CONSTRAINT VERIFICATION USING A CONSTRAINT ENGINEERING SYSTEM

A. Schäfer[a], J. Freuer[b], K. Hahn[a], W. Nebel[b], R. Brück[a]

andre.schaefer@universität-siegen.de

[a] *IMT - University of Siegen, Germany;* [b] *University Oldenburg, Germany*

[a] *http://www.rs.uni-siegen.de*, [b] *http://www.informatik.uni-oldenburg.de/*

Abstract

This demo presents a proof of concept of a meta-verification tool for the tool-independent definition and verification of constraints within the analog design flow.

1. Overview

The automatic consideration and verification of design constraints becomes increasingly important due to their continuous growth in number and complexity.
With the *Constraint Engineering System* (CES) we present a new verification method based on a unified representation of constraints. The Constraint Engineering System provides flexible, extensible, and multi-tool definitions of complex constraints and high order verification tasks. The CES does not replace existing verification and simulation tools. It rather offers a method of combining these tools for verification purposes [1].
The Constraint Engineering System is based on the approaches of Constraint Logic Programming (CLP). Describing constraints as predicates within Horn clauses [2] leads to a universal representation of constraints on an abstract, formal meta-level. To overcome semantical and representational differences between constraints from different sources a transformation model is defined that completely maps constraints into a universal constraint representation.
One of the key benefits of the CES is its extensibility by external verification and simulation tools. A tool can be integrated into the CES using an interface enabling the access to the data and the functionality of the tool. The interface translates the syntax and semantics of constraints as well as all verification task relevant design data to the internal clause-based representation. This mechanism provides a logical meta-level that links all connected tools together. Existing constraint information stored in a constraint management system can be utilized easily.

2. Constraint Engineering System

The Constraint Engineering System is based on the concepts of Constraint Logic Programming. CLP is an extension to logical programming languages like PROLOG that incorporate constraint resolution [3,4,5]. It has been developed in the preceding two decades from a branch of linear programming (LP) and basic concepts of artificial intelligence (AI).
The CES is a meta-verification-tool. Subverification tasks are delegated to external tools. The CES must ensure the possibility of a seamless integration of upcoming constraints. Therefore, one of the main targets of the CES is to provide a consistent representation of constraint data suitable for multiple tools. For this purpose, a formal description based on the Horn calculus (logic calculus) [2] was developed.

2.1 Architecture

The core of the CES consists of a CLP kernel as described by Jaffar in [3] (see Figure 1). Rules utilized by the CES form the knowledge base of the logic inference system. It is classified into a static and a dynamic part. The static part of the knowledge base is formed by rules that remain unchanged for all verification tasks. The dynamic part provides the design specific and therefore changing data and constraints.

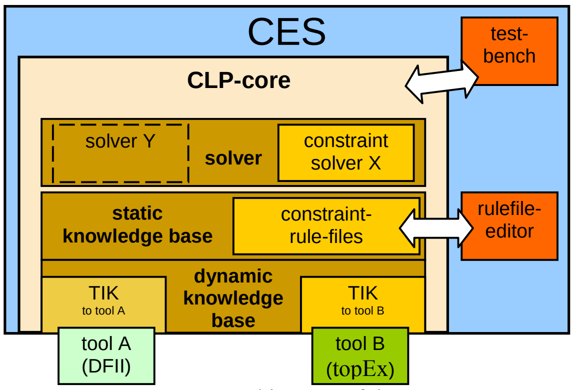

Figure 1: Architecture of the CES

The knowledge base is semantically divided into the following components:

Tool integration kit (TIK):
External tools export design data, constraints and verification capabilities which form the dynamic part of the knowledge base. The specific functionality of each tool is combined in a tool integration kit.

Constraint rule file and support rules:
The constraint rule file contains the set of possible queries. The constraint rule file is part of the static knowledge base of the CES. It depends on external tools and their support rules. High order clauses form so-called support rules. In this manner, support rules can be used as an abstraction of frequently performed queries. The support rules are also part of the static knowledge base. This mechanism enables an easy posibility to access the underlying knowledge base.

Both knowledge base parts are based on the same formal representation within the CES. Thus, the CES can access every knowledge aspect in a uniform way. This is essential for the multi-tool representation and handling of constraints. The rulefile-editor is no part of the CLP-core. It is an individual tool to creates and edit constraint-rulefile and rules of the dynamic knowledge base, which specify a TIK. The test-bench is a tool to review the CES, which uses the CLP-Core to solve given rules.

The following examples show the integration of a layout tool whereas the introduced clauses are compiled within the layout TIK. This kit exports clauses as part of the dynamic knowledge base in order to identify layout elements such as nets, polygons and pins. The rules use the identification scheme to uniquely address external tools and are defined as facts without a clause goal.

```
net(layout, NetID).
polygon(layout, PolyID).
pin(layout, PinID).
```

The rule *polygon* in this example holds true if there is a polygon within the design provided by the layout tool that has the id *PolyID*. Properties of pins (coordinates) and of polygons (coordinate lists) as well as their ownerships are determined by the following clauses.

These clauses do not require a tool ID since the tool can be implicitly resolved by the passed object ids (PinID, PolyID and NetID).

```
coordinate(PinID, (X, Y)).
coordinate(PolyID, [(X1,Y1), (X2,Y2),...]).
netPin(NetID, PinID).
netPolygon(NetID, PolyID).
```

The following clause *netLayout* is defined as static knowledge within the support rules stating that any net consists of pins and polygons. It compiles the given layout elements into tuples of the form (pinlist, polylist).

```
netLayout(NetID, (PinList, PolyList)) :-
    setof(Pos, (pin(NetID, PinID),
    coordinate(PinID, Pos)), PinList),
    setof(Pos, (polygon(NetID, PolyID),
    coordinate(PolyID, Pos)), PolyList).
```

Complex meta-verification can therefore be generically expressed without declaring a specific tool. Required tools are automatically selected by the CES from a list of registered tools. Generic verification tasks allow their application in similar verification areas and thus provide for the reuse of these tasks.

3. Demonstration

The CES currently includes TIKs to the Cadence Design Framework II (DFII), to a demo resistance extractor and to a demo net layout topology extractor.

For a star-shaped net parasitic point-to point resistances R_{C,P_n} exist between the net pins P_n, and the net center C. The following design constraint should be met:

All resistors R_{C,P_n} *within star-shaped nets must hold:*
$R_{C,P_n} \leq R_{max}$

The TIK that provides access to the resistance extractor (with tool ID resEx) supplies the clause resistance that computes the resistance between two coordinates (X1,Y1) and (X2,Y2) within a given net.

```
resistance(resEx,(PinList, PolyList),
    (X1,Y1), (X2,Y2), R).
```

The clause topologyClass provided by the topology extractor TIK (topEx) classifies a given net into one of the categories star, tree or mesh. The topology class star is additionally parameterized by the center (X, Y) of a star-shaped net.

```
topologyClass(topEx, (PinList, PolyList), Class).
```

To perform the verification task the verification clause *valStarRes* is added to the constraint rule file as part of the dynamic knowledge base to realize the meta verification task ($R_{C,P_n} \leq R_{max}$ within star-shaped nets). This verification task is therefore formally independent from the deployed tools.

```
valStarRes(NetID, PinID, Rmax) :-
    R>Rmax, net(_, NetID),
    netLayout(NetID, L),
    topologyClass(_, L, star(C)),
    netPin(NetID, PinID),
    coordinate(PinID, P),
    resistance(_, L, C, P, R).
```

The previously defined clause *valStarRes* can now be used for a verification with e.g. R = 5Ω using the verification query *valStarRes(N,P,5)*. The CES determines all resulting combinations of net IDs N and pin IDs P which fulfil the *valStarRes* predicate. This result states that every resistance $R_{C,Pn}$ of each obtained tuple N, P fails the verification requirements.

4. Conclusion

The CES is capable of processing verification tasks on a much higher level of abstraction than usually found in existing verification tools. The abstraction of constraints allows the processing of new classes of verification problems that previously could not sufficiently and comprehensively be addressed with conventional verification approaches. First tests of practical applications in analog system design tasks prove the power, flexibility, practicability, and potential of our approach.

In future we will focus our work on optimization strategies to further minimize unnecessary verification tasks and to parallelize independent verification tasks.

5. References

[1] J. Freuer, G. Jerke, A. Schäfer, K. Hahn, R. Brück, A. Nassaj, W. Nebel, „Ein Verfahren zur Verifikation hochkomplexer Randbedingungen beim IC-Entwurf, ANALOG, 2006

[2] A. Horn. On sentences which are true of direct unions of algebras. J. Symbolic Logic, 16:14–21, 1951.

[3] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) Language and System. ACM Trans. on Programming Languages and Systems, 14(3):339–395, July 1992.

[4] J. Cohen. Constraint logic programming languages. Commun. ACM, 33(7):52–68, 1990

[5] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, U.K., 1997

6. Acknowledgement

The present work was promoted in parts by the BMBF project LEONIDAS+ (conveyor sign: 01M3074).