS). Since we use quantifications on very large sets for interleave and choice, we cannot use an ordinary labelled transition system to represent a process expression; it would require too much space due to combinatorial explosion. An ELTS contains hierarchical states which can efficiently represent complex process expression and avoid state explosion inherent to standard LTS. For instance, an interleave $P|||Q$ is represented by a state which contains two substates, one for P and one for Q. Similar structures are used in hierarchical state machines like Harel's Statecharts. However, they must be extended to cater for quantification, nesting of quantifications, sequential composition, etc. ELTS should be more space efficient than the current version of EB^{3}PAI, because their structure is static. In EB^{3}PAI, a state is represented by a process expression tree. At each transition, new tree nodes must be allocated, since the new tree is constructed from some parts of the old tree and new nodes are allocated to represent the differences between the old state and the new state. We have conducted an experimentation with ELTS on the library case study, and it shows that response time can be reduced by 50 %. We must now formalize the concepts of ELTS, define algorithms that translate a process expression into an ELTS and determine sufficient conditions under which an efficient interpretation is obtained.

Since we use a trace semantics for EB^3 process expressions, we could also increase performance by transforming a process expression into a trace equivalent deterministic one. Deterministic expression can be executed faster, since the first transition found is executed. They are also easier to translate into an ELTS. A number of simple laws are known for classic process algebras like CSP and CCS. It remains to adapt and prove them for the EB^3 process algebra and find additional (more sophisticated) laws.

B. *Structured Action*

An action in EB^3 is triggered by a user input and it must provide a response. It is atomic: its effects are either performed in their entirety or not at all.

In EB^3, an action cannot be defined in terms of other actions. This constitutes a serious limitation for modeling IS. For instance, consider adding a transfer action in a library system, which allows a member to transfer his loan to another member. It is equivalent to the sequential composition of two basic actions, return, to terminate the current loan of the first member, and lend, to start a new loan for the other member. Since this transfer action must be atomic from a transactional viewpoint, it must be specified as an action, and its execution must bring the process in the same state as if a return followed by a lend had been executed. This leads to a surprisingly complex specification, losing all the simplicity of the initial specification. We may consider more complex examples, like a comprehensive terminateMembership action that cancels the membership of a member and all its dependencies (reservations, loans, fines, and so on). In this case, the number of basic actions to which terminateMembership corresponds evolves over time.

To overcome this limitation and preserve simplicity and readability of specifications, we would like to specify actions like transfer and terminateMembership, which we henceforth call *structured actions*, as a composition of other actions. Let $main$ be an IS specification, and let ω be a structured action defined by the sequence $\sigma_1, \ldots, \sigma_n$ of basic actions. To execute ω on $main$, we must execute $\sigma_1, \ldots, \sigma_n$ on $main$. This preserves simplicity since entity attributes can be defined simply in terms of $\sigma_1, \ldots, \sigma_n$.

A structured action ω could be defined with the usual process algebra operators, but it is not sufficient. We must introduce quantification of sequences using lists instead of sets, in order to deal with actions like terminateMembership. A very nice feature would be to define terminateMembership as a goal to be reached using a set of actions (e.g., cancel reservation, return). It would relieve the specifier from stating the ordering constraints again.

These modifications to the EB^3 notation entail significant modifications to EB^{3}PAI: it must deal with nondeterministic structured action, goal oriented definitions, recursivity.

C. Input-Output Rules

Input-output rules define the output for each action. For instance, the rule

```
Rule
  input    ShowBookDescription($bId$)
  output   $\langle bId, title(bId), name(author(bId))\rangle$
End
```

displays the book id, the title and the author of the book, by using the corresponding attributes from the class diagram. These rules are currently not supported by the APIS framework. They should be called by EB^{3}PAI upon the reception of a valid action. The current EB^3 notation allows for the specification of tuples based on entity attributes, and sets defined by an SQL SELECT statement on the database generated from the class diagram. Implementation of these rules should be straightforward; it should reuse some components that will be developed for the evaluation of guards in process expressions.

We have noted from the work on the generation of SQL transactions [15] that expressions based on entity attributes, which are defined as functions, are significantly more concise and simpler than equivalent SQL statements. For instance, a set expression like

$$\{x, f_1(x), f_2(f_3(y)) \mid x \in key_1 \land y \in key_2 \land f_4(x) = f_5(y)\}$$

concisely denotes a complex SELECT statement that we could generate: the free variables x and y denote the Cartesian product of two tables; the expression $f_2(f_3(y))$ corresponds to a join between the tables of f_2 and f_3; assuming that f_4 and f_5 denote set-valued roles in the class diagram, expression $f_4(x) = f_5(y)$ corresponds to a sub SELECT statement that must compute a "for all" query which is typically hard to code in SQL for the average programmer (either using double negations, set operators or counts). Note that the functional style of EB^3 attributes differs from the traditional relational calculus and relational algebra used in database theory, which are slightly less compact; our notation is closer to OQL expressions and object-oriented expressions (e.g., $f_2(f_3(y))$ corresponds to $y.f_3.f_2$).

D. Maintenance

User requirements changes inevitably occur during the life of a system. Maintenance is about the evolution of the IS specification, in order to adapt to new user requirements or changes in the environment. Maintenance should be conducted on the IS specification only, not on the generated implementation, which should be completely hidden from the specifier.

Let E_2 be a new specification derived from E_1 to accommodate new user requirements. Since we use specification interpretation, we must propagate these modifications to the current state of the system, which is a process expression E_1' reached from E_1 after executing all inputs submitted since system startup, i.e., $E_1 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_n} E_1'$. To appropriately reflect specification changes in the running system, we must find a specification E_2' such that $E_2 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_n} E_2'$. One strategy is to analyze the difference between E_1 and E_2 and to represent the modifications as a combination f of syntactic add, replace and remove operations such that $E_2 = f(E_1)$. Then, we must determine how f could be applied to E_1' to obtain E_2', without replaying the execution of the system trace $\sigma_1 \ldots \sigma_n$, which would be too expensive to compute. The syntactic operator f could be used to define a morphism between the syntax trees of E_1 and E_2, or their transition graphs (when they can be finitely and compactly represented). The result of the syntactic modifications must be validated by the specifier before they can be applied to the system state, because the application of f to E_1' is heuristical in nature.

If we use an ELTS as described in Section V-A, then the modifications into the executing system correspond to mapping the old ELTS into the new ELTS. These differences should be derivable also by analyzing the modifications and representing them as a syntactic combinator f as described for process expressions. Changes are easier to make in an ELTS than in a process expression, because ELTS states are static and enumerated prior to runtime.

For modifications to the entity attributes, we can use the traditional approach for IS, which is to change the database schema and update the database content using SQL update statements and conversion programs.

E. Validation of IS Specifications

Given an EB^3 specification, we would like to prove or to verify properties about the specification, in order to check its appropriateness for the user requirements at hand. We intend to address two kinds of properties : state invariance properties, (e.g., checking that a member can never exceed its loan limit) and dynamic properties (e.g., checking that two lend actions cannot be executed for the same book without having a return in-between). State invariance properties are hard to prove or check in a process algebra; however, they are easier to prove in a state-based notation like B or Z using standard weakest-precondition techniques. We intend to check state invariance properties by translating an EB^3 specification into an equivalent B specification and adding the properties to prove as invariants. We have already derived algorithms that can generate most of the B specification; we still need to generate operation preconditions that exactly represent the ordering constraints defined by the process expressions. However, these preconditions must be expressed in terms of entity attributes, whereas the EB^3 process expression does not contain any reference to entity attributes. For instance, the precondition c of an operation representing an action a corresponds to the set of process expressions reachable from $main$ that can execute a. Relating these expressions to c is not trivial. We plan to manually determine the preconditions for the typical patterns found in [12] and prove the equivalence with the process expressions once for all; a first step in that direction is described in [17]. A new specification can be constructed by instantiating the patterns and their preconditions can be adapted to take into account specific requirements; guards can be propagated and added to the preconditions.

For the user interface, the specification language described in Section IV-B should enable us to verify properties, using model checking. For instance, one may be interested to check that a user can change his shopping bag before payment is confirmed, or that the price of a book is always displayed. Since our GUI specification language is more abstract than a concrete program, it should be less subject to combinatorial explosion. However, we will have to deal with transition guards that may depend on outputs produced by an external service. To conduct the analysis, we will have to lift (extract) information from the specification of these services or used manually added hypotheses to guide the analysis. We have explored these problems in [9] and intend to build on this expertise.

F. Reuse of IS Specifications

A major trend in IS development is building new systems from existing ones. The IT domain is obviously entering an age of large systems integration through legacy modernization projects and enhancement projects. We want to study the mechanisms by which a specifier can take a specification S and use it to build a specification S' for a new environment. In particular, we want to consider specification componentisation, reuse by pattern instantiation and adaptation, specification inclusion and extension (UML like), and finally, specification integration, orchestration and choreography.

We have studied in [12] various patterns of EB^3 specifications determined by the structure of the class diagram (e.g., one-to-many relationship, weak entity, recursive association). These patterns are defined in an abstract but informal manner. We want to formalize them in order to support pattern instantiation and composition in a proper way, so that properties which are verified on an abstract pattern can be preserved through instantiation and composition.

Patterns can also be more specific and describe a particular application domain, for instance, an accounting system or a human resource management system, which should be very similar from one organization to another. But, we must also support instantiation and feature insertion, modification or deletion, in order to account for the specifics of an organization and deviations from the patterns. We hope that by defining operators for that purpose, properties checked for the abstract pattern can also be preserved. The instantiation of a pattern must permit the definition of specific attributes of classes and associations, and ordering constraints.

Object orientation is a key feature for reuse at the design and programming level. We want to adapt the object-oriented notion of inheritance to process expressions and functions. The meaning and adaptation of these concepts have been studied at the specification level, in the context of hybrid approaches, merging process algebra and model-oriented specifications, like *OhCircus* [5], which unifies classes and processes. Their notion of inheritance for attributes and operations is similar to a programming language approach; as for processes, they can be extended and the extension is composed in parallel with the inherited process. This is an option that can be included in EB^3, but other modes of composition, using choice, interleaving and guards for example, must be studied in order to streamline the specification activity.

Specification construction by stepwise refinement is another option to explore. Numerous notions of refinement have been proposed over the past decades. Generally speaking, refinement involves the notion of property preservation: a specification S' refines a specification S if S' does not contradict S. This notion can be interpreted in various ways, for instance by adding new actions, decomposing actions or restricting the behaviour (traces).

Another form of reuse is to connect an EB^3 specification to a legacy system. We must study basic protocols for connecting systems together (in a SOA style) and identify various mechanisms through which applications can efficiently and safely communicate. This form of reuse induce a new EB^3 specification style where the black boxes are now aligned with some enterprise business models and enterprise architecture models. By aligning the specifications with these models, we want to enable componentisation, service design and integration and service choreography. We also want to study the addition of system actors or users to an EB^3 specification. This addition will permit another type of reuse, which is system orchestration.

VI. Discussion

The APIS project distinguishes itself from the current state of the art in two fundamentals ways. First, the evolution in the industrial practice of IS development is centered around *design issues*: J2EE, SOA, MDA and others are making progress on the definition of standards and components that can be used to *build* systems, but they do not operate at the same level of abstraction as the notations proposed in APIS. The project is about developing *abstract notations* for specifying the *business logic* of an IS and supporting tools for automatically translating a specification into an implementation. The specification notations used do not deal with design issues; the synthesis algorithms and tools that will be developed will rely on the best industry practices at the design level (e.g., J2EE, SOA). Hence, the project is complementary to the current state of the art in industry.

An EB^3 specification allows for the decomposition of an IS into a set of services which can be externally called, as promoted by SOA. The EB^3 process algebra is similar to BPEL, but more powerful, since it allows quantifications on interleave and synchronization. BPEL doesn't define the behavior of atomic actions it calls, whereas EB^3 supports the complete abstract specification of these actions. EB^3 is also more abstract than the notations used in MDA, which are better suited for architectural design (e.g.,using UML) than for the abstract specification of functional behavior. Generative programming is very close in spirit to our approach; EB^3 is more event-driven and more abstract than existing approaches, and provides a comprehensive notation that covers both the GUI and the IS services. In addition, we intend to conduct validation through proof and model checking, something which is not typically addressed in MDA or generative programming.

The second fundamental distinction is with the current focus of research in formal methods. The bulk of the research in formal methods addresses safety critical systems, embedded systems, distributed systems and hardware design. Few are addressing IS and none in a way as comprehensive as the APIS project, since it covers the user interface and the functional behavior. Of course, we share the same mathematical foundations, for instance the EB^3 process algebra is inspired from well known process algebras, but our focus on synthesis leads us to explore new aspects that haven't been addressed so far by the formal methods community, like efficiently executing large quantifications in process expressions or removing nondeterminacy in IS specifications.

References

[1] AndroMDA, http://www.andromda.org.

[2] Andrews, T. *et al*: Business Process Execution Language for Web Services, May 2003, version 1.1., http://www-128.ibm.com/developerworks/library/specification/ws-bpel.

[3] Booth, D, *et al*: Web Services Architecture, February 2004, http://www.w3.org/TR/ws-arch.

[4] Butler, M., Ferreira, C., Ng, M. Y.: Precise Modelling of Compensating Business Transactions and its Application to BPEL. *Journal of Universal Computer Science* **11**(5) pp. 712–743, 2005.

[5] Cavalcanti, A., Sampaio, A., Woodcock,J.: Unifying Classes and Processes, *Software and System Modeling*, **4**(3), July 2005, pp 277–296.

[6] G. Cousineau and M. Mauny. *The functional approach to programming*. Cambridge University Press, Cambridge, 1998.

[7] Czarnecki, K.: Overview of Generative Software Development. In J.-P. Banâtre et al. (Eds.) *Unconventional Programming Paradigms (UPP) 2004*, Mont Saint-Michel, France, LNCS 3566, pp. 313-328, 2005.

[8] Elmasri, R., Navathe, S.: *Fundamentals of Database Systems*. Addison-Wesley, 2004.

[9] Evans, N., Treharne, H., Laleau, R., Frappier, M.: How to Verify Dynamic Properties of Information Systems, 2nd IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society Press, Beijing, China, September 26-30, 2004, 416–425.

[10] Fraikin, B., Frappier, M, Laleau, R.: State-Based versus Event-Based Specifications for Information Systems: a Comparison of B and EB^3, *Software and System Modeling*, Springer-Verlag, **4**(3), July 2005, 236–257.

[11] B. Fraikin. Interprétation efficace d'expression de processus EB3. PhD thesis, Université de Sherbrooke, Québec, Canada, 2006.

[12] Frappier, M., St-Denis, R.: EB^3: an Entity-Based Black-Box Specification Method for Information Systems. *Software and System Modeling*, **2**(2), July 2003, pp 134–149.

[13] Garavel, H., Lang, F., Mateescu, R. An Overview of CADP 2001. In *European Association for Software Science and Technology (EASST) Newsletter*, August 2002, vol 4, 13–24.

[14] Gervais, F., Batanado, P., Frappier, M., Laleau, R.: EB^{3}TG: A Tool Synthesizing Relational Database Transactions from EB^3 Attribute Definitions, 8th International Conference on Enterprise Information Systems (ICEIS 2006), Paphos, Cyprus, May 24-27, 2006, INSTICC Press, Volume Information Systems Analysis and Specification, 44–51.

[15] Gervais, F., Frappier, M., Laleau, R., Batanado, P.: EB^3 attribute definitions: Formal language and application. Technical Report **700**, CEDRIC, Paris, France, February 2005. http://www.dmi.usherb.ca/˜frappier/Papers/RC700.pdf

[16] Gervais, F., Frappier, M., Laleau, R.: Generating Relational Database Transactions from Recursive Functions Defined on EB^3 Traces, 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany September 5-9, 2005, IEEE Computer Society Press, 117–126.

[17] Gervais, F., Frappier, M., Laleau, R.: Gervais, F., Frappier, M., Laleau, R.: Refinement of EB^3 process patterns into B specifications. 7th International B Conference (B 2007), 2007, LNCS, Springer-Verlag, *to appear*.

[18] Groote, J.F., Ponse, A. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, Springer Verlag, 1995, 26–62.

[19] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.

[20] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: A Language Supporting Multi-Path Development of User Interfaces, In R. Bastide, P. Palanque, J. Roth (Eds) *Engineering Human Computer Interaction and Interactive Systems (EHCI-DSVIS) 2004*, Hamburg, Germany, July 11-13, 2004, LNCS 3425, pp. 200–220, 2005.

[21] Mammar, A., Laleau, R.: From a B Formal Specification to an Executable Code: Application to the Relational Database Domain, *Information & Software Technology*, Elsevier, **48**(4), pp. 253–279, 2006.

[22] Mammar, A., Laleau,R.: UB2SQL: A Tool for Building Database Applications Using UML and the B Formal Method, *Journal of Database Management*, 17(4), 2006, 70–89.

[23] Miller, J., Mukerji, J.: MDA Guide Version 1.0.1, Object Management Group, 2003, http://www.omg.org/mda/specs.htm.

[24] Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-Method Approach for Information Systems Modeling: from Object-Oriented Conceptual Modeling to automated programming, *Information Systems*, 26, pp. 507–534, 2001.

[25] Puerta, A.R.: A Model-Based Interface Development Environment. *IEEE Software*, **14**(4), pp. 41–47, July/August 1997.

[26] Puerta, A., Eisenstein, J.: XIML: A Universal Language for User Interfaces, http://www.ximl.org/documents/XimlWhitePaper.pdf.

[27] Terrillon, J.-G.: Description comportementale d'interface web. Master's thesis, Département d'informatique, Université de Sherbrooke, 2005.

[28] Vanderdonckt, J., P. Berquin: Towards a Very Large Model-Based Approach for User Interface Development. In *UIDIS'99: User Interfaces to Data Intensive Systems*, IEEE Computer Society, 1999.