

APIS¹ - Automatic Production of Information Systems

M. Frappier, B. Fraikin, M. Richard
Département de mathématiques
et d'informatique
Université de Sherbrooke, Sherbrooke,
Québec, Canada, J1K 2R1
{Frappier,Fraikin,Richard}@dmi.usherb.ca

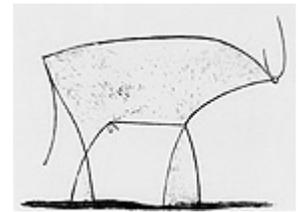
R. Laleau
Laboratoire CEDRIC
Institut d'Informatique d'Entreprise
Conservatoire National des Arts et Métiers
18 allée Jean Rostand
91025 Évry Cedex France
laleau@iie.cnam.fr

1 Synopsis



The objective of the APIS research project is to develop a case tool that generates executable information systems (IS) from formal specifications (abstract models). In other words, APIS aims at automating as much as possible the production of IS by lifting the level of abstraction at which software designers work. Instead of spending most of their time designing, programming and testing at a very low level of abstraction, software designers will be able to concentrate on specifying and validating; the APIS case tool will take care of generating an executable system from the specification.

The foundations of the case tool are abstract models of IS (formal functional specifications) and algorithms that generates an executable system from these models (model-based interface development environment – MB-IDE), and specification interpretation. The APIS project has to a) define a comprehensive set of integrated models for information systems, b) develop new algorithms for specification interpretation and code generation, and c) implement them in a case tool. The EB³ specification language will be the foundation for the models [5]. It is a formal, object-oriented, executable language which is very well adapted for the functional specification of IS. EB³ will be supplemented with a language to specify *graphical user interfaces* (GUI) in order to have a comprehensive model of an IS.



The APIS project will be the first to study the automation of information systems production from formal specifications. The success of the APIS project would represent a significant breakthrough in terms of *productivity* and *quality* in IS development. APIS could cut up to 50 % of the cost of IS development, because it would relieve software designers from design, programming and testing. Consequently, it would also avoid faults introduced during these activities.

2 Background and Related Work

2.1 Information Systems

IS are generally characterized by large data structures which are modified or queried by several users in concurrency. The difficulty of these systems typically resides in managing complex relationships between data structures, in complex calculations involving several data structures, in processing large volume of data, and in preserving data integrity through concurrent updates by several users. IS typically have little hard real-time constraints or inter-process communication.

An IS can be decomposed in three parts: i) the functional behavior (also called business logic), ii) the user interface, and iii) the database. The functional behavior defines the *transactions* of the system, which are of two types: a) *update* transactions modify the internal state of the system which is stored in a database, b) *inquiry* transactions display information about the system state (database).

The recent technological developments in IS mostly apply to the *programming* phase. Database management systems (DBMS), key elements of IS, now offer sophisticated distribution and concurrency control mechanisms, support very large databases, and offer sophisticated querying facilities. Technical breakthroughs like INTERNET, JAVA, CORBA, COM, EJB, frameworks and others make it easier to program distributed information systems or deploy them on the web on various platforms. Some DBMS case tools can generate forms to update-modify-delete records in a table, but this is insufficient to build a real GUI for an information system.

¹Egyptian mythology — a god depicted as a bull, symbolizing fertility and strength in war [OED]. Drawings by Pablo Picasso, 1945.

IS share strong similarities with each other. Nevertheless, each development project reinvents the wheel, more or less. The current industrial technology is very good at defining abstract database models and translating them into concrete database schemas, but it is still very weak for specifying the functional behavior and the GUI. The functional behavior is informally and partially described; its single precise description is the IS source code itself, which means that it is buried among various implementation details, complicating its maintenance. Each window of the GUI and the dialogue between the user and the system are manually derived and coded; the business logic of each transaction is manually coded.

Industrial case tools supporting information system 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 key in reducing development cost and increasing quality clearly resides in suppressing/mechanizing these three tasks.

2.2 Model-Based Interface Development Environments

Model-based interface development environments (MB-IDE) [8] address the issue of generating a 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, platform, etc. 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 (e.g., [4]) were able to generate a standard (single style) interface from a data model. Recent approaches have focused on providing more flexibility for specifying a wider ranger of GUI styles [9, 11] 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 are still at the research prototype level; there is no commercial product available. The industrial tool PollenLight, from e-Companion, shares similarities with MB-IDE. It generates a GUI from UML models, which are not as complete and sophisticated as the models used in MB-IDE or in formal specifications.

2.3 The EB³ Specification Language

The EB³ specification language is a formal, object-oriented, executable specification language developed at the University of Sherbrooke by Profs. Marc Frappier and Richard St-Denis. It was specifically designed for the abstract functional specification of information systems. It is founded on process algebra, traces, and JSD entities. It supports modular, iterative (incremental) specification.

Figure 1 provides a sample of an EB³ specification. It contains : a) a class diagram (called a business model), which defines entity types with their external events, attributes and associations; b) entity type specifications, which define the scenarios (valid sequences of input events) of entities using process expressions; c) entity attribute definitions, which are recursive functions on the system trace. A specification can also contain input-output rules to specify outputs for input traces, or SQL expressions can be used to specify queries on the business model.

EB³ is scenario oriented and more abstract than traditional state-machine oriented specification languages like B, Z or VDM. Entity process expressions provide a more explicit representation of the system behavior; they can be graphically represented using their entity structure diagrams (JSD-like). Attribute definitions are encapsulated in a single expression, which facilitates their understanding and maintenance. To ease the integration of EB³ in an industrial environment, we have also defined a UML-like syntax for EB³ which looks more familiar to practitioners; it allows them to specify using traditional UML construct [10].

2.4 UML

The *de facto* standards in industry for modeling information systems are UML and structured analysis techniques. UML contains several diagrams; some of them contain the same information, but it is represented in different

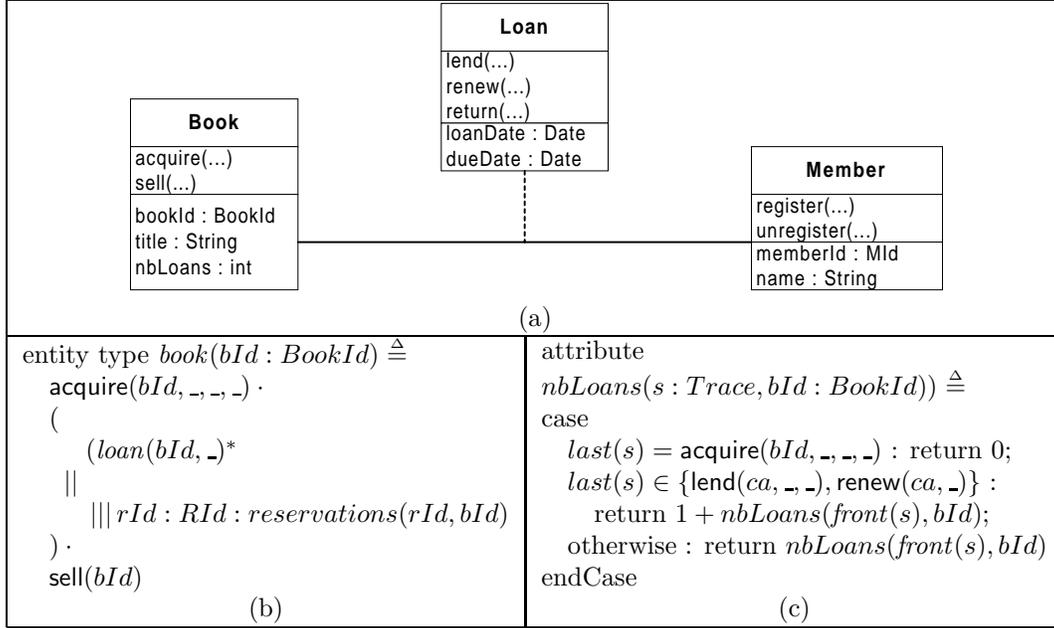


Figure 1: An example of an EB³ specification

formats. In [7, 12], the authors show how sequence diagrams can be translated into a statechart diagram. The first approach [7] uses a framework, called the minimally adequate teacher, to synthesize a state machine by interacting with the designer, in order to obtain additional details not available in the sequence diagrams. The second approach is automatic (no user intervention). Conflicts between sequence diagrams are detected and resolved. In both approaches, the resulting statecharts must be refined by designers in order to completely specify a system. Actions are defined using the Object Constraint Language (OCL) or by providing executable code.

Statecharts are orthogonal to EB³ specifications, in terms of specification structure. The first distinction between them is very similar to the distinction between an automaton and a regular expression (roughly speaking). Moreover, a statechart is an extended state machine: guarded actions on transitions modify global state variables. In EB³, the system state is given by a process expression and the system trace. Global state variables in statecharts correspond to entity attribute definitions in EB³. The behavior (i.e., evolution over time) of a global state variable is determined by the guarded actions modifying it. To understand the behavior of a global variable, one has to look through several actions. In contrast, the behavior of an entity attribute is defined in a single place, a recursive function on the system trace. We believe that this locality of information can simplify system understanding and streamline maintenance. Generally, adding an attribute to an entity does not induce any change to the rest of the EB³ specification; one simply has to write its recursive function on the system trace. In statecharts, one has to modify each action that affects the value of the global state variable.

3 Detailed Proposal

3.1 Research Problems

There are four main problems to solve in order to build a case tool that will automatically generate IS from specifications.

1. Define a comprehensive, formal IS specification language which would cover both the functional behavior and the GUI.
2. Define an efficient interpreter of IS specification.
3. Define algorithms to generate a GUI from the IS specific.
4. Define an interface between APIS generated systems and existing systems.

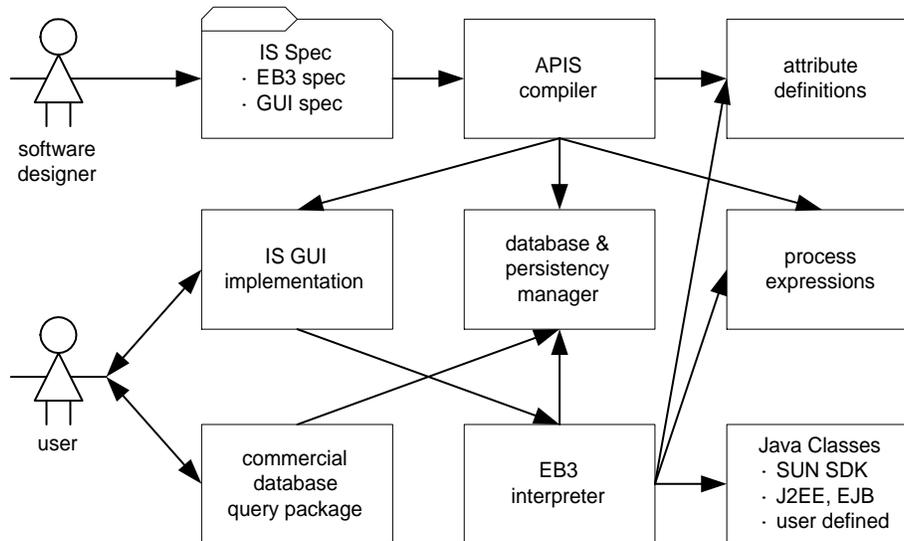


Figure 2: The architecture of the APIS tool

The first problem addresses the integration of two technologies. The current work on MB-IDE tackles the specification of the user interface alone, without describing the functional behavior of the system. Existing work on functional specification does the exact opposite. Obviously, the functional specification and the GUI specification have a lot in common; they both contain an application model, a data model, and a task model. Consequently, someone wishing to use the existing technology must repeat some information twice, in different formats, which is inefficient and constitutes a potential source of inconsistencies. Hence the first problem is to define a comprehensive metamodel of an IS.

The second problem is to develop a compiler or an interpreter for the functional behavior specification of the IS. Existing interpreters are too inefficient. They can handle a single user and a very small volume of transient, simple data. Any reasonable interpreter for an information system must support large volume of persistent data, multiple users in concurrency, data integrity through concurrent access, and easy data querying. The compiler/interpreter should cover the basic, common behavior found in the majority of IS, and provide hooks for manually developed solutions for more complex parts of the specification.

The third problem is to develop a program which generates the GUI from the IS specification. This problem is *partially* solved in the literature: there exists several research lab prototypes of MB-IDE, but they all use their own abstract models of the GUI; we cannot reuse these models as is, because they are not integrated to a functional specification. In addition, a complete description of their models, knowledge base and GUI generation algorithms is not available in the public domain; the MB-IDE literature is often limited to a high level description of the MB-IDE capabilities. Finally, each MD-IDE implementation is targeted towards a specific windowing package (e.g., Java and Swing) for a specific operating system (UNIX, Mac, Windows). The GUI generator should provide a default interface for an EB³ specification, in order to allow quick prototyping and validation with the users; this interface should be customizable by providing a GUI specification to refine the dialogue model, the presentation model and the user model.

The fourth problem addresses the issue of the interface between systems developed with a traditional approach (henceforth called *traditional systems*) and systems developed using our automated approach. It should be possible to access the database and functions of a traditional system from an IS specification. Dually, it should be possible to have read access to entities of an IS specification from a traditional system.

3.2 Methodology

Figure 2 illustrates the architecture of the APIS case tool. In the top left corner, the software designer creates an IS specification, which is composed of an EB³ specification and a GUI specification. The APIS compiler processes this specification and generates an implementation for the GUI, a database schema, compiled process expressions

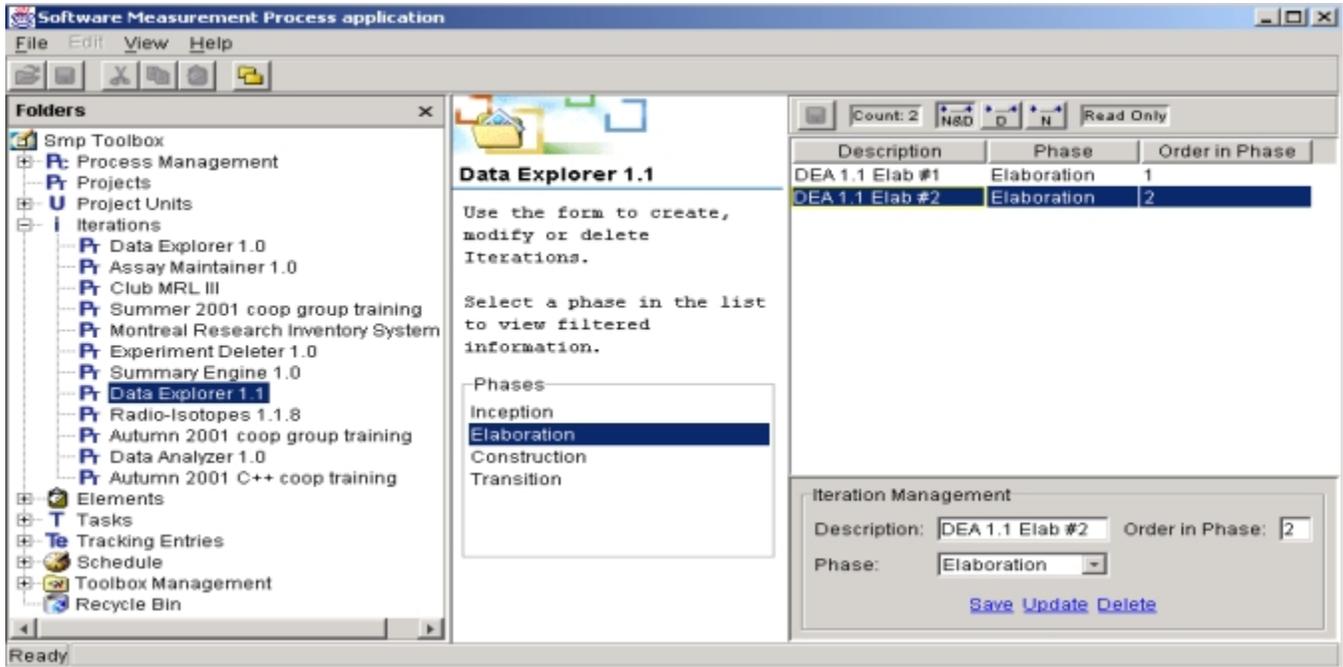


Figure 3: Sample default layout of the GUI

and compiled attribute definitions. The user may then interact with the IS by executing the GUI. User input are submitted by the GUI to the EB³ interpreter. The interpreter uses the compiled process expressions (represented as abstract syntax trees), the compiled attribute definitions (represented as abstract syntax trees and a database schema), predefined Java classes and the database to process an input and compute an output which is returned to the GUI for display to the user. The user may also interact with the IS database using a commercial database query package.

We will now describe our strategy for designing each box of the architecture. APIS will be implemented in Java, to foster portability and to benefit from the huge repository of components now available on the market. Our strategy could also work with other languages like C++. The IS specification language must support reuse of these components through the definition of hooks. The database support will be provided by IBM's Discovery Link package, which provides a uniform access to heterogeneous, distributed databases. We will use the J2EE standard and Enterprise Java Beans to foster scalability, distribution, portability and security.

IS Specification Language. As mentioned before, we will use the EB³ specification as a basis and extend it with a GUI specification language. The EB³ language already contains information about transactions, data, and tasks (an ordering of transactions). Figure 1 illustrates part of a specification. It remains to add GUI specific aspects : the dialogue, user and presentation models. Their semantics will be defined by rules mapping IS specification elements into GUI implementation elements. The APIS compiler will implement these rules in Java.

GUI Implementation. The APIS compiler must generate the GUI implementation from the IS specification. As an example of the structure of a generated interface, consider the window in Figure 3. The leftmost part contains the entity types with their associations. The middle part contains information about the entity type and some filtering capabilities on entity instances. The upper rightmost part lists some entities of the selected entity type. The lower rightmost part lists the transactions which can be applied to the entities. Entity types, associations, attributes, and transactions are extracted from the EB³ spec. The remaining information is extracted from the rest of the IS (the GUI specification part).

The example window of Figure 3 comes from an IS we recently developed in cooperation with an industrial partner. During this project and several others, we have acquired strong knowledge of this interface architecture. We also completed a project where we implemented the generation of a rudimentary interface (see Figure 4) from

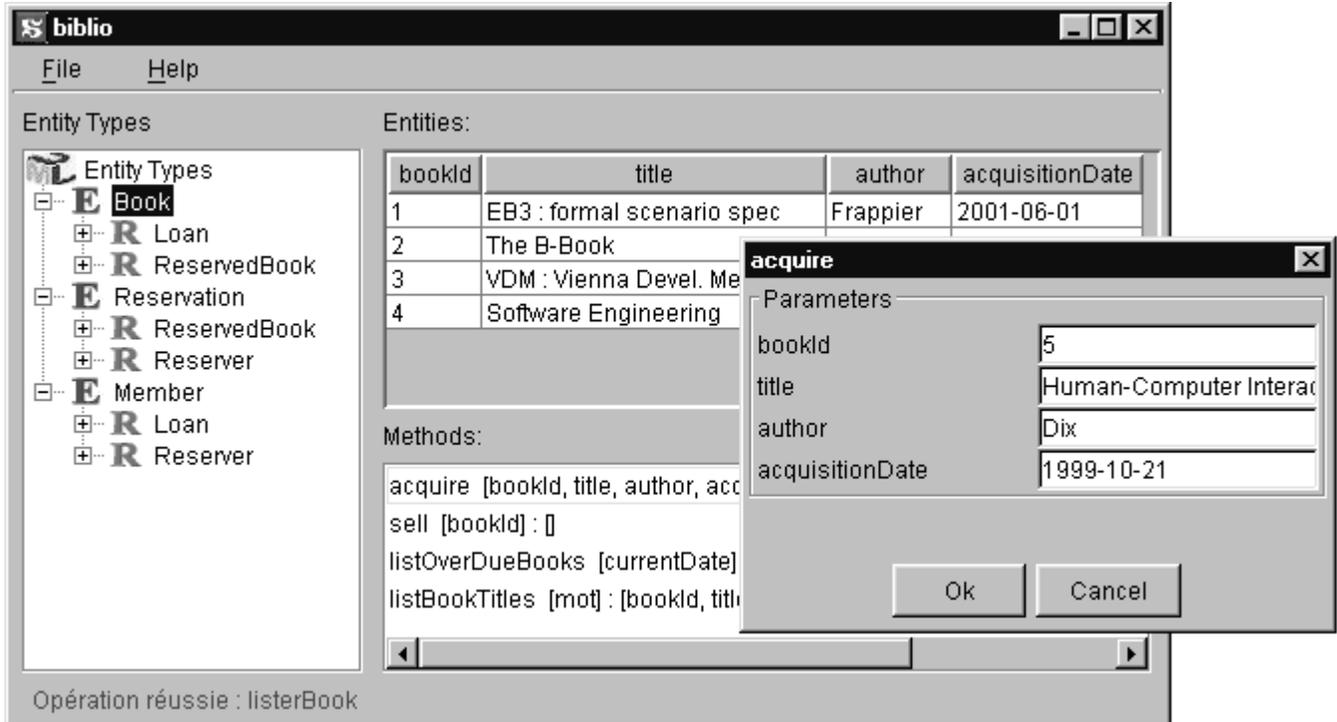


Figure 4: A prototype of a simple GUI generated from a business model

a business model of the EB³ spec, without exploiting all its richness, which further illustrates the feasibility of this approach.

Process Expressions. EB³ uses a process expression to define the valid sequences of inputs of the IS. When a user submits a transaction, the interpreter validates it by checking if the process expression can execute it. The semantics of process expressions is defined by transition rules. The interpreter must implement these transition rules efficiently in order to handle a large volume of data. The most difficult rules to implement efficiently are quantified operators like $\|x : T : E(x)$ for some set $T \triangleq \{v_1, \dots, v_n\}$, which interleaves n process expressions of the form $E(v_i)$. In IS specifications, a quantified interleave expression may contain thousands of operands (usually one for each entity of an entity type). We have to define an indexing mechanism of these operands based on a static flow analysis of the process expressions. When an input $i(\dots, x, \dots)$ is received, this indexing mechanism would quickly fetch the operand which may execute this input.

We have developed an interpreter which can execute process expressions by applying transition rules. The transition rules define a labeled transition system (LTS), where nodes are process expressions, represented as abstract syntax trees, and transitions (edges) are labeled by external events. Because an LTS is possibly infinite, the interpreter cannot pre-compute it. Hence, transitions are computed on the fly, and only the current node is stored. Process expressions may be non-deterministic. Because process expressions are used to specify the valid input traces of the IS, the interpreter must manage nondeterminacy in order to accept any trace that the process expression may accept. Nondeterminacy may lead to deadlocks if the traditional transition rules (e.g., LOTOS or CCS) are applied. For instance, process $a \cdot b \mid a \cdot c$ may accept a and then refuse c if the leftmost operand of the choice was selected at execution time. To avoid this potential deadlock at execution time, the interpreter explores all possible transitions and returns a choice between resulting process expressions when there is nondeterminacy (e.g., $b \mid c$). This solution avoids backtracking in the LTS, which is not acceptable in terms of computational and space complexity when large IS specifications are interpreted.

Process expressions must be stored in order to support persistent data access. An object-oriented database like ObjectStore will be used to store process expressions. Object Store provides concurrency control mechanisms for

multi-user access to its database.

Attribute Definitions. In EB^3 , an entity attribute is defined as a recursive function $f(s, k)$ where s is the system input trace and k is a key value of an entity (e.g., function $nbLoans$ in Figure 1). Recursion occurs on the front of s , that is, the value of $f(i_1 \dots i_{n-1} i_n, k)$ is determined from the value of $f(i_1 \dots i_{n-1}, k)$ or some other attribute. Such recursive functions can be computed efficiently by storing the value of $f(s, k)$ in a database. When a new input i is received, the value of $f(s \cdot i, k)$ is computed by reading the value of $f(s, k)$ from the database; the new value of the attribute is stored back to the database. Hence, even if EB^3 is trace based, there is no need to save the system trace in the interpreter, which would not be acceptable for an IS.

4 Conclusion

The APIS project aims at automating information systems development by generating an implementation from formal, abstract models. The implementation will be based on an interpreter and generated code in Java. The approach is domain specific and relies on process algebra and recursive trace-based definitions of data attributes which can be efficiently computed.

References

- [1] Behm, P., P. Benoit, A. Faivre, J.-M. Meynadier: Météor: A Successful Application of B in a Large Project. In *FM99: World Congress on Formal Methods*, Toulouse, France, Springer-Verlag, LNCS 1709, September 1999, 369–387.
- [2] Blaine, L., L.-M. Gilham, J. Liu, D.R. Smith, S. Westfold: Planware – Domain-Specific Synthesis of High-Performance Schedulers. In *Proceedings of the Thirteenth Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, October 1998, 270–280.
- [3] Stickel, M., R. Waldinger, M. Lowry, T. Pressburger, I. Underwood: Deductive Composition of Astronomical Software from Subroutine Libraries. In *Automated Deduction*, A. Bundy, ed., Springer-Verlag LNCS 814, 1994.
- [4] Foley, J., W.C. Kim, S. Kovacevic, K. Murray: UIIDE-An Intelligent User Interface Design Environment, in *Intelligent User Interfaces*, J. Sullivan and S. Tyler, Eds, Addison-Wesley, 1991, 339–384.
- [5] Frappier, M., R. St-Denis: Combining JSD and Cleanroom for Object-Oriented Scenario Specification. in *Object-Oriented Behavioral Specifications*, H. Kilov, B. Rumpe, I. Simmonds, eds., Kluwer Academic Publishers, 1999.
- [6] Laleau, R. Mammari, A.: A Generic Process to Refine a B Specification into a Relational Database Implementation. In *ZB 2000: Formal Specification and Development in Z and B*, First International Conference of B and Z Users, York, UK, Springer-Verlag, LNCS 1393, September 2000, 22–41.
- [7] Mäkinen, E., Systä, T.: An Interactive approach for synthesizing UML statechart diagrams from Sequence Diagrams. In *OOPSLA Workshop on Scenario-Based Round-Trip Engineering*, Minneapolis, Minnesota USA, October 16, 2000.
- [8] Puerta, A.R.: A Model-Based Interface Development Environment. *IEEE Software*, **14**(4), July/August 1997, 41–47.
- [9] Puerta, A.R. J. Eisenstein: Towards a General Computational Framework for Model-Based Interface Development Systems. In *IUI'99: ACM International Conference on Intelligent User Interfaces*, Los Angeles, January 1999, 171–178.
- [10] Richard, M.: Intégration de la méthode EB^3 dans un processus logiciel orienté objets. M.Sc. thesis, Département de mathématiques et d'informatique, Université de Sherbrooke, 2001.
- [11] 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.
- [12] Whittle, J. and Schumann, J.: Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.