A Generator for Graph-Based Design Representations^{*}

Ansgar Bredenfeld

GMD-SET, Schloß Birlinghoven, D-53754 Sankt Augustin, Germany, ansgar@borneo.gmd.de

Abstract

Tool integration services are an important means to make design tools inter-operate in a design environment. Today, the problem of fine-grained data integration is tackled from two directions. From the application domain using fixed design tool interfaces and internal forms based on specific design representations and from the database domain applying models and interfaces mainly based on semantic and/or object-oriented data models.

In our approach, we use concepts of both directions in order to further close the still existing semantic gap between a design tool and a framework database. We extend an object model comparable to existing ones by an abstract graph model which reflects structures and functionality occurring in various graph-based design representations. This extended model is used as basis for a new tool integration service. It provides a schema-driven code generator for implementations of graph-based design representations. In addition, presentation integration is achieved by providing a configurable graphical interface which directly offers visualization functionality and interactive access to fine-grained design data.

1. Introduction

Design environments consist of several design tools and services for data management and design methodology management [1]. Enabling and supporting interoperability of design tools in environments is the major goal of these framework services. Tool integration services have to offer mechanisms to easily plug tools into the environment. Tool integration may be classified in *data integration*, *presentation integration*, *control integration*, and *process integration* [2]. This paper is concerned with data integration but also considers aspects of presentation integration.

Data integration services provide tool interfaces between a common data repository of a framework and design tools. Data representations are needed as conceptual models of the data which is transferred via the interface and handled by the tools. The granularity of data representations may be used to further classify data integration in a framework.

^{*}. This research was supported in part by the commission of EC under ESPRIT contract 7364

Design data on a *coarse-grained* level are used as atomic entities in the context of design flow management. Coarse-grained design representations are often associated with textual design files expressed in specific description languages e.g. VHDL [3], EDIF [4], or Verilog [5]. Tool integration on this level is accomplished by tool encapsulation techniques utilizing so-called tool wrappers. They may be regarded as a more or less complex interface structure between an encapsulated design tool and the data repository of the framework. Design data transfer via this interface is usually controlled by a design flow manager. Often tool wrappers use temporary workspaces containing design files checked out from the data repository. Utilization of the fine-grained contents of design files is only possible after parsing done by the encapsulated tool.

Design tool algorithms operate on design data of *fine granularity*. Therefore, fine-grained design representations are suitable conceptual models for the data structures of design tools as well as for the tool interfaces providing design tools with design data from a database. In contrast to that, encapsulated design tools are only provided with design files. Their coarse granularity does not reflect fine-grained dependencies within and between design files. Recent research in the area of tool encapsulation [6] has proposed to split textual design files into text chunks in order to operate on design data of finer granularity. Nevertheless, real fine-grained tool integration remains the ultimate goal to allow for tight interoperability between design tools. Fine-grained design representations are a crucial prerequisite to allow for e.g. back-annotation and highlighting mechanisms within shared fine-grained design data. Interoperability on this level of granularity is not only needed within the scope of a single design representation e.g. a netlist, but also between several design representations. As only one example, imagine a system for interactive high-level synthesis e.g. [7] or [8] which needs back-annotation mechanisms between the original behavioural specification and a design structure synthesized from it in order to allow for interactive exploration and optimization of a synthesized design. Such systems need fine-grained links between the behavioural representation e.g. a control/data flow graph and the structural representation e.g. a netlist. In future, interoperability even between hardware and software representations will be an important issue of tool integration in hardware/software codesign systems [9].

An analysis of design representations shows that they are often based on more or less complex graph structures. This motivates our approach to identify some of their common structures and functionality needed by design tools for their manipulation. Since several data models for fine-grained tool integration exist [10,11,12,13,14], we use them as starting point for our model. We extend their modelling concepts with graph structures and functionality in order to offer a powerful tool integration service. This approach will further close the still existing semantic gap between database models and the often graph-based conceptual model of design tools. In addition, reuse of graph functionality will reduce the code needed to be written for integrating existing tools and especially for constructing new tools.

This paper is structured as follows. Section 2 gives an overview of related work. Section 3 briefly describes our abstract graph model. Section 4 focuses on a new tool integration service which allows to automatically generate implementations of and graphical interfaces for graph-based design representations. Section 5 gives a first case study on using a generated netlist representation to integrate a multi-level fault simulator, an existing Verilog parser, and a

graphical interface which is provided by our tool integration service. Section 6 concludes the paper with an outlook to future work.

2. Related Work

Standardized fine-grained design representations as basis for tool integration exist for netlists only. The CAD Framework Initiative, Inc. (CFI) has defined the Design Representation Programming Interface (DRPI) [15] which is a strongly typed interface between an integrated design tool and a framework database. Netlists are an important design representation based on graphs. Nevertheless, graph-based design representations also occur in other tool domains. Control flow graphs, data flow graphs, or state transition diagrams are only a few examples for them found in the high-level synthesis domain. They serve as conceptual models of internal forms in specific high-level synthesis systems e.g. [7] or [8].

Several data models for fine-grained tool integration have been proposed in recent years [10,11,12,13,14]. They are mainly based on object-oriented or semantic data models. The structure of design data is described in a schema. Design data stored in a framework database have to conform to this schema. Manipulation operations are provided to handle these design data. The models described in [13] and [14] use a small set of generic operations. Object types are passed as parameters to their interfaces. Therefore, type checking has to be done dynamically which is critical with respect to tool performance.

Although these data models made a first step to narrow the semantic gap between a simple internal model of a database and the complex conceptual model of a design tool, all of them lack explicit support of graph structures. Nevertheless, many conceptual models of design tools are based on more or less complex graph structures as shown before. In order to further close the still existing semantic gap between tools and databases, we extend an object model comparable to these existing models by an abstract graph model to allow for homogeneous modelling of complex graph-based design representations.

3. Abstract Graph Model

The object model we use as basis supports modelling concepts known from object-oriented programming languages like object types (classes), attributes (data members), inheritance (class hierarchy), polymorphism and additionally object identity known from object-oriented databases (OODB). Furthermore, we add concepts from semantic data models by supporting different types of relationships between object types (e.g. unidirectional and bidirectional references as well as containment relationships; each of them alternatively with one-to-one, one-to-many, or many-to-many cardinality). Language constructs representing these modelling concepts are used to express design data in a schema.

Operations allow for creation, deletion, navigation, and manipulation of design data conforming to the schema. We abstract these operations by a small set of formally defined generic functional primitives. They are represented in an intermediate form to be independent of a specific implementation base. This approach allows us to map functional primitives to either an object

manipulation language of an OODB or to efficient method implementations operating on C++ in-memory data structures.

Figure 1. Abstract graph schema

The language constructs of the object model are used to define the structure of our abstract graph model. It is given by the simplified *abstract graph schema* depicted in figure 1. A sample graph structure conforming to the abstract graph schema is illustrated in figure 2.

Figure 2. Sample graph structure

The central entity is a *component* having an interface structure consisting of a set of directed *ports*. Connectivity between components is modelled by a reference from each port to a

channel. Channels may reference many ports. They are the communication links between components. Signals may be transported via channels between components.

Abstraction is modelled by an *abstract component*. In general, abstract components provide descriptions of the interface structure and the behaviour of components derived from them. Two different abstraction mechanisms are supported. *Independent abstract components* (α, β) may be regarded as type descriptions of *atomic components* ($\alpha', \alpha'', \dots, \beta', \beta'', \dots$). Therefore, atomic components derived from the same independent abstract component have the same interface structure (α has 3 ports, β has 2 ports). These few object types are sufficient to model flat graphs of typed multi-port components connected by multi-channels. They are suitable to model e.g. flat netlists or flow graphs.

Additional concepts needed by graph-based design representations extend this basic graph structure. Grouping of components in a flat graph is done by disjoint *component sets* (A, B, C, D). They partition a flat graph. As a consequence channels are specialized into two disjoint sets of *inter-set channels* and *intra-set channels* (cf. figure 2).

The second abstraction mechanism is used to construct hierarchies. It allows to abstract a group of components (component sets) by a single *dependent abstract component* (a, b). This kind of abstract component is not an isolated type description as an independent abstract component but rather represents the interface structure of the abstracted component set. Structural hierarchies are constructed by deriving several *instance components* (b', c', c'') from dependent abstract components. Instance components have the same interface structure as their describing dependent abstract components.

Components are clustered in component sets which are in turn clustered on a higher level by *contexts*. This supports semantic clustering of graphs already in the model and is exploited to map graphs and their subgraphs to clusters in an underlying database.

Graph functionality is specified using the abstract graph schema as structural basis. Functionality includes e.g. definition of independent abstract components, graph construction and destruction by creating and deleting atomic components, connecting and disconnecting components, and arbitrary navigation in component graphs. Creation of dependent abstract components from component sets in conjunction with subsequent derivation of instance components is used to dynamically construct structural hierarchies. All transformation operations preserve integrity of connectivity between components.

In addition, functionality to support construction of design tools is provided, e.g. hierarchy transformations (folding, unfolding), signal flow analysis (levelizing), signal propagation, and behavioural descriptions of abstract components. A detailed description of this functionality is beyond the scope of this paper.

4. Model-Based Tool Integration Service

The abstract graph model was designed to improve fine-grained tool integration and to speed up tool construction. Therefore, we have to provide a process to make the abstract model usable as a tool integration service offering graph structures and functionality to a tool integrator or a tool constructor.

4.1 Design Representation Schema

More complex design representations consist of several interrelated graph-based structures. For example control/data flow graphs (CDFG) are used as behavioural representation of a design in the high-level synthesis domain. These CDFGs are a combination of two graphs - a control flow graph and a data flow graph. In order to be capable of modelling such structures we allow to use the abstract graph schema several times in a single schema. This mechanism may be considered an instantiation of the abstract graph schema on schema level. Therefore, we call occurrences of the abstract graph schema *graph schema instances*. Since each specific graph structure has its own interpretation of the abstract graph schema, we bind each object type of the abstract schema to a specific object type in a graph schema instances. Thus, each graph schema instance provides a specific graph structure with a set of well defined graph functionality.

Graph schema instances may be arbitrarily extended using the language constructs provided by the object model. Since graph schema instances are described in terms of these language constructs, object types, relationships, and attributes are simply added to the schema. Furthermore, object types may be specialized. The resulting schema containing several graph schema instances with arbitrary extensions and specializations is called a *design representation schema*. It is used as input for the code generator.

4.2 Code Generator

We have implemented a prototype of a new tool integration service based on the abstract graph model. Its overall architecture is given in figure 3.

The code generator is capable of creating implementations of design representations described by a design representation schema. It allows to map a schema to an object-oriented database and to an C++ in-memory data structure. The first is important for concurrent shared access to a design database, the latter crucial for efficient in-memory databases to gain acceptable tool performance. Since performance is regarded a key requirement especially for tool construction, we do not pass object types as parameters to a generic tool interface as done in [13] or [14]. Instead, we avoid dynamic type checking by generating a set of strongly typed operations for the object types of the schema. We use the generic functional primitives of the object model as templates for these operations. This avoids the performance degradation caused by dynamic type checking but keeps the advantages offered by a flexible schema description.

During the generation process, operations are mapped to in-memory C++ data structures with manipulation methods or to calls of the object manipulation language of an object-oriented database. We are able to easily adapt our schema to a schema of an OODB by providing

syntactical mappings of our language constructs and generic functional primitives to the data description language (DDL) and the data manipulation language (DML) supported by the OODB. Since our object model supports structural elements and operations of the ODMG object model [16], this mapping is more or less straight forward.

Figure 3. Overall architecture of the tool integration service

The specification of graph functionality is based on the functional primitives belonging to the abstract graph schema. This specification in intermediate form is taken as a kind of template for the methods implementations for each graph schema instance of the design representation schema. In addition, each extension to the schema results in further methods for their handling. All these methods are provided to the tool integrator or tool constructor as a generated C++ class library. We call these libraries *tool integration backplanes (TIB)*. A design tool only needs to include a generated header file and to link the generated library.

As an additional feature of our tool integration service, the code generator produces a wrapper description for the generated methods. This *method wrapper schema* is used to bind our methods as extensions to the interpreted language Tcl [17]. Thus, all generated methods are offered as Tcl commands to the tool integrator, tool constructor, or even the end user. This offers interactive textual access to a database containing design data conforming to the design representation schema.

4.3 Graphical Interface for Design Representations

Presentation integration has the ultimate goal to provide common interaction and presentation paradigms in the whole design environment [2]. Unified graphical data presentations and homogenous interaction mechanisms are suitable means to achieve this goal.

We consider presentation integration by providing a graphical interface for data visualization as an integral part of our tool integration service. Offering such an interface is a way to separate visualization functionality from the pure algorithmic kernel of a newly constructed design tool. Existing tools which are integrated using our tool integration backplanes may also profit from means for design data visualization as shown in our case study.

We have implemented an interactive graphical interface based on the extended Tcl and the graphical toolkit Tk [17]. It exploits the structure of the abstract graph model and offers a set of various display routines to visualize graphs. This approach allows to implement visualization functionality only once for the abstract graph schema and reuse it in each graph schema instance. Since a design representation expressed by a graph schema instance may prefer a specific visualization policy, we allow to control displaying of graphs. This is accomplished by generating a configuration file for the graphical interface. The tool integrator or constructor may change its default settings to achieve a specific visualization for each graph schema instance.

Each object of a graph structure is associated with a pop-up menu which offers the set of methods defined on the corresponding object type. Selection of a menu entry directly issues the corresponding method on that object. The tool constructor or tool integrator may add entries to this pop-up menus. Actions executed after selection of added menu entries are specified by scripts in extended Tcl. This allows to arbitrarily extend the graphical interface to the needs of the tool integrator or tool constructor.

The effort for getting interactive graphical access to a specific design database is now reduced to changing and extending the generated default configuration file of the graphical interface. The configured graphical interface then serves as an interactive tool to the design database. It allows for data input, tracing of method invocations, as well as querying the database.

5. Tool Integration Case Study

In a first case study, we integrated a state of the art multi-level fault simulator [18] with an existing Verilog [5] parser. We define a design representation schema for hierarchical netlists using a single graph schema instance. The entities of the netlist are bound to object types of the graph schema instance. For example an instance is modelled either by an atomic component (instance of a leaf cell) or an instance component (instance of a composite cell). Composite cells are modelled by a component set with a dependent abstract component. The first models the aggregation aspect, the latter the abstraction aspect of a cell. By this, semantically different aspects are explicitly separated. Nets are mapped to channels.

This design representation schema is fed to the code generator which produces a tool integration backplane from it. The resulting class library is linked to the fault simulator and to the Verilog parser. The mapping of the internal data structure of the simulator to the generated netlist

representation was straight forward since both are essentially hierarchical graph structures. We extend the internal data structure of the simulator to store back-annotation references for each instance in the netlist to allow for fine-grained interoperability. The simulator is now able to use the graphical interface which has been configured to serve as a visualization tool for hierarchical netlists.

The source code of the fault simulator and the Verilog parser had to be extended by calls of generated methods and some additional mapping code. The whole integration took only a week. The extension of the simulator data structure by back annotation references increases the memory requirements of the simulator by approx. 20% but allows for easy fine-grained interoperability. The runtime behavior of the simulator is not affected by this integration technique.

6. Conclusion

Our approach to fine-grained tool integration extends existing data models by an abstract graph model which defines common structures and functionality needed to handle graph-based design representations. In this, our model is a significant extension to the expressive power of existing models.

We implemented a tool integration service based on this model that automatically generates implementations of design representations from a schema description. Since we consider presentation integration as an integral part of tool integration, we provide a configurable graphical interface which serves as a flexible interactive visualization tool for generated graph-based design representations.

Currently, we are working on a second case study applying our model and service to tool construction. We are implementing a high-level synthesis algorithm using a generated design representation for high-level synthesis. We add two more graph schema instances for control and data flow graphs to the netlist schema used in our first case study. This will allow for fine-grained interoperability between the synthesis tool and the already integrated simulator.

References

- [1] Barnes, T.J., D. Harrison, A.R. Newton, R. L. Spickelmier, "Electronic CAD Frameworks", *Kluwer Academic Publishers*, Boston, 1992
- [2] Thomas, I., B. A. Nejme, "Definitions of Tool Integration for Environments", *IEEE Software*, March 1992
- [3] *IEEE Standard 1076*, VHDL Language Reference Manual, 1993
- [4] Electronic Industries Association. Electronic Design Interchange Format Version 3 0 0 Reference Manual, 1993, *EIA Standard EIA-618*
- [5] Thomas, D.E., P. Moorby, "The Verilog Hardware Description Language", *Kluwer Academic Publishers*, 1991

- [6] Schettler, O., S. Heymann, "Towards Support for Design Description Languages in EDA Frameworks", *Proc. of ICCAD*, November 1994
- [7] Wu, A.C.H., T.S. Hadley, D.D. Gajski, "An Efficient Multi-View Design Model for Real-Time Interactive Synthesis", *Proc. of ICCAD*, 1992, pp. 328-331
- [8] Ang, R.P., N. D. Dutt, "Equivalent Design Representations and Transformations for Interactive Scheduling", *Proc. of ICCAD*, 1992, pp. 332-335
- [9] *Proc. of the Third Int. Workshop on Hardware/Software Codesign*, September 1994, Grenoble, France
- [10] Sim, M.N., P.M. Dewilde, "An Object-Oriented Persistent Database Interface for CAD", *Proc. of EDAC*, 1990, pp. 363-367
- [11] Van der Wolf, P., R. van Leuken, "Object Type Oriented Data Modeling for VLSI Data Management", *Proc. of the 25th Design Automation Conference*, 1988, pp. 351-356
- [12] Heijenga, W., U. Jasnoch, E. Radeke, "DaDaMo - A Conceptual Data Model for Electronic Design Applications", *Proc. of EDAC*, 1992, pp. 394-398
- [13] Schettler, O., A. Bredenfeld, "BEPP0: A Data Model for Design Representation", *Proc. of EuroDAC*, September 1993, pp. 378-382
- [14] Filer, N., M. Brown, Z. Moosa, "Integrating CAD Tools into a Framework Environment Using a Flexible and Adaptable Procedural Interface", *Proc. of EuroDAC*, September 1994
- [15] "Design Representation Programming Interface: Electrical Connectivity", Version 1.3.0-050694, *CAD Framework Initiative*, Inc., Austin, USA, 1994
- [16] Cattell, R.G.G., "The object database standard : ODMG-93", *Kaufmann*, San Mateo, CA, 1994
- [17] Ousterhout, J. K., "Tcl and the Tk Toolkit", *Addison-Wesley Publishing Company*, Reading, MA, 1994
- [18] Meyer, W., R. Camposano, "Fast Hierarchical Multi-Level Fault Simulation of Sequential Circuits with Switch-Level Accuracy", *Proc. of 30th Design Automation Conference*, 1993, pp. 515-519