Static Verification of Dynamical Integrity Constraints: a Semantics Based Approach

Véronique Benzaken, Serenella Cerrito, Sébastien Praud
L.R.I., Bât 490, Université de Paris-Sud,
91405 Orsay Cedex, FRANCE
e-mail: benzaken, serena, praud@lri.fr

ABSTRACT. *We address the problem of statically proving, at compilation time, if the execution of a given transaction* m *is safe w.r.t. a dynamical integrity constraint in the framework of object oriented database systems. To do so, we formalize dynamical constraints with first order Linear Temporal Logic formulae and we use predicate transformer "à la Dijkstra", combined with automatic demonstration.*

KEY WORDS : *integrity constraints, static analysis, temporal logic, predicate transformers, automatic demonstration.*

1. Introduction

In this paper we address the problem of statically proving transaction safety with respect to dynamic (also called temporal) integrity constraints, in the framework of an object oriented database.

An important aspect of integrity management is to provide the database system with a tool that will guarantee that the database history (*i.e* the sequence of database states obtained by a sequence of updates) satisfies the integrity constraints (either static or dynamic). The specificity of dynamic constraints is that they express relations between different databases states in database's history.

Most of the time, the verification of constraint satisfaction is performed dynamically, each time an update occurs. Such verifications can be very costly and can severely affect the system performance. Paradoxically, the worst case corresponds to a situation where updates are *a priori* safe because the application programmer has included some tests in the program (defensive programming): in this case, the compiler will generate coherence verification checking algorithms which will be *redundant* with respect to the precautions already taken by the programmer.

However, to guarantee that the database keeps coherent is essential. Therefore, it would be interesting to reduce as much as possible the number of coherence verifications to be performed. Hence, the problem we address is the following: given a set of dynamic integrity constraints and given an update program, is it possible to guarantee statically, *i.e* at compilation time, that the histories produced by such a program will satisfy the constraints, so as to avoid systematic coherence verifications after each update execution ?

We place ourselves in a context where dynamic constraints are expressed in first order Linear Temporal Logic (LTL) and where updates are expressed in an object-oriented database programming language by means of methods (a language *à la* O_2).

Our approach, which extends the works presented in [BS98] to the case of dynamic constraints, undertakes a very detailed analysis of methods and provides some precise information on the impact they have upon constraints. In this sense it is a semantics-based approach. Partial but reliable information concerning methods is obtained by means of predicate transformers [Dij76, DS90]. A predicate transformer is a function that takes as arguments a formula and a program (in our case, a method), and yields another formula. We define two kinds of predicate transformers:

— A forward predicate transformer that, given a method m and a constraint C satisfied by the input data of m, returns a formula $\mathcal{POST}(m, C)$ that is necessarily satisfied by the output data of m. The formula $\mathcal{POST}(m, C)$ may be seen as a post-condition of the execution of m.

— A backward predicate transformer that, given a method m and a constraint C, returns a sufficient condition on the input data of m for C to be satisfied by the output data. This formula is noted $\mathcal{PRE}(m, C)$ and can be seen as a pre-condition to the execution of m.

We prove the soundness of the $\mathcal{POST}$ and $\mathcal{PRE}$ transformers, that is the fact that $\mathcal{POST}(m, C)$ and $\mathcal{PRE}(m, C)$ does indeed compute, respectively, a post-condition and a pre-condition of the execution of m. Once such a result is proved, the key idea of our approach is to use the predicate transformers, combined with first-order temporal theorem proving, to detect at compile-time that a method m is "safe" w.r.t. C, *i.e* it will never violate a constraint C. Using the forward (resp., backward) predicate transformer, if we manage to prove that $\mathcal{POST}(m, C)$ logically implies C (resp., that C logically implies $\mathcal{PRE}(m, C)$) then method m will preserve C. To prove any of the two implications, we extend classical theorem proving techniques to first-order temporal logic theorem proving.

More precisely, we use two different notions: (usual) validity and k-validity. The former notion allows to conclude that a method m is safe w.r.t. constraint C : we will show that is necessarily the case if $\mathcal{POST}(m, C) \rightarrow C$ (resp., $C \rightarrow \mathcal{PRE}(m, C)$) is a valid temporal formula. Now, the prover might fail to find a proof of the validity of the given formula, even if the formula is indeed valid [1]. In this case, the weaker notion of k-validity, allowing to express that the implication is true for all the database histories whose length is bounded by a positive integer k, plays a role. In fact, if we have a proof of the k-validity of the considered implication, we have a still useful

1. First-order linear temporal logic is intrinsically incomplete: see Section 6

information on method m: the method will preserve the integrity constraint C up to a given fixed number of updates. Our twofold approach reduces significantly the number of integrity constraints that have to be checked at run-time.

The paper is organized as follows. In Section 2 we present our framework. In section 3 we recall the syntax and the semantics of first-order Linear Temporal Logic and we describe how we use it to express dynamic integrity constraints and to model databases evolving trough time. In Section 4 we present a decomposition of both methods and constraints and we explain why such a decomposition is necessary. In Section 5 the predicate transformers are defined and are proved to be sound; moreover, we show how to apply them to the problem of static verification of dynamic constraints. In Section 6 we briefly present the principles of our theorem prover and we come back on the notion of k-validity. Finally, we compare our work to related ones and we conclude in Section 7.

2. Framework

We place ourselves in the context of an object-oriented language handling integrity constraints as well as temporal queries *à la TSQL2* ([BCST96]). We have usual classes, methods and a specific language for constraints. We present all these components through examples in the next sections.

2.1. *Classes and persistent roots*

An example of database schema is given in Figure 1.

```
Class Person type tuple(
       name: string,
       spouse: Person,
       money: integer)
method marry(p:Person);
end;
Class Prof inherit Person type tuple(
       student: set(Person))
Class PhD-student inherit Person type tuple(
       advisor: Prof,
            note:integer)
       setadvisor(p:Prof);
       fire;
end;
name Persons: set(Person);
name Profs: set(Prof);
name PhD-students: set (PhD-student);
```

Figure 1. *A schema*

Here, three classes are defined: `Person, Prof` and `PhD-student`, with tuple-structured types. A person has a `name`, a `spouse` and some `money`. Classes `Prof` and `PhD-student` inherit from `Person`. Thus `Person`'s attributes and methods are inherited in classes `Prof` and `PhD-student`. An attribute may be either *mono-valued* or *multi-valued*. If an attribute A is mono-valued, then, for a given object o, A associates an unique value to $o.A$. If an attribute A is multi-valued, then A may associate several distinct values to $o.A$ (in other words, A may associate a set of values to $o.A$, as it is the case, for instance, for the attribute `student` of the class `Prof` (see Figure 1)). Methods can be redefined in other classes and such redefinitions obey to the covariant rules, as defined in [BC96, Cas96]. The semantics of inheritance is the usual inclusion semantics. Persistence is achieved by reachability from persistent names (`Persons, Profs` and `PhD-students`). This means that only objects reachable from those names are stored.

2.2. *Integrity constraints*

Integrity constraints can be of two kinds: static or dynamic. A static constraint is usually represented by a closed formula of first order classical logic. We represent a dynamic constraint by a closed formula of first order Linear Temporal Logic. In this paper, we treat, for instance, dynamic constraints of the following form:

If A is true now then sometimes in the past B has been true
or
If A is true now then B has always been true in the past

Figure 2 gives some examples of dynamic constraints and a first sketch of their formalization in temporal logic.

$C1$: If a PhD-student is advised by Prof B then sometimes in the past he (or she) has been a student of Prof B.
$C1$:`forall` x `in PhD-students:`x`.advisor=y`$\rightarrow \overleftarrow{\Diamond}(x \in y$`.student)`
$C2$: If a Person is married then sometimes in the past he (or she) has been single.
$C2$`: forall` x `in Persons:` x`.spouse`$\neq$`nil`$\rightarrow \overleftarrow{\Diamond}(x$`.spouse=nil)`
$C3$: If a PhD-student has obtained at least twice the note 0, then he (or she) has no longer any advisor.
$C3$`: forall` x `in PhD-students:` $\overleftarrow{\Diamond}(x.\texttt{note=0} \wedge \overleftarrow{\Diamond} x.\texttt{note=0})$
$\rightarrow x.\texttt{advisor=}nil$

Figure 2. *Dynamic constraints*

Intuitively, $\overleftarrow{\Diamond}A$ and $\overleftarrow{\Box}A$ mean, respectively: A has been true sometimes in the past and A has been always true in the past. The precise semantics of the temporal operators $\overleftarrow{\Diamond}$ and $\overleftarrow{\Box}$ is given in Section 3.

2.3. *Methods*

In our framework, updates are expressed via methods. Usually precautions are taken by the application programmer to guarantee that updates lead to consistent states (defensive programming). Some examples of methods are given in Figure 3.

```
method setadvisor(b:Prof) in class PhD-student{
        if ⟐ self ∈ b.student)
        self.advisor:=b;}
method fire in class PhD-student{
        if (⟐(self.note=0 ∧⟐self.note=0))
        advisor:=nil;}
method marry(p:Person) in class Person {
        if (⟐ self.spouse = nil)
        self.spouse:=p;}
```

Figure 3. *Methods and temporal conditions*

Each one of these methods is safe with respect to at least one of the constraints in Figure 2. Run-time tests that are necessary to write safe methods (the conditions of the `if`) involve the use of *temporal boolean queries*. Boolean queries are temporal expressions (see for example [BCST96]) in which the use of quantifiers is allowed. For these methods, if no static checking is performed, the compiler will generate checking algorithms which will be redundant with the precautions taken by the programmer. Our approach allows to avoid such redundancies by proving at compilation time that such methods preserve integrity. The syntax of the methods in the considered update language is defined in Section 4.

3. LTL^{fin} Logic: its syntax, its semantics and its use to model evolving databases

3.1. *Syntax and Semantics*

The formal language that we use to express dynamic temporal constraints is the language of first-order Linear Temporal Logic $(LTL)^2$ (with equality), an instance of modal logics. Several versions of Linear Temporal Logic exist. First of all, different sets of temporal operators may be considered. Secondly, the underlying model of time may be either the sequence of natural numbers $\mathbb{N}$, as, for instance, in [SC85] and [AM90], so that time is seen as bounded in the past, but infinite in the future; or else any finite initial sequence of $\mathbb{N}$, as in [CCM98, CCMP99]: in this case, time is seen as finite also w.r.t. future, although we do not know how far is its "final instant". In the

2. "Linear" means just that time points are linearly ordered.

following, we use the expressions "infinite time semantics" and "finite time semantics" to indicate, respectively, the first and the second model of time.

In this paper, we consider a version with finite time, noted LTL^{fin} (as in [CCMP99]), since finite time sematics seems most suitable to the intended database application; such a choice is rather standard in a database setting (see, for instance [AHVdB96, BD98]). The language of the fragment of linear temporal logic that we consider extends classical first order predicate calculus, containing the connectives $\neg, \wedge, \vee$, and $\top$ (true)[3], and the first-order quantifiers $\forall$ and $\exists$, by means of the unary past time operators $\overleftarrow{\Box}$ (always in the past) and $\overleftarrow{\Diamond}$ (sometimes in the past). Thus, we limit ourselves to "backwards" reasoning; this provides an intuitive motivation of our choice of finite time semantics: at any given time point, only the time interval going from the current time point back to the initial one may be observed.

The signature of the considered logical language contains a set of predicate symbols P and a set of function symbols F. The set P contains, in particular, the binary predicate $=$, whose fixed interpretation is equality. As usual in modal logics, function symbols can be of two kinds: rigid ones (intuitively: their meaning is fixed over time) and non-rigid (their meaning may vary over time). So, let us say that $F = F_{rigid} \cup F_{non-rigid}$.

Intuitively, a temporal interpretation is a finite sequence of classical interpretations $\mathcal{M}_0, \mathcal{M}_1, \cdots, \mathcal{M}_k$, where $0, 1, \cdots, k$ are time points. Formally:

Definition 1 (Interpretation) *A temporal frame T is an initial segment of the natural numbers: $\langle 0, \cdots, k\rangle$; its elements are called time points.*
A temporal interpretation $\mathcal{M}$ is a 5-tuple $\langle T, D, \delta_P, \delta_{Fr}, \delta_{Fnr}\rangle$ where:

— T is temporal frame;

— D is a non empty set (the domain, possibly infinite)

— δ_P is a mapping from $P \times T$ to the set of relations over D such that, for each n-ary predicate symbolp, where $n > 0$, and for each $i \in T$, $\delta_P(p, i)$ is a subset of D^n;

— δ_{Fr} is a mapping from F_{rigid} to the set of (total) functions on D such that, for any n-ary function symbol f with $n \geq 0$, $\delta_{Fr}(f)$ is a function from D^n to D;

— δ_{Fnr} is a mapping from $F_{non-rigid} \times T$ to the set of (total) functions on D such that, for any n-ary function symbol f with $n \geq 0$, for any $i \in T$, $\delta_{Fnr}(f, i)$ is is a function from D^n to D.

As usual, a variable assignment σ is a a function from the set of variables Var to the domain D and a x-variant of σ is a variable assignment σ' such that, for any variable y other than x, $\sigma(y) = \sigma'(y)$. Given any term t, and any time point i, the denotation of t w.r.t. the interpretation $\mathcal{M}$, the variable assignment σ and the time point i, is noted $[t]_{\mathcal{M},\sigma,i}$ and is defined as in the classical case, since $\mathcal{M}$ may be seen as a finite sequence $\mathcal{M}_0, \cdots, \mathcal{M}_k$ of classical interpretations.

---

3. The connective $\rightarrow$ is taken to be a defined connective.

Given any formula F, the satisfiability relation $\mathcal{M}_{\sigma,i} \models F$ (to be read: F is satisfied by $\mathcal{M}$ w.r.t. the variable assignment σ at the time point i) extends the classical definition as follows:

1. $\mathcal{M}_{\sigma,i} \models \top$
2. $\mathcal{M}_{\sigma,i} \models p(t_1, .., t_n)$ iff $\langle [t_1]_{M,\sigma,i} \cdots, [t_n]_{M,\sigma,i} \rangle \in \delta_P(p)$, for $p \in P$.
3. $\mathcal{M}_{\sigma,i} \models t_1 = t_2$ iff $[t_1]_{M,\sigma,i} = [t_2]_{M,\sigma,i}$
4. $\mathcal{M}_{\sigma,i} \models \neg A$ iff $\mathcal{M}_{\sigma,i} \not\models A$
5. $\mathcal{M}_{\sigma,i} \models A \wedge B$ iff $\mathcal{M}_{\sigma,i} \models A$ and $\mathcal{M}_{\sigma,i} \models B$.
6. $\mathcal{M}_{\sigma,i} \models A \vee B$ iff either $\mathcal{M}_{\sigma,i} \models A$ or $\mathcal{M}_{\sigma,i} \models B$.
7. $\mathcal{M}_{\sigma,i} \models \forall x A(x)$ iff for any x-variant σ' of σ, $\mathcal{M}_{\sigma',i} \models A(x)$
8. $\mathcal{M}_{\sigma,i} \models \exists x A(x)$ iff there exists a x-variant σ' of σ such that $\mathcal{M}_{\sigma',i} \models A(x)$
9. $\mathcal{M}_{\sigma,i} \models \overleftarrow{\Box} A$ iff for all $j < i$, $\mathcal{M}_{\sigma,j} \models A$.
10. $\mathcal{M}_{\sigma,i} \models \overleftarrow{\Diamond} A$ iff there exists a $j \in T$ such that $j < i$ and $\mathcal{M}_{\sigma,j} \models A$.

As usual, when A is closed, σ may be omitted and we write just $\mathcal{M}_j \models A$ to mean that A is satisfied by the temporal interpretation $\mathcal{M}$ at the time point j; we say also that A *holds* at the time point i of $\mathcal{M}$.

Note that the semantics for the operators $\overleftarrow{\Box}$ and $\overleftarrow{\Diamond}$ is "strict": the present time point is excluded. Thus, $\overleftarrow{\Box} A$ holds at time point i iff A holds at all time points strictly preceding i and $\overleftarrow{\Diamond} A$ holds at i iff A holds at some time point strictly preceding i.Note also the following trivial consequence of the semantical definition of $\overleftarrow{\Box}$ and $\overleftarrow{\Diamond}$: $\mathcal{M}_0 \models \overleftarrow{\Box} A$ and $\mathcal{M}_0 \not\models \overleftarrow{\Diamond} A$, for any formula A. The operators corresponding to the "large" semantics of $\overleftarrow{\Box}$ and $\overleftarrow{\Diamond}$, *i.e* the semantics including the present time point, are noted, respectively, $\overleftarrow{\Box}_l$ and $\overleftarrow{\Diamond}_l$ and are defined in terms of the previous ones:

$$\overleftarrow{\Box}_l A =_{def} A \wedge \overleftarrow{\Box} A \qquad \overleftarrow{\Diamond}_l A =_{def} A \vee \overleftarrow{\Diamond} A$$

Definition 2 (Validity) *A closed formula A is valid iff A holds in any time point of any temporal interpretation $\mathcal{M}$.*

In the sequel, we make also use of a notion weaker than validity:

Definition 3 (k-Validity) *Let k be any natural number. A closed formula A is k-valid iff A holds in any time point of any temporal interpretation $\mathcal{M}$ having a temporal frame $T = \langle 0, \cdots, m \rangle$ where $m \leq k$.*

Thus, if A is k-valid, A might fail to hold in some time point of some interpretation having more than $k + 1$ time points. For instance, the formulae $\overleftarrow{\Box}\overleftarrow{\Box}\overleftarrow{\Box}\neg T$ and $(\overleftarrow{\Box}\overleftarrow{\Box}\overleftarrow{\Box}\neg T) \vee \forall x p(x)$ are 2-valid but they are not 3-valid.

A formula A is in *negation normal form (nnf)* when its only subformulae of the form $\neg B$ are such that B is atomic. Since any LTL^{fin} formula is logically equivalent to a formula in nnf, in the following we suppose, without loss of generality, that formulae are in nnf.

3.2. *Modeling evolving databases*

In Definition 1 we have made a technical choice on the domain where variables range, which needs to be motivated. In the classical literature on first order modal logics there are several alternatives w.r.t. domain definition: either the domain does not change over time, *i.e* it is the same for each $\mathcal{M}_i$ and $\mathcal{M}_j$, or else it can vary when time goes on, *i.e* $\mathcal{M}_i$'s domain may be different from $\mathcal{M}_j$'s domain when $i < j$. Here we have chosen the first alternative. Intuitively, the unique domain D models the set of the (potential) object identifiers in the database[4]. Such identifiers may or may not correspond to objects actually created in the database. Hence, at any given time point, the active domain of the current database is a finite subset of D. Moreover, in our formalization of a database history via a temporal interpretation, D is a possibly infinite set. The intuition behind such a choice is that we do not want to fix in advance an upper bound on the number of distinct objects that can be created in the base through time, even if the update language considered in our paper does not contain yet object-creation instructions (see Definition 4).

An LTL^{fin} expression saying that an object x (actually created in the database) belongs to a class $\mathtt{c}$ (see for instance x in Persons in Figure 2) is: $c(x)$, where c is a monadic predicate symbol. An LTL^{fin} expression saying that the value of a mono-valued attribute $\mathtt{a}$ of the object x is y is: $a(x) = y$, where a is a monadic function symbol. However, in the following sections we often make an abuse of language and we write, for instance, $x.\mathtt{spouse}$ instead of the LTL^{fin} term: $spouse(x)$. An LTL^{fin} expression saying that a value of a multi-valued attribute $\mathtt{a}$ of the object x is y is: $\mathtt{a}(x, y)$, where $\mathtt{a}$ is a binary predicate. However, by an abuse of language, we sometimes write $y \in \mathtt{a}(x)$.

Let B be an instance of a database evolving through time and let $B_0, \cdots, B_n$ be its history up to current time n, *i.e* the sequence of database states until the current one.That is, B_0 is the initial database, for any i such that $1 \leq i \leq n-1$, B_{i+1} is the database state resulting from an update on B_i and B_n is the current state. Such an history may be seen as a temporal interpretation (as, for instance, in [AHVdB96, BD98]). More precisely, if we identify any base B_i with a classical interpretation (which is quite standard), the database history $B_0, \cdots, B_n$ can be seen as the temporal interpretation $\mathcal{M}_B$ where the temporal frame is $\langle 0, \cdots, n \rangle$ and, for any closed formula A, for any $i \in \langle 0, \cdots, n \rangle$, $\mathcal{M}_{B_i} \models A$ if and only if A is true w.r.t. the base B_i.

Given any closed LTL^{fin} formula A we say that the history $B_0, \cdots, B_n$ satisfies A if and only if, for each i such that $1 \leq i \leq n$, $\mathcal{M}_{B_i} \models A$. In the following, we just write $B_i \models A$ to mean that $\mathcal{M}_{B_i} \models A$.

In our approach, a dynamic constraint is expressed by a closed LTL^{fin} formula. For instance, a constraint of the form:

If A is true now then B has been sometimes true in the past

4. Actually D contains also integers, since we need to express informations like “Any person’s money is greater than 1000”. Thus, to be precise, one should consider a many sorted language. However we make abstraction of such purely technical details in the paper.

is formalized as

$$\forall \overrightarrow{x}(A(\overrightarrow{x}) \rightarrow \overleftarrow{\Diamond}_* B(\overrightarrow{x}))$$

and a constraint of the form:

If A is true now then B has always been true in the past

is formalized as:

$$\forall \overrightarrow{x}(A(\overrightarrow{x}) \rightarrow \overleftarrow{\Box}_* B(\overrightarrow{x}))$$

where $\overrightarrow{x}$ denotes a variable vector $x_1, \cdots, x_n$ and $\overleftarrow{\Diamond}_*$ (resp. $\overleftarrow{\Box}_*$) represents $\overleftarrow{\Diamond}$ or $\overleftarrow{\Diamond}_l$ (resp. $\overleftarrow{\Box}$ or $\overleftarrow{\Box}_l$).

Let us note that a formula having the form $\forall \overrightarrow{x}(A(\overrightarrow{x}) \rightarrow \overrightarrow{Op}\ B(\overrightarrow{x}))$ where:

1. $\overrightarrow{Op} \in \{\overleftarrow{\Diamond}, \overleftarrow{\Box}, \overleftarrow{\Diamond}_l, \overleftarrow{\Box}_l\}$
2. A and B have the same free variables $\overrightarrow{x}$
3. A and B are classical (safe) first order formulae

corresponds to the case of Dynamic Algebraic Dependencies [BD95, BD98]. However, also other kinds of constraints can be expressed in our framework; for instance, the constraint

If a PhD-student has obtained at least twice the note 0 then he (or she) has no longer any advisor

may be be formalized in LTL^{fin} as:

$$\forall x\ [(PhD - student(x) \land \overleftarrow{\Diamond}(note(x) = 0 \land \overleftarrow{\Diamond} note(x) = 0)) \rightarrow advisor(x) = nil]$$

4. Decomposition of methods and formulae

The task of analyzing computer languages can be difficult, particularly when dealing with object-oriented programming languages. Indeed, in such languages, objects that are involved in an update are reached in a *navigational* way. Defining a predicate transformer directly for our language can be very difficult and proving it is correctly defined can be even harder. So, to bypass this difficulty, we first translate the source language into an intermediate form and then define the predicate transformers on that intermediate form.

Let us see, for instance, a simple assignment instruction:

`p.spouse.spouse:=q.spouse` (1)

This update is not about `p` or `q` but about objects that can be reached from `p` and `q`. Such an instruction can be decomposed in two simpler ones, as follows. First, the objects concerned by the update are determined by navigating along the paths defined by the attributes. In our case, these objects are `p.spouse` and `q.spouse`. Let us call them respectively o_1 and o_2. Then, the update o_1`.spouse:=`o_2 is performed.

This decomposition has to be done formally and systematically. So, we will use the following new instruction:

`forone` *object* `where` *condition* `do` *instruction*

This instruction executes *instruction*, where *object* is the only object that satisfies *condition*[5]. Instruction (1) is translated as:

```
forone o1 where o1 = p.spouse do
        forone o2 where o2 = q.spouse do
                o1.spouse:=o2
```

This translation simplifies the definition of the predicate transformers for assignment instructions. We just need to define how to deal with the instruction `forone` and only consider assignments of the form:

`parameter.attribute:=parameter`.

We give in the following definition the syntax of the whole intermediate language we shall use.

Definition 4 (Method) *A method is a program* $m(p_1, \cdots, p_n)$=`instruction` *where* m *is the method's name,* $p_1, \cdots, p_n$ *are its parameters and* `instruction` *is the body of* m*, defined by the following grammar:*

```
instruction::= parameter.attribute:=parameter
    | instruction;instruction
    | {instruction}
    | if condition then instruction
    | forall parameter where condition
            do instruction
    | forone parameter where condition
            do instruction
```

An instruction of the form: `parameter.attribute:=parameter` is said to be an *assignment instruction*, and it has the obvious meaning. The ";" is a *sequence* operator, so that the compounded instruction $i_1; i_2$ is performed by executing first i_1, then i_2. The characters { and } mark, respectively, the begin and the end of an instruction. The instruction "`if`" is a conditional instruction, that is `instruction` is performed when `condition` is true. Note that a conditional instruction may contain a test corresponding to a temporal boolean query; for instance, it might be: `if` $\overleftarrow{\Diamond}$`self.a=p then self.b=q`. The semantics of the "`forall`" is: for each `parameter` satisfying the `condition` execute `instruction`.

To clarify the semantics of methods, note that we take into account two sorts of "times", a time "global" to the database history and a time "local" to method calls. Given a database history $B_0, B_1, \cdots, B_k$ such that B_0 is the initial database and each B_{i+1} is obtained by an update of B_i via a method call m_i, the global time of the history is the sequence of instants $0, 1, \cdots, k$. Only the complete execution of a method makes the "clock" measuring global time progress. Temporal operators appearing in a boolean query occurring in the body of a method m_i make reference to such a global time. Thus, for instance, let the body of a method m_i be {`self.a = p; if` $\overleftarrow{\Diamond}$`self.a=p then self.a=q`}, with `p` $\neq$ `q`. Suppose that the method is applied to update B_i and that the segment of history $B_0, \cdots, B_i$ is such that, for any

5. The existence and unicity of such an object is guaranteed by static type checking and the semantics of our language.

$j \in \{0,..,i\}$, the value of `self.a` is different from `p`. Then the boolean test in the conditional instruction fails, and in the database B_{i+1} obtained applying m_i to B_i the value of `self.a` is p.

On the other hand, the order in which instructions in the body of a method are executed matters, which implies that when executing a complex method a a sort of *local time* is taken into account. For instance, consider the following methods m_1 and m_2, whose respective bodies are (where `p`$\neq$`r`):

```
m1: {self.a:=r;
        self.a:=p;
        if self.a=p then self.b:=q}
m2: {self.a:=p;
        self.a:=r;
        if self.a=p then self.b:=q}
```

After the execution of m_1 (resp. m_2) the value of `self.a` is `q` (resp. `r`). In a sense, we consider that when executing an instruction the "present time" is the time after the execution on the precedent instruction.

Similary to the forone-decomposition, and in order to avoid nested function symbols, formulae are previously decomposed as follows: for instance, we write

$$\forall x \exists y (x.\texttt{spouse} = y \wedge y.\texttt{spouse} = x)$$

rather than $\forall x(x.\texttt{spouse}.\texttt{spouse} = x)$.

In the next section we define our predicate transformers, which are functions whose arguments are a method and a formula and whose result is a formula.

5. Temporal predicate transformers

Our approach is to check in a static way (at compilation time) whether a method m preserves a constraint C. More precisely, given a database history $B_0, B_1, \cdots, B_n$ satisfying an integrity constraint C and a method m such that B_{n+1} is the result of the execution of m on B_n, we would like to be able to establish *before* the execution of m that $B_0, B_1, \cdots, B_n, B_{n+1}$ satisfies C.

To that purpose, in Section 5.1 we define a forward predicate transformer $\mathcal{POST}$, which is a function taking as an input a method m and a constraint C and yielding a formula $\mathcal{POST}(m, C)$. We prove that our definition of $\mathcal{POST}$ is sound, then we show that the validity of the LTL^{fin} formula $\mathcal{POST}(m, C) \rightarrow C$ is a sufficient condition for the preservation of the constraint C by m. An analogous result is also proved for the case where $\mathcal{POST}(m, C) \rightarrow C$ is k-valid.

Similarly, in Section 5.2 we define a backward predicate transformer $\mathcal{PRE}$, which is a function taking as an input a method m and a constraint C and yielding a formula $\mathcal{PRE}(m, C)$. We prove that our definition of $\mathcal{PRE}$ is sound, then we prove that the validity of the formula of LTL^{fin}: $C \rightarrow \mathcal{PRE}(m, C)$ is a sufficient condition for the preservation of the constraint C by m and we establish an analogous result for the k-validity case.

In Sections 5.1 and 5.2 the transformers are defined for the fragment of the update language that does not contain the `forall` instruction, which is delicate to handle. Such definitions are extended to the `forall` case in Section 5.4.

5.1. *A forward temporal predicate transformer*

The transformer is defined by induction on both the structure of the formula ϕ and the update m.

Definition 5 ($\mathcal{POST}$ transformer) *The predicate transformer $\mathcal{POST}$ is a binary function whose arguments are a method m and a LTL^{fin}formula ϕ (in nnf) and whose result is a LTL^{fin} formula $\mathcal{POST}(m, \phi)$ recursively defined as follows:*

— Formula parsing:
1. $\mathcal{POST}(m, \phi \wedge \psi) \equiv \mathcal{POST}(m, \phi) \wedge \mathcal{POST}(m, \psi)$
2. $\mathcal{POST}(m, \phi \vee \psi) \equiv \mathcal{POST}(m, \phi) \vee \mathcal{POST}(m, \psi)$
3. $\mathcal{POST}(m, \forall x \phi(x)) \equiv \forall x \mathcal{POST}(m, \phi(x))$
4. $\mathcal{POST}(m, \exists x \phi(x)) \equiv \exists x \mathcal{POST}(m, \phi(x))$
5. $\mathcal{POST}(m, \overleftarrow{\Diamond}\phi) \equiv \overleftarrow{\Diamond}\phi$
6. $\mathcal{POST}(m, \overleftarrow{\Box}\phi) \equiv \overleftarrow{\Box}\overleftarrow{\Box}\phi$

— Method's body parsing

7. if m$\equiv$ `u.a:=v` then

(a) if $\phi \equiv (x.\mathtt{a}{=}\ y)$ then
$\mathcal{POST}(m, \phi) \equiv (\mathtt{u}{=}\ x \wedge \mathtt{u.a{=}v}) \vee (\mathtt{u} \neq x \wedge \mathtt{u.a{=}v} \wedge x.\mathtt{a}{=}\ y)$

(b) if $\phi \equiv (x.\mathtt{a}{\neq}\ y)$ then
$\mathcal{POST}(m, \phi) \equiv (\mathtt{u}{=}\ x \wedge \mathtt{u.a{=}v}) \vee (\mathtt{u}{\neq}\ x \wedge \mathtt{u.a{=}v} \wedge x.\mathtt{a}{\neq}\ y)$

(c) if ϕ is an other literal
$\mathcal{POST}(m, \phi) \equiv \phi \wedge \mathtt{u.a{=}v}$

8. if $m \equiv i_1; i_2$ then
$\mathcal{POST}(m, \phi) \equiv \mathcal{POST}(i_2, \mathcal{POST}(i_1, \phi))$

9. if $m \equiv \{i\}$ then
$\mathcal{POST}(m, \phi) \equiv \mathcal{POST}(i, \phi)$

10. if ($m \equiv$ `if` ψ `then` i) then
$\mathcal{POST}(m, \phi) \equiv \mathcal{POST}(i, \psi \wedge \phi) \vee (\neg\psi \wedge \phi)$

11. if $m \equiv$ `forone` v `where` $\psi(v)$ `do` i then
$\mathcal{POST}(m, \phi) \equiv\ \exists\, v\, \mathcal{POST}(i, \psi(v) \wedge \phi)$

Lemma 1 *Let $B = B_0, \cdots, B_{old}$ be a database history, m an assignment instruction, B_{new} the base resulting from the execution of m on B_{old} and ϕ any LTL^{fin} formula. If $B_{old} \models \phi$ then $B_{new} \models \mathcal{POST}(m, \phi)$.*

Sketch of proof. By induction on the logical structure of ϕ, using the items 1-7 of $\mathcal{POST}$ definition.

Theorem 1 (Soundness of $\mathcal{POST}$) *Let $B = B_0, \cdots, B_{old}$ be a database history, ϕ any LTL^{fin} formula and m any method. Let B_{new} be the base resulting from the execution of m on B_{old}.*

If $B_{old} \models \phi$ then $B_{new} \models \mathcal{POST}(m, \phi)$.

Sketch of proof. The proof is essentially by induction on the structure of m, using items 8-11 in $\mathcal{POST}$ definition. However, the case where m's body is the sequence of instructions $i_1; i_2$ and ϕ's main logical operator is a temporal operator is delicate. There, a simple call to the inductive hypothesis will not do and what needs to be observed, instead, is that if $\phi = \overleftarrow{\Diamond}\psi$ then $\mathcal{POST}(i_1; i_2, \phi) = \phi$ and if $\phi = \overleftarrow{\Box}\psi$, then $\mathcal{POST}(i_1; i_2, \phi) = \overleftarrow{\Box}\overleftarrow{\Box}\phi$. In both cases, if $B_{old} \models \phi$, then $B_{old+1} \models \mathcal{POST}(i_1; i_2, \phi)$, as an immediate consequence of the semantics of $\overleftarrow{\Diamond}$ and $\overleftarrow{\Box}$.

5.2. *A backward temporal predicate transformer*

Definition 6 ($\mathcal{PRE}$ transformer) *The predicate transformer $\mathcal{PRE}$ is a binary function whose arguments are a method m and a LTL^{fin} formula ϕ (in nnf) and whose value is a LTL^{fin} formula $\mathcal{PRE}(m, \phi)$ inductively defined as follows:*

— Formula parsing:

1. $\mathcal{PRE}(m, \phi \wedge \psi) \equiv \mathcal{PRE}(m, \phi) \wedge \mathcal{PRE}(m, \psi)$
2. $\mathcal{PRE}(m, \phi \vee \psi) \equiv \mathcal{PRE}(m, \phi) \vee \mathcal{PRE}(m, \psi)$
3. $\mathcal{PRE}(m, \forall x \phi(x)) \equiv \forall x \mathcal{PRE}(m, \phi(x))$
4. $\mathcal{PRE}(m, \exists x \phi(x)) \equiv \exists x \mathcal{PRE}(m, \phi(x))$
5. $\mathcal{PRE}(m, \overleftarrow{\Diamond}\phi) \equiv \overleftarrow{\Diamond}_l\, \phi$
6. $\mathcal{PRE}(m, \overleftarrow{\Box}\phi) \equiv \overleftarrow{\Box}_l \phi$

— Method's body parsing

7. if m$\equiv$ `u.a := v` then

(a) if $\phi \equiv (x.\mathtt{a} = y)$ then
$\mathcal{PRE}(m, \phi) \equiv (\mathtt{u} = x \wedge y = \mathtt{v}) \vee (\mathtt{u} \neq x \wedge x.\mathtt{a} = y)$

(b) if $\phi \equiv (x.\mathtt{a} \neq y)$ then
$\mathcal{PRE}(m, \phi) \equiv (\mathtt{u} = x \wedge y \neq \mathtt{v}) \vee (\mathtt{u} \neq x \wedge x.\mathtt{a} \neq y)$

(c) if ϕ is an other littéral
$\mathcal{PRE}(m, \phi) \equiv \phi$

8. if $m \equiv i_1; i_2$ then
$\mathcal{PRE}(m, \phi) \equiv \mathcal{PRE}(i_1, \mathcal{PRE}(i_2, \phi))$

9. if $m \equiv \{i\}$ then
$\mathcal{PRE}(m, \phi) \equiv \mathcal{PRE}(i, \phi)$

10. if $m \equiv$ "`if` ψ `then` i" then
$\mathcal{PRE}(m, \phi) \equiv (\mathcal{PRE}(i, \phi) \wedge \psi) \vee (\neg\psi \wedge \phi)$

11. if $m \equiv$ `forone` v `where` $\psi(v)$ `do` i then
$\mathcal{PRE}(m, \phi) \equiv \exists\, v\, \mathcal{PRE}(i, \phi) \wedge \psi(v)$

Lemma 2 *Let $B = B_0, \cdots, B_{old}$ be a database history, m an assignment instruction, B_{new} the base resulting from the execution of m on B_{old} and ϕ any LTL^{fin} formula. If $B_{old} \models \mathcal{PRE}(m, \phi)$ then $B_{new} \models \phi$*

Sketch of proof. Similar to the proof of the corresponding lemma for $\mathcal{POST}$.

Theorem 2 (Soundness of $\mathcal{PRE}$) *Let $B = B_0, \cdots, B_{old}$ be a database history, ϕ any LTL^{fin} formula and m any method. Let B_{new} be the base resulting from the execution of m on B_{old}.*

$$\textit{If } B_{old} \models \mathcal{PRE}(m, \phi) \textit{ then } B_{new} \models \phi$$

Sketch of proof. Similar to the proof of the corresponding theorem for $\mathcal{POST}$.

5.3. *Using Predicate Transformers*

In the previous sections we have defined a forward and a backward predicate transformer. Here we study the application of these notions in order to statically detect constraint preservation.

Definition 7 (Safe method) *A method m is safe w.r.t. an integrity constraint ϕ if, whenever $B_0, \cdots, B_n$, where $0 \leq n$, is a database history satisfying ϕ and B_{n+1} is obtained by executing m on B_n, the new history $B_0, \cdots, B_n, B_{n+1}$ satisfies ϕ.*

Theorems 1 and 2 allow to get the following result, stating sufficient conditions for methods' safety.

Corollary 1 *Let m be a method and ϕ a constraint. Method m is safe w.r.t. ϕ if:*
(1) $\mathcal{POST}(m, \phi) \rightarrow \phi$ is valid
or if
(2) $\phi \rightarrow \mathcal{PRE}(m, \phi)$ is valid.

Example 1 *Let C be the dynamic constraint: If a person is married then he (or she) has been single, formalized by the LTL^{fin} formula:*
$\forall x\{(person(x) \wedge x.spouse \neq nil) \rightarrow \overleftarrow{\Diamond} x.spouse = nil\}$
Let m be the method: `if` $\overleftarrow{\Diamond}$*self.spouse = nil* `then self.spouse := p`. *By applying $\mathcal{POST}$ to m and C (and simplifying) the formula $\forall x(F_1 \vee F_2 \vee F_3 \vee F_4 \vee F_5 \vee F_6)$ is obtained, where:*

— *F_1 is $(\overleftarrow{\Diamond} self.spouse = nil) \wedge self.spouse = p \wedge \neg person(x)$*
— *F_2 is $(\overleftarrow{\Box} self.spouse \neq nil) \wedge \neg person(x)$*
— *F_3 is $(\overleftarrow{\Diamond} self.spouse = nil) \wedge self = x \wedge self.spouse = p$*
— *F_4 is $(\overleftarrow{\Diamond} self.spouse = nil)$*
$\wedge\, self \neq x \wedge self.spouse = p \wedge x.spouse = nil$

— F_5 *is* $(\overleftarrow{\Box} self.spouse \neq nil) \wedge x.spouse = nil$

— F_6 *is* $(\overleftarrow{\Diamond} x.spouse = nil)$

Formulae F_1 and F_2 correspond to a case where x is not a person, thus x cannot lead to C's violation. Formula F_3 corresponds to the case where x is the object to which m has been applied, x has been single and is presently married; thus, such an x cannot violate C. Formulae F_4 and F_5 imply that x is presently single, hence, again, such an x cannot violate C. Finally, formula F_6 says that x has been single, so that x cannot violate C. It easy to verify that if $\mathcal{B}_{old} \models C$ then $B_{new} \models \forall x(F_1 \vee F_2 \vee F_3 \vee F_4 \vee F_5 \vee F_6)$ (as stated by Theorem 1). Clearly the formula $\forall x(F_1 \vee F_2 \vee F_3 \vee F_4 \vee F_5 \vee F_6) \rightarrow C$ is valid. Thus, as stated by Corollary 1, method m is safe w.r.t. C.

Besides the notion of safe method, we need a weaker notion.

Definition 8 (k-safe method) *Let k be a positive integer. A method m is k-safe w.r.t. a constraint ϕ if, whenever $B_0, \cdots, B_n$ is a database history satisfying ϕ, $0 \leq n < k$ and B_{n+1} is obtained by executing m on B_n, the new history $B_0, \cdots, B_n, B_{n+1}$ satisfies ϕ.*

Theorems 1 and 2 allow to get also the following result:

Corollary 2 *Let m be a method and ϕ a constraint. Method m is k-safe w.r.t. constraint ϕ if:*

(1) $\mathcal{POST}(m, \phi) \rightarrow \phi$ is k-valid

or if

(2) $\phi \rightarrow \mathcal{PRE}(m, \phi)$ is k-valid

Corollary 2 needs some explications. An attempt to automatically prove the validity of the formula $\mathcal{POST}(m, \phi) \rightarrow \phi$ (resp. $\phi \rightarrow \mathcal{PRE}(m, \phi)$) may fail, even in the case where the formula is indeed valid. This is a consequence of the intrinsic incompleteness of the logic LTL^{fin} (see Section 6). The weaker notion of k-validity, allowing one to express that the considered implication holds for any state of any database history whose length is upper bounded by k, can then be used to get interesting informations on m. Indeed, if, at a given time point i smaller than k method m is proved to be k-safe w.r.t. a constraint C, until the instant k there will be no need to verify C's preservation. However, at time point $k + 1$, method m is no longer necessarily safe and it might need to be tested, incrementing the bound k.

Let us observe that, since we are in an object oriented framework, methods can be overloaded. When a method m is redefined with $m_1, \cdots, m_n$ different implementations, we say that a method m is safe (or k-safe) w.r.t. a constraint ϕ if each m_i is safe (or k-safe) w.r.t. ϕ.

5.4. *Methods with loops*

Database programming languages allow the use of a specific loop. In our language, it is defined as:

$i \equiv$ `forall` *variable* `where` *condition* `do` *j*

The semantics of the loop is the parallel-execution one. Intuitively, the idea is that if the instruction *j* is safe w.r.t. a constraint and if the entry for the loop satisfies the constraint, then the loop `forall` will also be safe. Then we extend Definitions 5 and 6 of the predicate transformers as follows (C being the initial constraint that we want to test):

$$\mathcal{POST}(i,\phi) = \begin{cases} C \text{ if } \mathcal{POST}(j,C) \rightarrow C \text{ and } \phi \rightarrow C \text{ are valid} \\ \top \text{ else} \end{cases}$$

$$\mathcal{PRE}(i,\phi) = \begin{cases} C \text{ if } \mathcal{PRE}(j,C) \rightarrow C \text{ and } \phi \rightarrow C \text{ are valid} \\ \top \text{ else} \end{cases}$$

If neither $\mathcal{POST}(j,C) \rightarrow C$ nor $\phi \rightarrow C$ are valid then $\mathcal{POST}(i,\phi)$ takes the value $\top$ (which is trivially satisfied by any history). In order to deal with loops, we have to include in the process of building our predicate transformers some theorem proving steps. The proof of the fact that Theorems 1 and 2 carry over the above extension to loops is closely related to the proof of the analogous result, for the case of the forward classical predicate transformer, that can be found in [Sch97].

6. A theorem prover for LTL^{fin}

In the previous sections, we have shown that the safety of a method m w.r.t. a dynamical constraint C can be assured if a given LTL^{fin} formula, depending on m and C, can be proven to be valid.

Several proof systems have been proposed for variants of linear temporal propositional logic: see, for instance, [Wol85, BFG$^+$89, AM90, SGL97] for the infinite time semantics, and [CCM98] for the finite one. However, very little work has been done on the first-order case. The proof system in [AM90] deals with first-order, but it is incomplete. Moreover, no practical proof search strategy, allowing to build up a theorem prover (not just a proof editor), is indicated.

The fact that first-order LTL^{fin} proof theory has been so poorly explored is hardly surprising. In fact, in [Aba89, Krö90] it is proved that no recursive complete axiomatization can exist for LTL (with the infinite time semantics). In both the cited works the key idea allowing to prove this negative result is the possibility of coding arithmetic in LTL. This kind of technique is not available in the case of LTL^{fin}, because temporal frames are *finite* initial segments of $\mathbb{N}$. However, it can be proved that an analogous incompleteness result holds also in the case of LTL^{fin} and, in particular, for the LTL^{fin} fragment where the temporal operators are the past time modalities $\overleftarrow{\Box}, \overleftarrow{\Diamond}$ and $\overleftarrow{\bigcirc}$ (at previous time point) (see [CCMP99]). The result can even be strengthened so to avoid to require the presence of $\overleftarrow{\bigcirc}$.

The existence of such negative results explains some particularities of the theorem prover that we propose, which is based on the proof system for "full" first order LTL^{fin} defined by Cerrito, Cialdea-Mayer and Praud for LTL^{fin} in [CCMP99]. Such a system, which extends [CCM98] to first order, handles two kinds of expressions:

1. disequations on time points, *i.e* , expressions of the form $s \leq t + n$, where s and t denote time points and n is an integer.

2. labeled formulae, *i.e* , expressions of the form $[s, t]A$, whose intuitive meaning is: the formula A holds in each point of the time interval $[s, t]$.

The calculus takes as input a formula A (to be possibly proved) and a initial set of disequations over time points, which is either a singleton or empty. In the first case, the set contains an initial disequation stating that only temporal frames upper bounded by a given integer k are explored. In the second case, no bound is set on the size of temporal frames. As it is proved in [CCMP99], the calculus has the following properties:

1. When the input contains an initial disequation setting the bound k on the size of temporal frames, the calculus is sound and complete w.r.t. k-validity.

2. When there is no initial disequation, the calculus is sound but (obviously) incomplete w.r.t. validity tout court.

Obviously, even when an initial disequation setting the bound k on temporal frames is set, the calculus may not terminate, since it extends classical first order logic. However, such a proof system can be used as the base for an automatic theorem prover for LTL^{fin}, by setting an upper bound on the size of searched proofs (as it is standard for first order theorem provers). This, of course, leads to a loss of completeness, but, when a proof for A is found, A is indeed k-valid (resp. valid)[6].

We are presently working on the implementation of such a theorem prover.

7. Related work and conclusion

In this paper we study the problem of detecting, at compilation time, dynamic constraints preservation by updates written in a object oriented database programming language. We extend the works done for static constraints in [BS98, Sch97].

In order to achieve the goal of static detection of method safety, we have proposed an approach based on method's semantics, which combines static analysis together with first order theorem proving techniques in temporal logic. We uses Dijkstra's predicate transformer calculus to (partially) capture the semantics of updates. Unfortunately, the exact semantics of updates is intrinsically unreachable for languages that allow aliasing and assignments [Cla79] which is the case for our language. This theoretical limit explains why we only have sufficient conditions rather than a characterisation for method safety. Unlike other approaches, we do not restrict ourselves to a simplified language, but we take into account a rich one, and we handle a large class of dynamic constraints (all those which can be expressed by the considered fragment of LTL^{fin}).

The problem of preservation of *static* constraints has been studied by several authors. In [SS89] an expert system based on Boyer-Moore logic is used to prove that

6. The same approach is taken in [BS98] for the case of static constraints

a transaction is safe. The main problem with that approach is that the consistency of the axioms on which the expert system is founded is not guaranteed (as acknowledged by the authors themselves). Therefore, the system is not formally founded. [Qia90] presents an axiom system for transaction analysis. However, the applicability of such an axiom system is unclear. Actually in [Qia93], the author follows the orthogonal approach of transaction synthesis, and justifies such a change saying that conventional techniques for program analysis do not have the ability to reason about integrity constraints. [Mag97] study analysis of static constraints, in the context of relational databases, and bear some similarities with [BS98, Sch97].

In the case of *dynamic* constraints, most of existing works ([UG87, Cho92] for instance) consider only relational databases and dynamic coherence verification. The work in [BD95, BD98] is, up to our knowledge, the first attempt to statically handle dynamic constraints. Differently from us, the authors choose to perform transaction synthesis, rather than proving that a given transaction is safe. Their results allow to give the application programmer some primitive update programs and some programming rules which are sufficient to write programs capable to generate all (and only) the database histories which are "legal" w.r.t. the constraints. Clearly, the problem that we address is different. Here, we opt for a more "liberal" attitude, and, rather than giving the programmer a set of instructions to be followed, we leave him free to write no matter which program, charging the compiler with the task of verifying if such a program is safe.

Another important question, for the static analysis of constraints, is whether there is a systematic way of automatically correcting faulty versions of methods, to yield safe versions. The answer to this question is positive, and the backward predicate transformer ($\mathcal{PRE}$) can be used to this purpose. Consider a method m that might violate an integrity constraint C. The backward predicate transformer generates a sufficient condition on the input data of this method for it to be safe with respect to C. The implementation of m can be modified to include that preliminary condition.

We are currently implementing the temporal prover extending the prototype described in [BS98] to the case of dynamic constraints. In the future, we plan to enrich the considered update language so as to take into account, for instance, object creation and migration. It could also be interesting to extend our predicate transformers to other temporal operators ($\overleftarrow{\bigcirc}$, for instance). Finally, an inportant open question is the existence of classes of formulae for which the theorem prover is complete w.r.t. to validity *tout court*.

Références

[Aba89] Martin Abadi. The power of temporal proofs. Theorical Computer Science, 65(1):35–84, June 1989.

[AHVdB96] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal versus first-order logic to query temporal databases. In Proc. of the 15th Int. Conf. on Principles of Databases (PODS'96). ACM, 1996.

[AM90] Martin Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. Journal of the ACM, 37(2):279–317, April 1990.

[BC96] John Boyland and Giuseppe Castagna. Type-safe compilation of covariant specilization: a pratical case. In Proc. of the 10TH European Conference of Object-Oriented Proogramming (ECOOP'96), LNCS, 1996.

[BCST96] Michael H. Bőhlen, Jan Chomicki, Richard T. Snodgrass, and David Toman. Querying tsql2 databases with temporal logic. In Proc. of the 4th International Conference on Extending Database Technology (EDBT'96), LNCS. Springer-Verlag, 1996.

[BD95] Nicole Bidoit and Sandra DeAmo. A first step towards implementing dynamic algebraic dependencies. In Proc. of the 5th International Conference on Database Theory (ICDT'95), LNCS. Springer-Verlag, 1995.

[BD98] Nicole Bidoit and Sandra DeAmo. A first step towards implementing dynamic algebraic dependencies. Theoretical Computer Science, 2(190):115–149, january 1998.

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: a framework for programming in temporal logic. In Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Springer-Verlag, 1989.

[BS98] Véronique Benzaken and Xavier Schaefer. Static management of integrity in object-oriented database design and implementation. In Proc. of the 6th International Conference on Extending Database Technology (EDBT'98), LNCS. Springer-Verlag, 1998.

[Cas96] Giuseppe Castagna. Object-Oriented Programing: A Unified Foundation. Progress in Theoretical Computer Science Series. Birkauser, Boston, 1996.

[CCM98] Serenella Cerrito and Marta Cialdea Mayer. Bounded model search in linear temporal logic and its application to planning. In Proc. of the Automated Reasoning with Analytic Tableaux and related Methods (TABLEAUX'98), LNCS. Springer-Verlag, 1998.

[CCMP99] Serenella Cerrito, Marta Cialdea-Mayer, and Sébastien Praud. First order linear temporal logic over finite time strucures. In Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99), LNAI. Springer-Verlag, 1999.

[Cho92] J. Chomicki. History-less checking of dynamic integrity constraints. In Proc. of the International Conference on Data Engineering. IEEE, 1992.

[Cla79] E.M. Clarke. Programming languages constructs for which it is impossible to obtain good hoare axiom systems. Journal of the ACM, 26(1):129–147, January 1979.

[Dij76] E.W. Dijkstra. A Discipline of Programming. prentice-hall, 1976.

[DS90] E.W. Dijkstra and C.S. Scholten. Predicate Calculus and Program Semantics. Text and Monographs in Computer Science. Springer-Verlag, 1990.

[Krö90] F. Kröger. On the interpretability of arithmetic in temporal logic. Theoretical Computer Science, 73(1):47–60, June 1990.

[Mag97] Nicolas Magnier. Validation des Transactions dans les Bases de Données : Classes Décidables et Vérification Automatique. PhD thesis, Université de Bordeaux I, 1997.

[Qia90] Xiaolei Qian. An axiom system for database transactions. Information Processing letters, 36(4):183–189, November 1990.

[Qia93] Xiaolei Qian. The deductive synthesis of database transactions. ACM Transactions on Database Systems, 1993.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. Journal of the ACM, 32(3):733–749, 1985.

[Sch97] Xavier Schaefer. Bases de Données Orientées Objets, Contraintes d'Intégrité et Analyse Statique. PhD thesis, Université de Paris I, Panthéon-Sorbonne, 1997.

[SGL97] P.H. Schmitt and J. Goubault-Larrecq. A tableau system for linear-time temporal logic. In Proc. of the 3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), LNCS. Springer-Verlag, 1997.

[SS89] T. Sheard and D. Stemple. Automatic verification of database transaction safety. tods, 14(3):322–368, September 1989.

[UG87] Lipeck U.W. and Saake G. Monitoring dynamic integrity constraints based on temporal logic. Information System, 12(3):255–269, 1987.

[Wol85] P. Wolper. The tableau method for temporal logic: an overview. Logique et Analyse, 28:119–152, 1985.

Véronique Benzaken *is full professor in the Department of Computer Science of the University Paris XI since 1998. She belongs to the Database team of the Laboratoire de Recherche en Informatique (L.R.I.).*

Serenella Cerrito *is assistant professor in the Department of Computer Science of the University Paris XI since 1991. She belongs to the Database team of the L.R.I.).*

Sébastien Praud *is a Ph.D. Student in Computer Science at the University Paris XI since 1997. His research advisors are Véronique Benzaken and Serenella Cerrito.*