Mathematical Problems of Computer Science 23, 2004, 12–19.

Constraint Management in Database Applications via Checker Framework

Armen J. Asatryan

Institue for Informatics and Automation Problems of NAS of RA
UNICAD CJSC
e-mail armen.asatryan@unicad.am

Abstract

The paper presents concepts and ideas underlying an approach for constraint management in VLSI design database applications and designs flows. In this approach constraints are defined as scripts of EVA strongly typed scripting language and stored in meta–databases called *check catalogue*. Interaction of applications with check catalogue is supported by subsystem of DBMS called *Checker Framework*. Checker Framework provides facilities for creation, modification and deletion of the constraints and has several features that enhance the constraint management in database applications.

Keywords
consistency constraints, object-oriented database systems, scripting.

1 Introduction

In this paper, we propose an efficient technique for constraint management in VLSI database applications and design flows. VLSI design databases generally use an OO data model and have specific features that do not present in ordinary, non-engineering databases. They fill up progressively and iteratively, i.e. data is imported step by step (first comes technology data, then libraries, then design data) and usually design flows are defined on them.
In our approach we try to use the specificity of VLSI design databases for organization of constraint management in applications defined on them. The well–known techniques are unapplicable here because of many disadvantages and inefficiency. The usage of application–driven technique [8, 9, 10], where constraints are maintained by encoding them into applications, gives disadvantages like redundancy i.e. constraint must be specified in every method that might violate it, and understandability for semantics of constraints since they are encoded in statements of a programming language. Of course, in some cases precautions that have already been taken by the programmer inside class methods are the best solution, they eliminate the redundant run-time integrity checking and the user doesn't have to suffer from mysterious program failures. However, we can't make sure that precautions are indeed sufficient and these methods will never violate the constraints. Another disadvantage of application–oriented technique is that constraints are scattered in application and modifying of the constraints is a hard task.

Using of event–condition–action (ECA) rules facilities for maintaining consistency [2, 3, 5, 6, 7], gives us a possibility of disabling and enabling constraints, but modifying constraints is still a problem. Besides, this approach becomes useless, because of the absence of reacting behavior in VLSI design databases.
Unlike non–engineering databases, VLSI design databases inherit plain storage strategy, i.e. the engineering data stored in database is bound by weak constraints. Database itself can be consistent, but can contain inconsistent data with regard to applications/design flows. For example, assume that our database store VLSI design technology information containing layer data - *width, spacing, height*, etc. A reasonable constraint imposed on these data by application could be

The width of any layer must belong to the $[W_{min}, W_{max}]$ *range.*

Database itself can permit the layers with the unspecified width values. The layer that doesn't have its width value specified doesn't violate the database integrity constraint. This constraint imposes conditions driven by application. The introduction of precautions into members of database classes that prevent the importing of layers that don't violate the specified constraint is unacceptable and results in an unnecessary loss of information. Moreover, compliance to the constraints of one application can violate constraints defined by others.
We propose an approach of the *centralized constraint maintenance* via *Checker Framework* (CF). The main purpose of CF is to share constraint management between database and applications/flows and separate the constraint specifications from applications, thus to provide the independence of constraint maintenance to this approach.
In VLSI design database applications constraints are distinguished by high complexity and therefore usually encoded in application classes and methods. To manage the high complexity of constraints the powerful constraint definition language is needed and a scripting language seemed like an ideal solution. Unfortunately typeless nature of scripting languages could allow certain kind of errors to go undetected. EVA scripting language is our solution to this problem. It's a *strongly typed* language and has a capability of importing types of database objects, i.e. object types becomes accessible from EVA language. Application constraints can be encoded in EVA-scripts which will give us a possibility to easily do modifications on constraints as well as enable or disable them.
The reminder of the paper is organized as follows: Section 2 presents a quick review of EVA scripting language. How constraints are defined in EVA language is described in Section 3. In Section 4 we discuss the structure of Checker Framework and an issue of consistency enforcement by using CF. We end with concluding remarks in Section 5.

2 EVA Scripting Language

EVA scripting language is designed in the form of traditional programming languages [16, 15] and has constructs that are unique in it. Language is defined as a synthesis of programming languages C/C++/Java [18, 19]. An important and useful feature of EVA is its capability of importing the types of database objects, i.e. object types become accessible from EVA language. Such facilities are provided also by SWIG [13]. It takes C/C++ declarations and creates the wrappers needed to access those declarations from other scripting languages including Tcl, Perl, Java, etc. In contrast, EVA generates the wrappers and integrates them into EVA compiler.

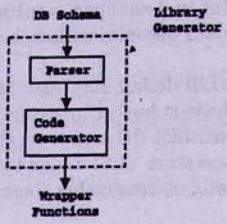

Figure 1. **Structure of Library Generator**

The type importing process is performed by the tool called *Library Generator* (Figure 1). At the core of the Library Generator is YACC [20] parser for reading class declarations of database schema along with some utility functions. To generate wrappers, Library Generator calls a dozen of functions to write the wrapper code of storing the extracted type information, like class names, types of members and member functions' arguments, return types of functions, values of enumerations, etc., into EVA compiler. Then, after language compilation, beside of built-in types provided by language itself, we obtain the extension of EVA called *exported* types. This makes EVA a *strongly typed* language, unlike Tcl [17], and helps manage complexity and detect errors at compile time.

Types, names, variables and arithmetic. Language syntax has C family languages style and implements reference semantic, like in Java [19]. Variable names and scope resolution rules are defined in the same way as in C family languages. The concise set of operations defined in EVA forms a subset of C++ operations. The arithmetic, logical, comparison and assignment operations defined for built-in types (boolean, integer, float and string) have the same semantics as in C++. EVA has a keyword **const** to indicate that the variable value should not be changed.

Language provides a conventional set of statements for expressing selection and looping: **if–else**, **while** and **for**. The **break** and **continue** statements can be used to control loop flow. The semantics of these constructs are the same as in C, only controlling expressions should have boolean type, and not an integral.

Enumerations are also supported in EVA and have the same semantics as in C++, but language doesn't provide facilities for definition of compound statements like unions, structures and arrays.

Functions. In EVA function definition has the same style as in C, except that it doesnt use extern declaration. Function name could be overloaded and cannot be declared inline like in C++. Default arguments, pointers to functions and ellipsis(...) in the function argument list are not supported.

EVA also supports the *reference type* construct to provide the way of passing references of the built-in type variables to functions for changing their values. The following example shows the usage of the references:

```
void func(int param1,int& param2, Cell param3)
{
  //change the param1 itself
```

```
    param1 = 10;
    //change the variable, referenced by param2 Pin
    param2 = 20;
    // change the object because it's a reference
    Pin newPin = param3.addPin( "VDD" );
}
```

Object creation/destruction Reference semantic requires usage of operator **new** for a new object creation, like in Java, and a garbage collector that will free unused memory. Semantic that is provided by EVA is very simple:

TypeName obj = **new**TypeName($args_{opt}$);

The following solution for garbage collection is used: EVA has a special keyword **local**, which tells the collector that it should free the memory before going out of scope:

```
int i;
for(i = 0; i < 100; i += 2)
{
    Polygon poly = new Polygon();
    // do something with poly
    ...
} // exit the block, all allocated polygons will be freed later
for (i = 0; i < 100; i += 2)
{
    local Polygon poly = newPolygon();
    // do something with poly
    ...
    // tries to free memory for poly before next cycle
} // exit the block, all allocated polygons are freed
```

I/O and preprocessor. Currently EVA doesn't support file input/output interface. As a mechanism of other file inclusion EVA has a statement **include** *string*, where *string* is a file name to include.

3 Constraints

We propose a unified modelling of the application constraints via EVA scripting language. All application constraints are defined as scripts in EVA scripting language - *checks*. The compound checks are built from existing ones using the **include** construct of EVA language. Script execution results true if the constraint holds on current database state. Otherwise, if constraint is violated, false is returned as a result of script execution. Advantage of encoding constraints into scripts is that violation handling of the latters is encapsulated in scripts. As a result, the application programmers can concentrate their efforts only on the violations of their own checks.

Here is a EVA-program segment that illustrates how the *width-restriction* constraint, defined in pervious sections, can be expressed in EVA langauge (note that TechLayer, TechLayerCollection and DB are *exported* types):

```
...
bool res = true;
TechLayerCollection::const_ iterator it = DB→layers().begin();
TechLayerCollection::const_ iterator end = DB→layers().end();
for(; it != end; ++it)
{
   TechLayer layer = *it;
   if(layer.width() < Wmin && layer.width() > Wmax)
   {
      //violation handling
      ...
      res = false;
   }
}
```

4 Checker Framework

One of the important aspects of constraint management is to provide the feature of independence of constraint maintenance [11, 12], i.e. ability to change constraints specifications without changing the application programs. To provide this feature for our approach we separate constraint specifications from applications using a meta–database called *check catalogue* which is a repository of all information about applications constraints. This gives us another advantage, since all constraints are placed together, their modification becomes an easy task.

The proposed *Checker Framework* is the subsystem of the DBMS and supports interaction of the applications/flows with the check catalogue. The check catalogue is the library of checks defined as EVA–scripts. Because of the iterative functioning behavior of the VLSI design databases, each application represents a working stage and consistency checks can be performed only once at a stage. DBMS provides interface for creation, modification and deletion of application checks. Each check is compiled and stored in the check catalogue. During the working stage the application requests CF for execution of corresponding check and in presence of violations the stage execution is stopped and violation handling operations are performed. Schematically interaction of the applications with CF is shown in Figure 2.

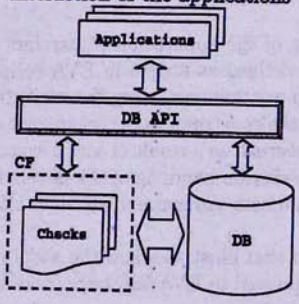

Figure 2. Constraint maintenance in database applications

Constraint management in design flows is based on interleaving the design process with *checkpoints*. At each checkpoint corresponding checks are executed and in presence of violations repairing actions are performed (Figure 3).

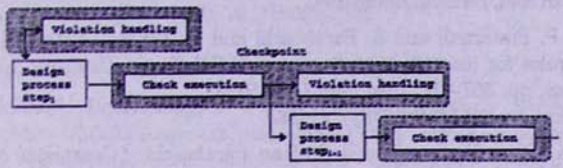

Figure 3. **Constraint maintenance in design flows**

5 Conclusion

In this paper, we have proposed a framework for constraint management of applications and design flows on VLSI design databases. Based on specificity of VLSI design databases, our approach provides effective means to maintain correctness of database applications and design flows.
In comparison to other approaches to constraints maintenance of database applications, our approach is characterized by two main advantages - first, constraints are specified declaratively using the EVA–scripting language and second, constraints are separated from applications using a CF, thus providing the independence of constraint maintenance.

6 Acknowledgments

The author is very grateful to Prof. Hrant Marandjian for his useful comments and suggestions.

References

[1] A. Asatryan, "An Approach for Constraint Management of VLSI Design Database Applications and Design Flows ", *Proceedings of the Conference on Computer Science and Information Technologies, Yerevan, Armenia*, September, pp. 375-378, 2003.

[2] N. W. Paton and O. Diaz, "Active Database Systems", *ACM Computing Surveys*, vol. 1, no. 31, pp. 63-103, 1999,
citeseer.nj.nec.com/paton99active.html.

[3] K. R. Dittrich and S. Gatziu and A. Geppert, "The Active Database Management System Manifesto: A Rulebase of a ADBMS Features", *Proceedings of the 2nd International Workshop on Rules in Database Systems*, vol. 985, Springer, pp. 3-20, 1995, *citeseer.nj.nec.com/dittrich95active.html.*

[4] U. Jaeger and J. C. Freytag, "An Annotated Bibliography on Active Databases", *SIGMOD Record* , vol. 24, no. 1, pp. 58-69, 1995,
citeseer.nj.nec.com/article/jaeger95annotated.html.

[5] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance",*Proceedings of the 16th VLDB Conference*, Brisbane, Australia, D. McLeod and R. Sacks-Davis and H. Schek, pp. 566-577, 1990, *citeseer.nj.nec.com/ceri90deriving.html.*

[6] S. Ceri and P. Fraternali and S. Paraboschi and L. Tanca, "Automatic generation of production rules for integrity maintenance", *ACM Trans. Database Syst.*, vol. 19, no. 3, ACM Press, pp. 367-422, 1994, issn 0362-5915, *http://doi.acm.org/10.1145/185827.185828.*

[7] Stefano Ceri, Piero Fraternali and Stefano Paraboschi, "Constraint Management in Chimera", *IEEE Data Eng. Bull.*, vol. 17, no. 2, pp. 4-8, 1994.

[8] V. Benzaken and X. Schaefer, "Static Integrity Constraint Management in Object-Oriented Database Programming Languages via Predicate Transformers", *Lecture Notes in Computer Science* , vol. 1241, 1997, *citeseer.nj.nec.com/article/benzaken97static.html.*

[9] H. V. Jagadish and X. Qian, "Integrity Maintenance in Object-Oriented Databases", *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman, (Los Altos CA), Vancouver* , 1992, *citeseer.nj.nec.com/jagadish92integrity.html.*

[10] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman , (Los Altos CA), Barcelona* , 1991, *citeseer.nj.nec.com/gehani91ode.html.*

[11] H. Oakasha and S. Conrad and G. Saake, "Consistency management in object-oriented databases", *Concurrency and Computation: Practice and Experience*, vol. 13, no. 11, pp. 955-985, 2001, *citeseer.nj.nec.com/296653.html.*

[12] Oakasha, H. and Saake, G., "Integrity Independence in Object-Oriented Database Systems", *Kurzfassungen — 10. Workshop "Grundlagen von Datenbanken", Konstanz (02.06.–05.06.98)*, no. 63, Universität Konstanz, Fachbereich Informatik, M. H. Scholl and H. Riedel and T. Grust and D. Gluche , pp. 94-98, 1998, *citeseer.nj.nec.com/oakasha98integrity.html*

[13] D. M. Beazley, "SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++", *4th Annual Tcl/Tk Workshop*, July, 1996.

[14] J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century", *IEEE Computer magazine* , March, 1998.

[15] A. V. Aho and R. Sethi and J. D. Ullman, "Compilers principles, techniques, and tools", Addison-Wesley, 1986.

[16] T. W. Pratt and M. V. Zelkowitz, "Programming Languages: Design and Implementation", Prentice-Hall, 1996.

[17] J. K. Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley, 1994.

[18] B. Stroustrup, "The C++ Programming Language (3rd Edition)", publ. Addison-Wesley, 1997.

[19] D. Flanagan, "Java In A Nutshell (2nd Edition)", publ. O'Reilly & Associates, 1997.

[20] J. R. Levine and T. Mason and D. Brown, "Java In A Nutshell (2nd Edition)", Addison-Wesley, 1992.

Տվյալների հենքերի կիրառական ծրագրերի սահմանափակումների մշակում ստուգիչների համակարգի կիրառմամբ

Ա. Ջ. Ասատրյան

Ամփոփում

Սույն հոդվածում ներկայացվում է սահմանափակումների մշակման մոտեցում երմեծ Ինտեգրված Սխեմաների Նախագծման (VLSI design) հենքերի կիրառական ծրագրերում եվ հոսքերում: Ըստ առաջարկվող մոտեցման՝ սահմանափակումները ներկայացվում են EVA տիպիզացված սկրիպտային լեզվի միջոցով եվ պահվում են ստուգիչների մետահենքում: Ծրագրերի եվ ստուգիչների մետահենքի փոխազդեցությունը իրականացվում է Ստուգիչների Համակարգի միջոցով, որը հանդիսանում է հենքի դեկավարման համակարգի ենթահամակարգ: Ստուգիչների համակարգը պաշ-տասշ-խաշ-նաշ-տու է սահմանափակումների ստեղծման, փոփոխման եվ ջնջման համար, ինչպես նաեվ ապահովում է միջոցներ՝ ծրագրերում սահմանափակումների մշակման արդյունավետ կազմակերպման համար: