An Approach to Consolidation of Database Check Constraints

Nikola Obrenović*, Ivan Luković**
* Schneider Electric DMS NS Llc., Novi Sad, Serbia
** Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
nikola.obrenovic@schneider-electric-dms.com, ivan@uns.ac.rs

Abstract—Independent modeling of parts of an information system, and consequently database subschemas, may result in conflicts between the integrated database schema and the modeled subschemas. In our previous work, we have presented criteria and algorithms for resolving such conflicts and a consolidation of a database subschema with the database schema with respect to various database concepts, e.g. domains, relation schemes, primary key constraints, etc. In this paper we present an approach and new algorithms for identification of conflicts and subschema consolidation against check constraints.

I. INTRODUCTION

Modeling of relational database schemas can be performed in two ways: 1) directly, by having the entire database schema modeled at once, and 2) part-by-part, by modeling independently parts of database schema, i.e. subschemas. When the schema is modeled part-by-part, the created subschemas need to be integrated into a unified database schema.

Our previous work advocates that an Information System (IS), including a relational database schema as its underlying foundation, should not be designed directly. Designing the whole IS at once can easily overcome designer's capabilities and result with a model of poor quality ([1, 2]). Therefore, we have developed a methodology for a gradual development of a database schema ([1, 3]), followed by a tool supporting the methodology, named *Integrated Information Systems CASE* or *IIS*Case* for short. In *IIS*Case*, designers specify models of isolated parts of an IS, i.e. information subsystems, in an independent way, by using a platform-independent model (PIM) of form types ([1, 4]). A form type concept is an abstraction of screen forms or documents that users utilize to communicate with the IS. By specifying form types of an information subsystem, designers also specify a database subschema with its constraints, as it is presented in [1, 4].

By applying a number of algorithms, *IIS*Case* transforms a set of form types into a relational database schema. The formal description of the transformations is out of the scope of this paper and can be found in [4, 5, 1]. Thereby, a database subschema is obtained from the set of form types specified at the level of an information subsystem. Similarly, the database schema of the whole IS is derived from the union of form types of all information subsystems.

In [2], it is shown that each database subschema must be formally consolidated with the integrated schema, in order to obtain a valid specification and implementation of an IS. A subschema is formally consolidated with its schema if each its concept, such as domain, attribute, relational scheme or constraint, is consolidated with the appropriate concept of the schema. Also in [2], the author presents algorithms for checking consolidation of subschemas with the schema with respect to domains, relational schemes, primary keys, uniqueness constraints, referential integrity and inverse referential integrity constraints. The proposed algorithms are also implemented in *IIS*Case*.

This paper extends our previous work with an algorithm for consolidation of subschemas with respect to check constraints. So as to provide consolidation test, we need to find a solution to the implication problem for two check constraints, i.e. how to detect if one check constraint is a logical consequence of the other one. However, the nature of check constraints is different from other types of database constraints, since they represent complex logical expressions. Consequently, a test of logical consequence of check constraints requires different methods than those used for other types of database constraints, such as functional dependencies and keys. Therefore, another goal of this work is to formulate the appropriate method for a test of logical consequence of check constraints.

Beside the Introduction and Conclusion, this paper consists of four sections. The related work is presented in Section 2. In Section 3, we present the algorithm for subschema consolidation with respect to check constraints only. The method for testing implication of two check constraints is presented in Section 4. In Section 5, some details of the algorithm implementation are discussed.

II. RELATED WORK

In [6], the authors have presented an overview of methodologies and techniques for integration of independently modeled relational database subschemas and detection of conflicts between the subschemas. By that, at least one of the presented methodologies addresses conflicts at the level of attribute naming, domains, cardinalities, primary keys or entity usage. However, none of the methodologies considers conflicts between check constraints. Furthermore, to the best of our knowledge, such a methodology has not been defined yet.

Also in [6], the authors concluded that the most of surveyed methodologies propose general guidelines for subschema integration, but they lack an algorithmic specification of integration steps. On the other hand, we propose the subschema integration which is formally defined and implemented in *IIS*Case*.

III. CONSOLIDATION OF CHECK CONSTRAINTS

In *IIS*Case*, check constraints can be modeled at the level of domain, attribute or component type, which is a logical part of a form type ([7]). When check constraints are transformed from the model of form types into the relational model, they become check constraints at the level of domain, attribute or a set of relational schemes, respectively ([8]).

The concepts of domain and attribute are modeled in the scope of entire IS, i.e. a domain or attribute inherits the same definition in each information subsystem as it is defined in the scope of the IS. Consequently, a check constraint at the level of a domain or attribute in a database subschema is identical to the check constraint at the level of the same domain or attribute in the database schema.

On the other hand, a component type check constraint is modeled in the scope of an information subsystem. A database subschema is a result of transformation of form types that represent one information subsystem. Likewise, a relational database schema is obtained by transforming the union of all information subsystem specifications into the relational data model. Therefore, after transformations, a component type check constraint exists both in the database schema and in the appropriate database subschema.

Two or more information subsystems can contain model of the same data, each of them from its own point of view. Consequently, two information subsystems can impose different constraints over the same set of data. In other words, two check constraints from different information subsystems may refer to the overlapping sets of relation schemes of database schema, which is illustrated in Example 1.

Example 1. Let us consider two form types UNIVERSITY ORGANIZATION (Figure 1) and FACULTY ORGANIZATION (Figure 2) which belong to different information subsystems of a university IS. The form type UNIVERSITY ORGANIZATION is used for manipulation of information about faculties and their respective departments at the level of the whole university. On the other hand, the form type FACULTY ORGANIZATION is used at the faculty level for manipulating data about faculty departments.

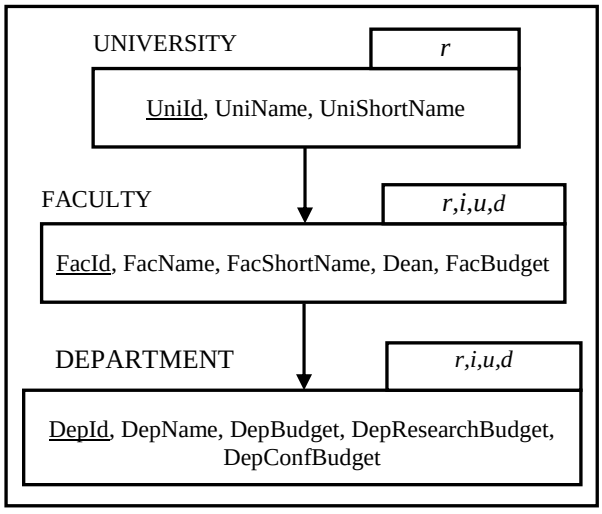

Figure 1. UNIVERSITY ORGANIZATION form type.

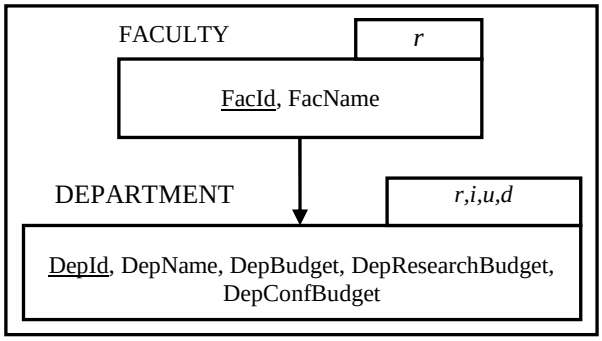

Figure 2. FACULTY ORGANIZATION form type

Form type UNIVERSITY ORGANIZATION consists of three component types: UNIVERSITY, FACULTY and DEPARTMENT, used for viewing and manipulating data about the university, faculties and the belonging departments, respectively. In order to control budget levels of the faculties and departments, the following check constraint is modeled in the DEPARTMENT component type:

DepBudget > *DepResearchBudget* AND
DepResearchBudget > *DepConfBudget*.

Form type FACULTY ORGANIZATION consists of the same two component types FACULTY and DEPARTMENT. However, component type FACULTY is used only for viewing existing faculties, while DEPARTMENT is also used for insert, update and delete operations. This information subsystem might be designed by another designer, whose point of interest differs from the first one. Therefore, he or she could model a different check constraint over DEPARTMENT component type:

DepBudget > *DepConfBudget*.

Form types UNIVERSITY ORGANIZATION and FACULTY ORGANIZATION are transformed into the following set of relation schemes given in the form N(R,K), where N is the name of the relation scheme, R is the set of attributes and K is the set of keys:

- University({UniId, UniName, UniShortName}, {UniId});
- Faculty({FacId, FacName, FacShortName, Dean, FacBudget, UniId},{FacId}); and
- Department ({FacId, DepId, DepName, DepBudget, DepResearchBudget, DepConfBudget, UniId}, {FacId+DepId}).

Thereby, relation scheme Department inherits check constraints from both component types. □

With respect to all constraints, a subschema is consolidated with its schema iff for each schema constraint of interest, there is an equally strong or stronger constraint in the subschema ([2]). Thereby, a schema constraint is of interest if it affects data modeled through the observed subschema.

Hence, for each check constraint in the schema, the consolidation algorithm first determines the information subsystems, i.e. their database subschemas, which the observed check constraint is of interest for. In the following text, we denote these subschemas as the corresponding subschemas. Further, in each of the corresponding subschemas, the algorithm checks if there

```
PROCESS CheckCheckConstraints(I(S, I_CC, SI_SUB),
                              O(Ind, Report),
                              IO( ))
  SET Report ¬ ∅
  SET Ind ¬ True
  DO CheckEachCheckConstraint (∀i_S ∈ I_CC)
    DO CheckEachSubschema (∀(S_i, I_i) ∈ SI_SUB)
      IF Attr(i_S) ∩ Attr(S_i) ≠ ∅ THEN
        IF Attr(i_S) ⊆ Attr(S_i) THEN
          SET Found ¬ False
          DO CheckSubschemaConstraints (∀i_SS ∈ I_i)
            IF i_SS ⇒ i_S THEN
              SET Found ¬ True
              BREAK
            END IF
          END DO
          IF Found = False THEN
            SET Ind ¬ False
            SET Report ¬ Report ∪ (S_i, i_S)
          END IF
        ELSE
          SET Ind ¬ False
          SET Report ¬ Report ∪ (S_i, i_S)
        END IF
      END DO
    END DO
END PROCESS
```

Figure 3. Algorithm for subschema consolidation with respect to check constraints

is a check constraint equally strong or stronger than the schema check constraint. If this condition is satisfied for each schema check constraint, database subschemas are consolidated with the schema with respect to check constraints. The pseudo code of the algorithm is given in Figure 3.

In the pseudo-code, the following notions are used:

- S – a set of relation schemes of the database schema;
- I_{CC} – a set of check constraints of the database schema;
- SI_{SUB} – a set of pairs (S_i, I_i), where S_i denotes set of relation schemes of subschema i, while I_i denotes set of check constraints of the subschema i;
- Attr – a function that returns set of attributes referenced by its argument, e.g. a subschema or a check constraint;
- *Report* – set of pairs (S_i, i_S) where i_S is the schema check constraint which makes subschema S_i unconsolidated with the database schema; and
- *Ind* – a Boolean indicator stating whether all subschemas are consolidated with the database schema with respect to check constraints.

Proving the implication between check constraints is the essential part of the consolidation algorithm, and it is presented in the following section.

IV. Implication Problem of Check Constraints

As it is presented in the consolidation algorithm, in order to check consolidation between a database subschema and a database schema, we need to be able to determine whether a subschema check constraint implies the corresponding schema check constraint, i.e. we evaluate validity of the formula:

$$i_{SS} \Rightarrow i_S, \tag{1}$$

where i_{SS} is a subschema check constraint and i_S is the corresponding schema check constraint. In the further text, we denote formula (1) also as the check constraint implication formula.

The body of a check constraint is a logical expression. Its interpretation, i.e. evaluation, is a three-state Boolean function which evaluates to *true*, *false* or *unknown*. Its result determines whether a tuple satisfies (*true*), violates (*false*) or neither satisfies nor violates the constraint (*unknown*). For the sake of simplicity, the term check constraint is further also used to denote the logical expression of the constraint.

In the further text, it is assumed that all check constraints are given in the conjunctive normal form (CNF):

$$\wedge\left(\bigvee_{i=1}^{m} l_i\right), \tag{2}$$

where each l_i represents an atomic logical expression, denoted as literal. The transformation of a logical formula into its CNF is described thoroughly in [9].

The literals of check constraints usually are not just Boolean variables or predicates. Instead, they are often expressions of various types: from integer and real, linear or non-linear arithmetic or over date, string or set types. The formal definition of a check constraint logical expression, which determines all possible literals, may be found in [7].

Example 2. The following expressions may represent check constraint literals:

- $A > 0$;
- $0.3*A + B > 15$;
- *DOB* > ToDate('1900-01-01');
- *SURNAME* LIKE "JOHN%"; or
- *X* IN [1,2,3,5,7,11],

where *A*, *B*, *DOB*, *SURNAME* and *X* are database schema attributes defined over some domains, i.e. data types. □

Since logical expressions of check constraints normally comprise sub-expressions of various types, they are more complex in regard to the implication problem test than database constraints of many other types. Let us observe the following examples. Key is a typical database constraint, formalized just with a single Boolean predicate, Key(N, X), where N is the name of the relation scheme and X is a set of attributes, while functional dependency is a single Boolean predicate of the form X→Y, where X and Y are attribute sets ([2]). Consequently, testing the implication of functional dependencies is a deterministic problem, for which we have the appropriate polynomial algorithm that do not consider domains of attributes in any way. On the contrary, testing the implication of check constraints is, in its general case, more complex problem, since any algorithm for this purpose needs to consider the properties, relations and operations over domains associated to all attributes included in the constraint.

Proving validity of logical formula (1), where each literal is a proposition, i.e. a Boolean variable, or a Boolean predicate, is a kind of a Boolean satisfiability problem (SAT problem, [10]). This class of problems belongs to the automated theorem proving problems and there is a vast number for algorithms and tools named SAT solvers intended for its solving ([11]).

However, an application of SAT solving techniques for proving (1) would imply that each check constraint literal is treated without taking its actual meaning into account. Also, the relations between different literals, which can be derived from the meaning, would be disregarded, as it is illustrated by the following example.

Example 3. In Example 1, two check constraints are introduced:

i_1*: DepBudget > DepResearchBudget AND DepResearchBudget > DepConfBudget*

and

i_2*: DepBudget > DepConfBudget*.

These check constraints contain the following literals:

- l_1: *DepBudget > DepResearchBudget*;
- l_2: *DepResearchBudget > DepConfBudget*; and
- l_3: *DepBudget > DepConfBudget*.

Let us further assume that i_1 is a subschema check constraint and i_2 is its corresponding schema check constraint.

By taking into account the transitivity property of the operator *greater than* over integer or real variables, one can infer the following relation between the abovementioned literals:

$$l_1 \wedge l_2 \Rightarrow l_3.$$

On the other hand, a SAT solver would treat the operator *greater th*an only as an uninterpreted two-argument Boolean predicate and could not infer any relation between the literals. Consequently, a SAT solver could not infer that i_1 implies i_2, i.e. that the subschema is consolidated with the schema with respect to check constraints i_1 and i_2. □

Therefore, in order to prove validity of (1), we also need to interpret the semantics of check constraint literals, which a pure SAT solver is not capable of. This disadvantage of SAT solvers initialized development of another research field named Satisfiability Modulo Theory (SMT, [11, 12]). SMT algorithms represent extensions of SAT algorithms with the knowledge and capability to reason over additional theories of interest, such as: linear arithmetic over integer or real numbers, non-linear arithmetic's over real numbers, theory of uninterpreted functions, theory of arrays, bit-vector theory, etc. By the SMT terminology, such theory is referred to as the background theory, while the reasoning methods deployed inside a theory are named the decisions procedures. In analogy to SAT solvers, software tools implementing SMT algorithms are named SMT solvers.

All SMT solvers provide checking the satisfiability of a logical formula and have an explicit command for this purpose. On the other hand, most available SMT solvers do not provide an explicit command for proving the validity of a logical formula. However, validity proof of a logical formula is a dual problem to proving its satisfiability ([13]). That is, we can prove that a logical formula is valid by proving that the formula's negation cannot be satisfied. By using this approach, we prove validity of (1) and consequently prove logical implication of check constraints.

V. Integration of *IIS*Case* and SMT Solvers

In order to test subschema consolidation, a SMT solver is integrated into *IIS*Case* in the following manner.

The specification of the negation of (1) is first transformed into the form and language required by the SMT solver and written into an input file for the SMT solver. With the input file, *IIS*Case* executes the SMT solver as an external process, which tries to prove the satisfiability of the input formula.

Further, the SMT solver creates an output file with the result of the satisfiability check, which is parsed by *IIS*Case*. If the satisfiability check fails, the check constraint implication formula is valid.

The creation of the input file for SMT solver consists of the following steps:

1. Transformation of the negation of (1) into CNF;
2. Preprocessing the negation of (1) in order to remove expressions not supported by the SMT solver; and
3. Transformation of the negation of (1) into the language understandable by the SMT solver.

All of these steps are further described in the subsequent subsections.

A. Transformation of The Check Constraint Implication Formula's Negation into CNF

To the best of our knowledge, all SMT solvers require processed formulas to be represented as a set of clauses, where each clause represents a conjunct of the formula's CNF.

Therefore, since we need to prove unsatisfiability of the negation of (1), which is:

$$\neg (i_{SS} \Rightarrow i_S), \tag{3}$$

we need first to transform it into its CNF:

$$i_{SS} \wedge \neg i_S. \tag{4}$$

Further, formulas i_{SS} and $\neg i_S$ are replaced with their CNF forms, respectively, in order to obtain the CNF of the whole (3), i.e. the set of input clauses for a SMT solver.

B. Preprocessing of Check Constraint Implication Formula's Negation

The state-of-the-art SMT solvers support a large number of background theories ([13]). However, to the best of our knowledge, none of the currently available SMT solvers supports operations over date, string or set variables which are allowed in the definition of a check constraint.

Therefore, in order to use SMT solver for proving implication of check constraints, the negation of check constraint implication formula needs to be transformed into a logical formula that can be interpreted by the SMT solver, i.e. a formula that does not contain date, string nor set operations. By that, the resulting formula's satisfiability must imply the satisfiability of the original formula. Additionally, the transformations need to preserve as much knowledge as possible about original literals and relations between them. This approach of preprocessing a logical formula before proving its

satisfiability is known as the *eager* strategy for solving SMT problems ([13]). In this work, we propose the following transformations of literals that contain date, string or set operations.

1) Trasformations of Literals Containing Date Variables

Literals that contain date variables and operations over dates retain the same operator. On the other hand, date variables are declared as integer variables and date constants are converted into number of milliseconds from January 1st 1970. By this, expressions over date variables are transformed into expressions from linear arithmetic over integer numbers.

Example 4. The literal

$$DOB > ToDate(‘1969\text{-}01\text{-}01’)$$

is transformed into

$$DOB > -31536000000. \square$$

2) Trasformations of Literals Containing String Variables

Literals containing strings are transformed into Boolean propositions through the following subsequently executed steps:

1. Each pair of different literals l_i and l_j is transformed into propositions p_i and p_j, respectively, and the formula (4) is extended with the conjunct
   $$p_i \Rightarrow p_j, \text{ i.e., } \neg p_i \vee p_j,$$
   iff both l_i and l_j contain operator LIKE and l_j can be inferred from l_i according to the following condition. Literal l_j can be inferred from l_i iff the following relation applies between right operand RO_i of l_i and right operand RO_j of l_j. Let s_i be the array of strings created by splitting RO_i by character ‘%’ and let s_{ik} be the k-th member of that array. Analogously, let us define s_j and s_{jk} for RO_j. If arrays s_i and s_j are of the same length and each s_{ik} is a substring of s_{jk}, literal l_j can be inferred from l_i.
2. For each pair of literals containing strings, l_i and l_j, if they are identical and
   2.1. neither of them processed in step 1, they are transformed into the same proposition p_i; or
   2.2. one of them is transformed into a proposition p_k in step 1, the other literal is transformed into the same proposition.
3. Each literal l_i containing a string variable and not processed through steps 2 and 3, becomes a proposition p_i.

Example 5. Let us define the following two check constraints, each of them containing only one literal:

$$i_1 = l_1\text{: } NAME \text{ LIKE ‘J\% DOE’ and}$$
$$i_2 = l_2\text{: } NAME \text{ LIKE ‘JO\% DOE’.}$$

Let us further assume that i_2 is a subschema check constraint while i_1 is the corresponding schema check constraint, and that we need to prove validity of

$$i_2 \Rightarrow i_1, \quad (5)$$

i.e. to prove satisfiability of

$$l_2 \wedge \neg l_1. \quad (6)$$

If we split right-hand operands of each literal i_k, $k \in \{1,2\}$, over character ‘%’, we obtain the following arrays:

$$s_1 = \{‘J’, ‘\ DOE’\} \text{ and } s_2 = \{‘JO’, ‘\ DOE’\}.$$

According to the first abovementioned step, since each member of s_1 is substring of the member of s_2 at the same position, it is concluded that l_2 implies l_1. Hence, each l_k, $k \in \{1,2\}$, is replaced with a proposition p_k and (6) is extended to the following formula:

$$p_2 \wedge \neg p_1 \wedge (p_2 \Rightarrow p_1). \quad (7)$$

Since (7) is an unsatisfiable formula, it is concluded that (5) is valid. □

3) Trasformations of Literals Containing IN Operators.

Literals containing IN operators are also transformed into Boolean propositions through the following three steps, executed in the given order:

1. Each pair of literals l_i and l_j are transformed into proposition p_i and p_j, respectively, and the formula (4) is extended with the conjunct
   $$p_i \Rightarrow p_j, \text{ i.e., } \neg p_i \vee p_j,$$
   iff right operand of l_i is a subset of the right operand of l_j.
2. If literals l_i and l_j are identical and
   2.1. neither of them processed in step 1, they are transformed into the same proposition p_i; or
   2.2. one of them is transformed into a proposition p_k in step 1, the other literal is transformed into the same proposition.
3. Each literal l_i containing a string variable and not processed through steps 2 and 3, becomes a proposition p_i.

Example 6. Let us observe the following two check constraint literals, belonging to the same check constraint implication formula:

$$l_1\text{: } X \text{ IN [1,3,5,7,9] and } l_2\text{: } X \text{ IN [1,5,9].}$$

Since [1,5,9] is a subset of [1,3,5,7,9], the first transformation step is applied to the two literals, where each l_k, $k \in \{1,2\}$, is replaced with a proposition p_k, and the check constraint implication formula is extended with the conjunct

$$p_2 \Rightarrow p_1. \square$$

C. Transformation of Check Constraint Implication Formula’s Negation into a SMT Language

Each SMT solver provides an input language for specifying a SMT problem and interaction with the solver. Also, a large number of the modern SMT solvers support the standardized SMT-LIB language ([14]).

As none of the existing SMT solvers can solve all problems, it is useful to check satisfiability of a logical formula with more than one solver. Therefore, we transform check constraints specifications into SMT-LIB language.

An input SMT-LIB file consists of three sections:

1. declarations of attributes and functions used in the set of clauses,
2. the set of clauses derived from the negation of the check constraint implication formula and
3. the command that starts satisfiability check.

Example 7. Let us observe the two check constraints from Example 1 and test if:

$$DepBudget > DepResearchBudget \text{ AND}$$
$$DepResearchBudget > DepConfBudget$$

implies

DepBudget > DepConfBudget.

For this purpose, a SMT-LIB file is created, with the specification of the check constraint implication formula's negation, as it is presented in Figure 4.

The first file section contains declarations of attributes referenced in the check constraints, given in the SMT-LIB syntax.

The second file section contains clauses that correspond to the negation of the implication formula of the two check constraints:

- *DepBudget > DepResearchBudget*;
- *DepResearchBudget > DepConfBudget; and*
- *¬(DepBudget > DepConfBudget).*

In SMT-LIB language, binary operators are given in the prefix notation.

The last section contains the command "check-sat" that starts SMT algorithm over clauses given in the previous file section. □

A detailed description of the SMT-LIB syntax may be found in [15].

VI. CONCLUSION

In order to maintain consistency and provide correct manipulation of data through information subsystems, the database subschemas have to be consolidated with the integrated schema. From the aspect of check constraints, consolidation means that each schema constraint that spans subschema data must have a corresponding subschema constraint which is equally strong or stronger. In this way, a subschema check constraint must imply the corresponding check constraint of the integrated database schema. We implemented an algorithm for testing check constraints consolidation and embedded it into *IIS*Case* tool.

We further concluded that the check constraint implication problem represents a SMT problem and consequently, should be solved by utilizing SMT solvers.

We also defined and implemented transformations of check constraint PIM specifications into the form and language understandable by SMT solvers. Each SMT solver can solve a subset of all possible SMT problems, but none of them can solve all of them. Therefore, by using the standardized SMT-LIB language, it is possible to utilize multiple SMT solvers to check satisfiability of a logical formula.

As a part of our future work, we will provide transformations of check constraint specifications into non-standard SMT languages, e.g. CVC ([16]), in order to extend the list of SMT solvers which can be integrated with *IIS*Case*. Also, we intend to extend one of the existing SMT solvers with the rules for handling operations with date, string and set variable, as it is described in Section V.B.

```
;declarations section
(declare-fun DepBudget () Real)
(declare-fun DepResearchBudget () Real)
(declare-fun DepConfBudget () Real)

;clauses section
(assert (> DepBudget DepResearchBudget))
(assert (>DepResearchBudget DepConfBudget))
(assert (not(> DepBudget DepConfBudget)))

;command for starting the satisfiability test
(check-sat)
```

Figure 4. Algorithm for subschema consolidation with respect to check constraints

REFERENCES

[1] I. Luković, P. Mogin, J. Pavićević and S. Ristić, "An Approach to Developing Complex Database Schemas Using Form Types", *Software: Practice and Experience*, vol. 37, no. 15, pp. 1621-1656, 2007.

[2] S. Ristić, "Problem Research of Database Subschemas Consolidation" (PhD thesis, in Serbian), University of Novi Sad, Faculty of Economics, Subotica, Serbia, 2003.

[3] I. Luković, S. Ristić, P. Mogin and J. Pavićević, "Database Schema Integration Process – A Methodology and Aspects of Its Applying", *Novi Sad Journal of Mathematics*, vol. 36, no. 1, pp. 115-150, 2006.

[4] I. Luković, "Automated Generation of Relational Database Subschemas Using the Form Types" (MSc thesis, in Serbian), University of Belgrade, Faculty of Electrical Engineering, Belgrade, Serbia, 1993.

[5] J. Pavićević, "Development of a CASE Tool for Automated Design and Integration of Database Schemas" (MSc thesis, in Serbian), University of Montenegro, Faculty of Science, Podgorica, Montenegro, 2005.

[6] C. Batini, M. Lenzerini, S. B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration", ACM Computing Surveys (CSUR), vol. 18, no. 4, pp. 323-364, 1986.

[7] I. Luković, A. Popović, J. Mostić and S. Ristić, "A Tool for Modeling Form Type Check Constraints and Complex Functionalities of Business Applications", *Computer Science and Information Systems* (*ComSIS*), vol. 7, no. 2, pp. 359-385, April 2010.

[8] N. Obrenović, S. Aleksić, A. Popović and I. Luković, "Transformations of Check Constraint PIM Specifications", *Computing and Informatics*, vol. 31, no. 5, pp. 1045-1079, December 2012.

[9] E. Mendelson, *Introduction to Mathematical Logic*, *4th Edition*, Chapman & Hall, London, United Kingdom, 1997.

[10] S. A. Cook, "The complexity of theorem-proving procedures", *STOC '71 Proceedings of the third annual ACM symposium on Theory of computing*, pp.151–158, New York, USA, 1971.

[11] F. Marić, "Formalization and Implementation of Modern SAT Solvers", *Journal of Automated Reasoning*, vol. 43, no. 1, pp 81-119, June 2009.

[12] L. de Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer", in *Formal Methods: Foundations and Applications*, pp. 23-36, Springer-Verlag, Berlin, Heidelberg, Germany, 2009.

[13] C. Barrett, R. Sebastiani, S. A. Seshia and C. Tinelli, "*Satisfiability Modulo Theories*" (book chapter), in A. Biere, M. Heule, H. Maare and T. Walsch, "Handbook of Satisfiability", IOS Press, USA, February 2009.

[14] D. R. Cok, "The SMT-LIB v2 Language and Tools: A Tutorial", available online: http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf, December 2013.

[15] C, Barrett, A. Stump and C. Tinelli: "The SMT-LIB Standard Version 2.0", available online: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf, December 2013.

[16] "CVC4 User Manual", available online: http://cvc4.cs.nyu.edu/wiki/User_Manual, December 2013