Efficient verification of product model data: an approach and an analysis

V.A. Semenov, A.A. Bazhan, S.V. Morozov & O.A. Tarlapan
Institute for System Programming of the RAS, Moscow, Russia

ABSTRACT: In the paper an approach to verification of product model data is developed and discussed. The general approach encompasses all the variety of data and constraints assumed by EXPRESS language and provides for efficient solutions to verify product model data both completely and incrementally under single and multiple updates caused by insertion, deletion, and modification operations. The implementation methods are also of the paper subject. The methods are oriented on static analysis of specifications, compilation of optimized codes for the integrity checking and maintaining procedures and their efficient runtime execution. In the paper a performance analysis is conducted to compare the developed OpenSTEP Checker application with available similar programs and to form some qualitative criteria for efficient verification of product model data in various application contexts. The results are given in conformity to IFC standard that is of principal importance for achieving semantic interoperability in the architecture, engineering, and construction.

1 INTRODUCTION

The STEP is a family of emerging international Standards for the Exchange of Product Model Data developed by the ISO Technical Committee 184 "Industrial automation systems and integration" (ISO 1994). The STEP provides standardized mechanisms to specify information models using the EXPRESS language as well as to exchange and to share model-driven data in the ways neutral to potential software platforms and applications. Since such data are usually generated and shared by different applications, some integrity constraints may be violated, data consistency may be falsified and application interoperability may be destroyed. Therefore, STEP-compliant applications and distributed systems must incorporate some mechanisms to ensure that integrity constraints are always satisfied and the shared data are consistent and meaningful for all the stakeholders involved in joint multidisciplinary projects.

The tasks of integrity enforcement attracted much effort in the area of database management systems, particularly in the deductive and relational database communities (Pacheco 1997). The issues of integrity checking and maintenance of object-oriented data were also investigated (Mayol & Teniente 1999), but in some restrictions imposed on the information models, integrity constraints and update operators, not enabling expansion of the results into the verification tasks relating to general EXPRESS schemata.

In the paper the approach to verification of product model data is developed and discussed. The general approach encompasses all the variety of data and constraints assumed by EXPRESS information modeling language. In particular, it allows such constraints as referential integrity, cardinality constraints, attribute domain and object domain constraints, uniqueness and global rules. The approach provides for efficient solutions to verify product model data both completely and incrementally under single and multiple updates caused by insertion, deletion, and modification operations. Complete verification is usually employed to guarantee consistency of the information accumulated in persistent data stores or exchanged between interoperable software applications. Incremental verification allows to manage the information consistency more effectively if the data updates have local and latent character. In particular, this is the case if common multidisciplinary data are shared among separate software applications involved in distributed collaborative transactions (Ramamritham & Chrysanthis 1997).

The approach implementation issues are also of the paper subject. The presented methods are oriented on effective static analysis of EXPRESS specifications, compilation of optimized codes for the integrity checking and maintaining procedures and their highly efficient runtime execution.

Realization of the complete verification is rather straightforward. It is based on translation of constraint predicates given by declarative specifications

of information schema into some imperative language and corresponding execution environment.

Realization of the incremental verification is more sophisticated as it must guarantee acceptable localization of all the potential violations caused by possible data updates. The fundamental means of the developed incremental method are reconstructed dependency and inference graphs permitting to interrelate EXPRESS constraints and possible data updates in pre-compile time and to significantly reduce the search space for the integrity checking procedures executed at runtime (Semenov et al. 2004b).

Being realized in a strong way, the incremental verification enables to detect potential violations without performing costly evaluations of all the constraints imposed by an information schema. Nevertheless, it requires additional expenses on analysis of established inference relations between constraints and updates and may result in lower total efficiency. In the paper a performance analysis is conducted to compare the developed OpenSTEP Checker application with functionally similar programs and to form some qualitative criteria for efficient verification of product model data in various application contexts. The results of analysis are given in conformity to emerging IFC standard (IAI 1999) that is of principal importance for achieving semantic interoperability of software applications in the architecture, engineering, construction, and facility management industry domains.

2 OBJECT-ORIENTED METAMODEL

2.1 Formal model

So, we define an object-oriented data schema as a structure $S = \langle T_S, \prec, Attr_S, Func_S, Rule_S \rangle$ with the following meaning:

- $T_S = T^D{}_S \cup \overline{T^D}{}_S \cup C_S$ — a set of data types of the information schema consisting of basic types $T^D{}_S$ mapped into basic semantic domain D, derived types $\overline{T^D}{}_S$ mapped into multi-valued constrained semantic domain $\overline{D}$, and object types C_S;
- $\prec$ — a partial order on T_S reflecting the generalization/specialization relations induced to employ an inclusion polymorphism;
- $Attr_S$ — a set of attributes and constants. Each attribute $a_C \in Attr_S$ defined by the object type C is represented by a pair of operation signatures $a_C : C \mapsto T$, $a_C : C \times T \mapsto C$ for functions accessing the attribute value of the type T. Defining an attribute $a_C \in Attr_S$ of the type C' establishes an association relation between the object types C, C' with the corresponding role a_C;
- $Func_S$ — a set of imperative methods $f \in Func_S$ with a generic signature $f : T_1, \ldots, T_n \mapsto T'_1, \ldots, T'_m$, where the input and output data types $T_1, \ldots, T_n$, $T'_1, \ldots, T'_m \in T_S$ may be basic, derived or object types. For schema functions $f \in Func_S$ the signature is reduced to $f : T_1, \ldots, T_n \mapsto T'$. Derived attributes of objects $f_C \in Func_S$ can be represented by the specialized signature $f_C : C \mapsto T'$;
- $Rule_S$ — a set of semantic rules $r \in Rule_S$ imposing integrity constraints upon object-oriented data. They may limit the number, kind and organization of objects as well as impose relationships among their states. The rules are represented by the predicate signature $r : T_1, \ldots, T_n \mapsto logical$. Depending on data types $T_1, \ldots, T_n \in T_S$ of the schema the scope of the rules may be expanded into separate data, object instances and whole object populations.

In a way quite similar to the algebraic specification approach (Richters & Gogolla 1998), we provide a signature $\Sigma_S = (T_S, \prec, \Omega^D{}_S \cup \Omega^{\overline{D}}{}_S \cup Attr_S \cup Func_S \cup Rule_S)$, where $\Omega^D{}_S$, $\Omega^{\overline{D}}{}_S$ are sets of operations defined on basic and complex derived types. The signature formed by such way describes all of the types and the operations belonging to the information schema S as well as contains the initial set of syntactic elements upon which the expressions $Expr(\{var_T\} \mid T \in T_S)$ with variables indexed by the types can be defined.

A set of interrelated objects defined by the information schema S with attribute values and associations established among them constitutes the state of an object-oriented model M_S. In conformity with our discussion, M_S is a product model data defined by some information schema S.

A named subset of objects of the model M_S relating to the same type $C \in C_S$ is referred to by object population $p \subset M_S$. Let $C(p) \in C_S$ is an object type assigned to the population p of the model M_S. The set of all the objects of the model M_S belonging to the same type $C \in C_S$ is called by type extent $ext(M_S, C)$. Obviously that if the types $C_1 \in C_S$, $C_2 \in C_S$, and $C_1 \prec C_2$, then the subset relation $ext(M_S, C_1) \subseteq ext(M_S, C_2)$ takes place for the derived type extents. Moreover, for any population p of the model M_S such as $C(p) \prec C$, the relation $p \subseteq ext(M_S, C)$ is always true.

For the purposes of clarity we explain further the introduced formalism and consider shortly syntax and semantics of the EXPRESS language that is of great importance for specification of arbitrary product model schemas using the general object-oriented notation.

2.2 EXPRESS data model

EXPRESS introduces basic data types {*Real, Integer, Number, Boolean, Logical, String, Binary*} $\subseteq T^D$ of usual semantics. Complex types T^D and object types C are defined using {*Bag, Set, List, Array, Enumeration, Select, Defined*} and {*Entity*} declarations correspondingly. These metatype constructions permit to

define different sorts of aggregates, enumerations, selections, classified objects as well as to derive nested data types with hierarchically imposed constraint sets from already defined ones.

Simple data types are *Boolean*, *Logical*, *Number* (including particular subtypes *Real* and *Integer*), *String*, and *Binary*. Interpretation of the simple types is usual, but we extend each set with a special value *?* denoting the undefined value. A type $T \in T^D$ is mapped to a semantic domain D by a function $I : T \mapsto D$ as follows: I (*Boolean*) = {*true, false*} $\cup \{?\}$, I (*Logical*) = {*true, false, unknown*} $\cup \{?\}$, I (*Real*) = $R \cup \{?\}$, I (*Integer*) = $Z \cup \{?\}$, I (*String*) = $A^* \cup \{?\}$, where A is a finite alphabet, A^* is a set of all sequences over the alphabet A, and I (*Binary*) = $\{0,1\}^* \cup \{?\}$. The *String* and *Binary* variables may have fixed or varying sizes with limits defined by corresponding derived types.

Repertoire and semantics of the operations defined on the simple types is well understood. These are arithmetic operations on numeric operands of *Number*, *Real*, *Integer* types and standard mathematical functions; translation of the numbers into string representation and backwards; logical operations on operands of *Logical* or *Boolean* types; concatenation, pattern matching, indexing, subset operations on *String* and *Binary* operands as well as comparison operations defined for all the simple types. Some operations have the same overloaded name symbol and can be distinguished only by looking at their argument types.

Constructed types that may be either enumeration or selection data types extend the set of basic types. The domain of an enumeration type *Enumeration* is given by an ordered set of values represented by unique names $I(Enumeration(a_i, i = 1,\ldots,n)) = \{a_i \in A^*, i = 1,\ldots,n\} \cup \{?\}$. The literal values of the enumeration type are referred to as enumeration items. Ordering of the enumeration items results in comparison operations.

A selection data type *Select* defines a derived type represented by a list of the other underlying types. The selection instance is an instance of one of the types specified in the selective list. Therefore the selection data type is a generalization of its underlying types and its value domain is the union of the underlying type domains $I(Select(T_i, i = 1,\ldots,n)) = I(T_1) \cup \ldots \cup I(T_n) \cup \{?\}$.

A *Defined* data type is an user extension to the standard data types available in EXPRESS excepting that it enables to define additional semantic constraints imposed upon specified data. Therefore the defined type is always a concretization of its underlying type and its value domain $I(Defined(T)) = I(T) \cup \{?\}$.

Object types are defined by declaring their explicit, inverse, derived attributes and local rules. Generalization relations between object types are established via simple and multiple inheritance mechanisms. The inherited object type shares all the attributes and rules encapsulated by its parent supertypes, but can redefine them by specializing attribute types and imposing more restricted semantic constraints upon object states.

On object types the following common operators are defined. The relational operators = and <> are intended to compare the real states of objects. Unlike deep comparison operators, the operators :=: and :<>: permit to identify objects themselves. For a given object the function *rolesof* returns a list of qualified names of associations in which it takes part. The function *usedin* returns a set of objects connected with a given object via an association given by its qualified name. Availability of these functions in the language repertoire makes possible the navigation over object-oriented models via associations in both directions.

Multi-valued expressions in EXPRESS are described by aggregation metatypes *Array*, *List*, *Bag*, and *Set*. So, *Array* data type is a fixed-size structure where indexing of the elements is essential. Optionally, arrays can admit that not all of the elements have a value. *List* data type represents an ordered collection of like elements. A list can hold any number of elements allowing or, optionally, not allowing their duplication. *Bag* data type is a collection of elements in which order is not relevant and duplication is allowed. And, finally, *Set* is a collection of elements in which order is not relevant and where duplicate elements are not allowed. The number of elements in lists, bags, sets may vary, depending on their limit specifications.

Set operators +, -, * are defined to union, to difference, and to intersect collections of compatible types T_1, T_2. Relational operators, including subset and superset operators <=, >=, are intended to establish relations of equality and generality between collections. Pairs of relational operators =, <> and :=:, :<>: are semantically equivalent for all the aggregation data types except for aggregates of objects with differences pointed above. The membership operators *value_in* and *in* test a given element for membership in an aggregate by value and instance equivalence. The *query* expression evaluates a logical condition individually against each element of an aggregate and returns an aggregate containing the elements for which the logical expression evaluates *true*. The indexing operator *[]* extracts a single element from an aggregate. Operations *sizeof*, *hiindex*, *loindex, hibound, lobound* return the number of elements in an aggregate, the upper and the lower indices of array elements, the upper and the lower bounds of lists, bags, sets correspondingly. The *insert* and *remove* operations are defined additionally to simplify manipulations with lists.

Aggregation, selection, and defined types may be nested allowing the construction of more complex multidimensional structures that cannot be avoided

in such non-trivial information models as STEP application protocols (ISO 1994).

EXPRESS provides for predefined abstract data types *Generic, Aggregate, BagOfGeneric, SetOfGeneric, ListOfGeneric, ArrayOfGeneric* that can be used to specify functional methods in generic manner. Being defined on *Generic* type some operators can be applied also to any data. These are the function *exists* that tests whether a given variable has defined state, the function *nvl* that returns alternative value if a variable is in undefined state, and the function *typeof* that forms the set of all type names the given value belongs to.

2.3 EXPRESS constraint model

In addition to declarative part, the EXPRESS language provides for imperative constructions necessary to specify the integrity constraints imposed upon object-oriented data. The semantic constraints are defined in the form of rules that can be conditionally categorized into three groups.

The first group is related to rules defined for separate data items and imposed upon them:

- limited width of $data \in String$, $data \in Binary$: $length(data) \le n$, $n > 0$ for strings and binaries of varying length or $length(data) = n$, $n > 0$ for data of fixed length;
- limited number of elements in aggregates: $lobound(aggr) \le sizeof(aggr) \le hibound(aggr)$ for lists, bags, and sets $aggr \in ListOfGeneric \cup BagOfGeneric \cup SetOfGeneric$, and $sizeof(aggr) = hiindex(aggr) - loindex(aggr) + 1$ for arrays $aggr \in ArrayOfGeneric$;
- nonidentity of elements in unique aggregates: $aggr[i] :\neq: aggr[j]$ for all indices such as $i \neq j$ and $i, j = 1,\ldots,sizeof(aggr)$ in sets and unique lists $aggr \in SetOfGeneric \cup ListOfGeneric$, and $i, j = loindex(aggr),\ldots,hiindex(aggr)$ in unique arrays $aggr \in ArrayOfGeneric$;
- defined elements in non-optional arrays $aggr \in ArrayOfGeneric$: $exists(aggr[i]) = True$, where $i = loindex(aggr),\ldots,hiindex(aggr)$;
- domain rules for a defined data type $data \in Defined(T)$: $rule_i(data) = True, i = 1,\ldots,n$, where the rules $rule_i$ are given by logical expressions. The value domain of the defined type *Defined*(*T*) is formed as a domain of the underlying type *T* except for the values violating at least one rule.

The second group consists of the rules assumed or defined by separate object types and shared by their instances. The following constraints are covered by this group:

- type compatibility of object attributes and assigned values $obj.a = val$, $obj \in C$: $value\_in(type(obj.a), typeof(val)) = True$. The types must be equivalent or the attribute type must generalize the value type;
- required values of non-optional attributes of objects $obj \in C$: $exists(obj.a_i) = True$, $i = 1,\ldots,n$;
- limited cardinality of inverse associations $C_1.a \xrightarrow{\{m:n\}\ \{p:q\}} C_2.b$ in objects $obj \in C_2$: $p \le sizeof(obj.b) \le q$. If an inverse attribute is specified as a set of objects, an uniqueness constraint $obj.b[i] :\neq: obj.b[j]$ for all the indices such as $i \neq j$, $i, j = 1,\ldots,sizeof(obj.b)$ is additionally imposed upon the objects participating in such associations;
- nonidentity of value sets for unique attributes $C.a_1,...,a_k$: $obj_i.a_1 :\neq: obj_j.a_1$ or … or $obj_i.a_k :\neq: obj_j.a_k$ for $obj_i, obj_j \in ext(M_S, C)$, $i \neq j$, i.e. no two objects of the model cannot share the same set of unique attribute values;
- domain rules for objects $obj \in C$: $rule_i(obj) = True$, $i = 1,\ldots,n$, where the rules $rule_i$ are given by some logical expressions. The inspected object belongs to value domain if all the domain rules predefined by its types are satisfied. These constraints are used to bound the values of individual attributes and their combinations for separate objects and their groups.

Finally, the third category of constraints available in the EXPRESS language is global rules: $rule_i(ext(M_S, C_1),\ldots,ext(M_S, C_k)) = True, i = 1,\ldots n$ where the rules $rule_i$ are given by some logical functions with factual parameters being corresponding object type extents. These rules are defined immediately by an information schema and enable interrelate the states of whole object populations.

Notice that the result of verification of any rule is logical one meaning that the constraint may be asserted, unknown or violated. According to the EXPRESS semantics only violated constraints are interpreted as not conforming to the schema. Therefore to be valid the model must contain all valid objects and also satisfy all the uniqueness and global rules with asserted or unknown status. To be valid the object must satisfy all the attribute constraints and domain rules with asserted or unknown status too.

3 VERIFICATION

There are two typical statements of verification problems in conformity to STEP-driven product model data. First one is the complete verification that consists in checking of all the data within an object-oriented model M_S in strong correspondence with all the rules formalized and enumerated below. The second one is the incremental verification reasoning from local and latent character of possible updates in the model M_S.

The single updates may be object deletion $delete(obj)$, $obj \in M'_S$, object modification *modify(obj)*, $obj \in M'_S, M''_S$ and object insertion $insert(obj)$, $obj \in M''_S$, where M'_S, M''_S — preceding and current states of the model. Multiple updates

are compositions of the single updates. Often, being occurred within transactions, the updates have partial character, which may be effectively exploited for localization of potential violations without performing costly evaluations of all the constraints for the model. Because of complete verification of complex large-scale product model data is a computationally expensive task, the incremental approach may result in higher efficiency and makes the consistency enforcement policy to be applicable in some practice-important cases.

3.1 Complete verification

Complete verification is rather straightforward and can be conducted using the following transparent algorithm:

```
for each Object in Model
    for each Attribute of Object
      Check AttributeRule (Object, Attribute)
    for each domain Rule of Object
      Check DomainRule (Object, Rule)
for each uniqueness Rule defined for type C
    Check UniquenessRule (ext(Model,C), Rule)
for each global Rule defined for types C1,…,Ck
    Check GlobalRule (ext(Model,C1),…,ext(Model,Ck),Rule)
```

It is suggested that the algorithm logs the errors pointing out what rules are violated and for which attributes, objects and type extents. In the first external loop all the constraints related to separate object attributes and object domains are checked. The next loop realizes checking of uniqueness rules defined for some object types. To evaluate the rules the corresponding type extents have to be extracted from the inspected model. In the third loop the algorithm checks all global rules with factual parameters being proper type extents of the model.

3.2 Incremental verification

Here we outdraw the developed method for incremental verification of data defined by the EXPRESS schemata. The method takes into account all variety of the constraints available in EXPRESS and exploits the introduced concepts of dependency and inference graphs. In more details the method is presented and described in our work (Semenov et al. 2004b).

First of all, to effectively analyze data dependencies, transitions between model populations should be defined. Every transition either direct or associative one realizes some relation between the populations and produces some transition function that being evaluated makes the states of the interrelated populations to be consistent. A direct transition $r \in R^D$, $r : p_i \to p_j$ $(i \neq j)$ realizes a subset relation between object populations of compatible types so as if $C(p_i) \prec C(p_j)$ then $p_i \subset p_j$ and if $C(p_i) \succ C(p_j)$ then $p_i \supset p_j$. Every associative transition $r \in R^A$, $r : p_i \to p_j$ (possibly, $i = j$) realizes some association relation defined by the schema for the corresponding object types $C(p_i)$ and $C(p_j)$ in explicit or implicit way. In particular, if a transition realizes an association *role*, then the relation takes form $Use(p_i, role) \subset p_j$, where $Use(p_i, role)$ is a set of all the objects associated with the population objects p_i by the *role*.

A production function for the direct transition $r \in R^D$, $r : p_i \to p_j$ can be defined as $p'_j = p_j \cup obj \in p_i \mid C(obj) \prec C(p_j)$. For the associative transition $r \in R^A$, $r : p_i \to p_j$ a production function is formed as $p'_j = p_j \cup obj \in Use(p_i, r) \mid C(obj) \prec C(p_j)$.

The transitions can be inverted. A production function of the inverted direct transition $r \in R^D$, $\bar{r} : p_j \to p_i$ is given by $p'_i = p_i \cup obj \in p_j \mid C(obj) \prec C(p_i)$. For the inverted associative transition $r \in R^A$, $r : p_j \to p_i$ a production function can be formed as $p'_i = p_i \cup obj \in Usedin(p_j, r) \mid C(obj) \prec C(p_i)$, where $Usedin(p_j, role)$ is a set of all the objects with which the population objects p_j are associated by the *role*. Being applied twice, the inversion operation returns the transition in its original state.

Let S is an information schema, P is a set of populations with prescribed types $C(P) \in C_S$ and attribute subsets $A(P) \in Attr_S$, and $R = R^D \bigcup R^A$ is a set of direct and associative transitions between populations P. We call the structure $AG\langle P, A, R\rangle$ where populations P are represented by vertices, attributes A — by marks assigned to vertices, transitions R — by directed edges incident to corresponding population vertices by attributive graph for the schema S.

Navigation graph $NG\langle P, U, V, A, R\rangle$ is like an attributive graph $AG\langle P \cup U \cup V, A, R\rangle$, where some population vertices U and V are interpreted as sources and sinks. In the used graphic notation the source vertices are marked by incoming edges, the sink vertices — by outgoing edges. For the navigation graph the operations of inversion, composition, and reduction can be constructively defined. In particular, inversion of the navigation graph results in inversion of all the transitions R and changing places of all the source and sink vertices U, V.

The navigation graph $NG\langle P, U, V, R, A\rangle$ for the schema S gives a declarative query in M_S. Parameters of the query are initial states $u \subset M_S$ of the source populations U, and the result is the states $v \subset M_S$ of the sink populations V such as all the transition relations are satisfied. The query can be evaluated by means of production functions applied to all transitions of the graph until the population states are not updated.

Although navigation graphs allow cycles, the definition of the query is constructive as it guarantees determinism: the query processing routine is always finished for a finite number of evaluations of production functions and the query result does not

depend on the way how the transitions were serialized.

The introduced navigation graphs and queries have an important application for analysis of data dependencies caused by imposed constraints. For this purpose a dependency graph $DG_f\langle P,U,V,R,A\rangle$ can be constructed for each method $f \in F_S$ of the schema S.

The method dependency graph is a navigation graph that can be constructed for any method of the schema like procedure, function, derived attribute, domain or global rules in accordance with some formal procedure. The procedure assumes that formal and factual parameters of the method, local variables, regular path expressions having object-related types correspond to the graph vertices P, U, V. At that, local variables and regular expressions are represented by the vertices P, input parameters and factual output parameters of called methods — by the sources U, output parameters and factual input parameters of called methods — by the sinks V. The object-related types mean here object types, aggregates of objects, selections of objects, and proper nested derived types.

Attribute subsets A are formed and prescribed to corresponding vertices P, U, V, if attributes of all the other types are used in the method expressions having object-related types. Transitions in the dependency graph correspond to operations of type casting, set-theoretic operations, calls of methods, and also to separate terms in regular path expressions. For brevity we omit the details and address to the mentioned work, where the procedure has been completely described.

The schema dependency graph $DG_S\langle P,U,V,R,A,F\rangle$ is a graph composed of the dependency graphs $DG_f\langle P,U,V,R,A\rangle$ constructed for all the methods $f \in F_S$ of the schema S. As a result of the composition all the elements belonging to the particular dependency graphs are marked by the method identifiers F and method calls are resolved in the following way. The sinks being factual input parameters of called methods are connected via direct transitions with sources being formal parameters of these methods. The sinks being formal output parameters of called methods are connected via direct transitions with sources being factual parameters of the methods. At that, all the sources not participating in method calls resolution and being sources of verification rules are remaining sources, and all the other vertices are becoming sinks (at more detailed consideration additional categorization of vertices is possible and may be useful).

Being navigation graph and defining queries in the inspected model, the schema dependency graph can be applied to localize the model objects that would participate in verification of separate rules. For this purpose the sources of the analyzed rule $f \in F_S$ have to be initialized by the model objects and a query based on the schema dependency graph has to be evaluated. The result of the query will include all the model objects potentially participating in verification of the rule. Indeed, taking into account static associative and subset relations between object populations and omitting conditional statements, the method and schema dependency graphs result in extended querying and guarantee localization of all the objects participating in the rule verification.

The graph $IG_S\langle P,U,V,R,A,F\rangle$ inverted to the schema dependency graph $DG_S\langle P,U,V,R,A,F\rangle$ is called by inference graph.

By construction the inference graph enables to localize rules that might be violated as a consequence of some updates occurring in the model. To form a navigation query the sources of the inference graph have to be initialized by modified objects. The result of the query evaluation is the model objects accumulated in sinks. Each non-empty sink population gives both objects and assigned rule $f \in F_S$ that might be violated through modifications. It is essential that violations may occur only if some attributes and associations of the modified objects have been changed. The formed attribute subsets A and the constructed transitions R are just those metadata that permit to conduct such analysis. Navigation over the inspected model using the inference graph extends the sets of objects and rules that could be disturbed and, therefore, it guarantees localization of all the rule violations.

Using the assertions above, algorithms for incremental verification can be proposed. Here we present the algorithm applicable to single and multiple updates of all kinds. Modification operations are represented as a succession of the deletion and insertion operations.

```
construct scheme inference graph (SIG)
for each DELETED Object in Model
    for each Attribute' in RolesOf(Object)
      for each not deleted Object' in UsedIn(Object, Attribute')
          Add AttributeRule(Object', Attribute') to Check List
    for each not deleted Object' in Use(Object, Attribute)
      for each Attribute' of Object' inverse to Attribute
          Add AttributeRule(Object', Attribute') to Check List
    for each Source vertex in SIG such as
          Type(Object) in TypeOf(Source)
      Add Object to Source populations
evaluate query
for each Sink vertex in SIG
    for each Object' in Sink populations with assigned Rule
      if (Rule is domain) then
          Add DomainRule(Rule, Object') to Check List
      if (Rule is unique) and (Rule is not in CheckList) then
          Add UniquenessRule(Rule) to Check List
      if (Rule is global) and (Rule is not in CheckList) then
          Add GlobalRule(Rule) to Check List
delete DELETED Objects from Model

insert INSERTED Objects in Model
for each INSERTED Object in Model
    for each Attribute of Object
```


```
      Add AttributeRule (Object, Attribute) to Check List
    for each not inserted Object' in Use(Object, Attribute)
      for each Attribute' in Object' inverse to Attribute
        Add AttributeRule(Object', Attribute') to Check List
    for each Source vertex in SIG such as
        Type(Object) in TypeOf(Source)
      Add Object to Source populations
evaluate query
for each Sink vertex in SIG
    for each Object' in Sink populations with assigned Rule
      if (Rule is domain) then
        Add DomainRule(Rule, Object') to Check List
      if (Rule is unique) and (Rule is not in CheckList) then
        Add UniquenessRule(Rule) to Check List
      if (Rule is global) and (Rule is not in CheckList) then
        Add GlobalRule(Rule) to Check List

for each Rule(Rule, Object, Attribute) in Check List
    if (Rule is attribute) then
      Check AttributeRule (Object, Attribute)
    if (Rule is domain) then
      Check DomainRule (Object, Rule)
    if (Rule is uniqueness defined for type C) then
      Check UniquenessRule (ext(Model,C), Rule)
    if (Rule is global defined for types C1,…,Ck) then
      Check GlobalRule (ext(Model,C1),…,
                        ext(Model,Ck),Rule)
```

At the first phase the algorithm produces a list of rules that might be violated and have to be subjected to subsequent verification. Elements of the list are triples containing attribute, object, and rule. The list is formed of attribute rules for the inserted objects, cardinality attribute rules for the objects directly associated with the deleted and inserted objects as well as from domain, global and uniqueness rules localized by means of navigational querying. The queries based on the schema inference graph are initialized and evaluated twice: for localization of potential violations caused by deleted objects, then for localization of violations caused by inserted objects.

At the second phase the algorithm verifies suspicious rules that are extracted from the list, evaluated, and logged if some disturbances have been detected.

4 IMPLEMENTATION AND ANALYSIS

The presented approach to verification of STEP-driven product model data provides for implementation methods based on static analysis of the information schema specified at the EXPRESS language, compilation of executable codes for the integrity checking and maintaining procedures and their highly efficient runtime execution.

We foresee significant potential of the approach presented to verify such complex high-scale product data like those of defined by Industry Foundation Classes (IFC). This information standard is developed by International Alliance for Interoperability in conformity to architecture, engineering, construction, and facility management (IAI 1999). Significant resources consumed to control the data consistency are one of crucial points of available product data management technologies and solutions.

We consider three basic aspects and methods to increase efficiency of data integrity checking and maintaining procedures in accordance with the presented approach. These methods have been realized in OpenSTEP Checker: an application built on the general-purpose software platform by ISP RAS (Semenov et al. 2004a).

The first aspect is to exploit so-called early binding formation and implementation of data access interfaces taking into account particular information schemas. It enables to access product data and to evaluate the schema methods (rule predicates, derived attributes, functions, and procedures) with more speed than in the usage of late binding realizations. This advantage is achieved because execution of precompiled imperative instructions is always more efficient than their interpretation.

The second aspect is optimized implementation of the schema methods taking into account of peculiarities of the EXPRESS language. As our experience proves, the performance of the verification procedures can be increased by optimization of:

- queries given by object identifiers, inverse associations, and subtyping relations;
- operations on arithmetic and logical operands with extended (unknown) states;
- identity comparison operations;
- methods of recalculated derived attributes.

The third aspect is to take advantage of local and latent character of the model updates peculiar to transaction processing. The algorithm presented above enables to effectively control and maintain data consistency during user sessions. Its realization can be combined with two mentioned methods.

To estimate operational costs IFC2x files were used as timing benchmarks for some available verification programs, namely: OpenSTEP Checker, IfcObjCounter v.1.1 by Forschungszentrum Karlsruhe GmbH, Institute for Applied Computer Science (http://www.iai.fzk.de), Express Engine v.3.1.4 developed within an open source project (http://sourceforge.net/projects/exp-engine), and Express Data Manager v.4.080 by EPM Technology AS (http://www.epmtech.jotne.com). Timing measurements were made with data sets generated by CAD systems and ranging in size from 1000 up to 100,000 objects. Complete verification option was checked for all the explored applications. Costs to load data from STEP files were ignored, only verification costs were measured. Because of some programs didn't provide built-in CPU timing capabilities, all times were measured using a stop-watch.

The programs exhibited acceptable $O(N \log N)$ growth characteristic. An asymptotic performance of the IfcObjCounter was significantly slower, so we were unable to complete measurements on data sets with more than 50,000 objects. Its $O(N^2)$ extreme

behavior and a higher degree complexity might be explained by insufficient optimization of underlying verification algorithms and operations. In particular, verification of uniqueness rules resulted in reduction of the total performance more than 50 times even on small data sets.

Being based on Common LISP, the Express Engine demonstrated much slower characteristics compared with the OpenSTEP Checker. The conducted measurements showed average 30 times performance losses. It may be caused by slow speed of interpretation of LISP instructions and possible complication of implementation of verification programs against applications directly running in execution environments.

The Express Data Manager is quite balanced interpretation system that proceeds with arbitrary data and attended schemas both loaded at runtime. Nevertheless, it yields the pas to the OpenSTEP Checker that employs early binding method to pre-compile widely used EXPRESS schemas and to link the corresponding libraries to the target application dynamically. This method enabled to increase performance of the program more than 3 times.

Thus, the OpenSTEP Checker is the fastest application among the considered ones. Another advantage is a capability to verify the data in an incremental way. As such analysis is connected with additional expenses on evaluation of navigation queries, a priori quantitative criteria for the final efficiency have to be established. To estimate potential benefits of the incremental verification and to establish such criteria, special experiments have been performed. The experiments simulated series of short transactions each of which consisted of single object modification. The whole series covered all the objects contained in the inspected models. The total and averaged (related to the model size) CPU times of incremental analysis and verification were measured.

The experiments detected that the averaged time is roughly proportional to the time required for complete verification of the model. For the selected benchmarks this factor varied approximately from 0.001 up to 0.002. Therefore on short transactions covering a few random objects the averaged performance speedup amounts to hundreds and thousands times. At that, if the transactions include the updated objects participating in a few uniqueness or global rules, the speedup may be drastically reduced up to 10-25. On the other part, the conducted experiments showed that being applied to long transactions covering all the model objects, the incremental analysis might require the CPU resources 3-6 times exceeding the complete verification time. It means that the threshold value for fraction of updated objects at that the incremental method gives benefits may be varied widely enough. Nevertheless, if the fraction of object updates in the model is less than 15 percentages, the usage of the incremental method is quite motivated in most cases.

5 CONCLUSIONS

Thus, the approach to efficient verification of STEP-driven product model data has been presented. The approach combines ideas of exploiting early binding implementation of data access interfaces, deep optimization of verification programs and alternative usage of complete and incremental methods. The conducted timing experiments showed that the efficiency of the consistency checking and maintaining procedures can be significantly increased and the presented approach to efficient verification is realizable and quite marketable. The demo version of the OpenSTEP Checker can be downloaded from the project site www.ispras.ru/~step.

The presented work is supported by the Russian Foundation for Basic Research (grant 04-01-00527) and the Russian Science Support Foundation.

REFERENCES

IAI 1999. IFC Technical Guide, International Alliance for Interoperability, <http://www.iai.org.uk/documentation/IFC_2x_Technical_Guide.pdf>

ISO 1994. ISO 10303: 1994, Industrial automation systems and integration — Product data representation and exchange.

Mayol, E. & Teniente, E. 1999. A survey of current methods for integrity constraint maintenance and view updating. *Advances in Conceptual Modeling: ER '99 Workshops on evolution and change in data management, Reverse engineering in information systems, and the world wide web and conceptual modeling, Paris, France, November 15–18, 1999*: 62-73.

Pacheco, M. A. 1997. Dynamic integrity constraints definition and enforcement in databases: a classification framework. *Proceedings of the IFIP TC-11 Working Group 11.5 First working conference on integrity and internal control in information systems, Zürich, Switzerland, December 1997*: 65-87.

Ramamritham, K. & Chrysanthis, P. 1997. *Executive briefing: advances in concurrency control and transaction processing*. IEEE Computer Society Press.

Richters, M. & Gogolla, M. 1998. On formalizing the UML object constraint language OCL. In Tok-Wang Ling (ed.), *Proceedings of ER'98 — 17th International conference on conceptual modeling, LNCS* 1507: 449-464. Berlin: Springer.

Semenov, V.A., Bazhan, A.A., Morozov, S.V. 2004a. Distributed STEP-compliant platform for multi-modal collaboration in architecture, engineering and construction. *Proceedings of X international conference on computing in civil and building engineering, Weimar, June 02–04 2004*: 318-319.

Semenov, V.A., Morozov, S.V., Tarlapan, O.A. 2004b. Incremental verification of object-oriented data using constraint specifications. In V.P. Ivannikov (ed.), *Proceedings of Institute for System Programming* 8(2): 21-52. Moscow: ISP RAS.