The Software Support for a Structural Synthesis Approach of Analog Circuits

Frank Grützmacher and Jürgen Kampe
Ilmenau Technical University, Department of Electronic Circuits and Systems,
D-98684 Ilmenau, Germany

Abstract

In this paper we discuss the use of new software approaches and technologies to build a tool which supports the analog high level synthesis outlined in [1]. Together with the audience we want to discuss the use of the so-called Enterprise Application Server (EAS) and distributed components. EAS systems are currently one of the hottest topics in the software area. We will shed a little light on some of the current buzzwords and analyze our requirements to see whether or not the use of the different technologies can help us to build software support for complex analog synthesis. What benefits or pitfalls can we expect in using them?

1 Introduction

The interfaces of electronic information processing ICs contain analog modules for performing data and signal conditioning. Great efforts have been made to enable high-level design methodologies to increase the design productivity, even for analog system components.

In coincidence with the system design terminology, a directed transition from the behavioral to the structural domain of the Y-diagram is called structural synthesis [2].

In the late 1980’s, the structural synthesis of analog blocks was based on architectural selection [3][4]. These tools were unable to overcome the specifications inherent structure.

In connection with hierarchical decomposition, design automation tools became more flexible in the early 1990’s [5]–[8]. These design tools are implemented on the basis of an expert system. Hierarchical decomposition gave the first strategy for a structural reorganization of the design with both the specification and the implementation belonging to the same level of abstraction from the data processing point of view.

Next, formal techniques were introduced to the process of structural synthesis [9]. Formal techniques enable design approaches covering different levels of data abstraction. This gives a second and more general way for the structural reorganization of the design by introducing multiple cycles of behavioral modeling, transformations of behavioral models, and structural refinements.

[1] introduces a design strategy which is based on a multi-phase design flow, separating the design of the data flow and the design of the electrical signal flow. All steps for the data flow design belong to the functional analog synthesis and all steps for the energy flow design belong to the electrical synthesis. For each synthesis phase, levels of data abstraction are defined and appropriate design representations are introduced.

Based on this design strategy, the implementation of the synthesis tools is given here, combining rule based and pattern recognition approaches on the one hand, and mathematical, formal approaches on the other, in an open, experimental design system.

2 The Analog High-Level Design Approach

2.1 The Design Flow

A multiphase design flow enables us to split the overall synthesis process into two phases: An analog functional synthesis that is oriented towards information processing, and an electrical synthesis oriented towards analog circuit operation.

In the function driven design phases, structural selection and decomposition is performed on the basis of the system’s or circuit’s function and on the basis of classified properties of the data or the signals, corresponding to the level of abstraction. Thereby, only main principles are to be taken into account; more complex structures are established by means of formal methods.

The sizing-oriented design phases begin with symbolic relationships, expressing functional relationships between the elements. According to the level of abstraction, data or signal parameter values are considered, derived from the design specifications. In addition, sizing-dependent structural transformations are applied. These transformations take into account the dependency of structural selection on the specific sizing and these drastically reduce the number
of alternatives that have to be considered for each structural selection step. Fig. 1 gives an overview on the design steps.

Formal representation of structural and behavioral design primitives defines classes of design primitives at a given level of abstraction.

Among the design primitive classes, formal relationships between the design representations are established: behavioral terms $H_1, \ldots, H_n$ models an structural aspect $r$, a behavioral term $H_k$ may be derived from a set of behavioral terms $H_1, \ldots, H_n$ by applying a transformation rule, an aspect $r_k$ refines one or more behavioral terms $H_p$, and an aspect $r_k$ is hierarchical decomposed by a set of aspects $r_1, \ldots, r_m$.

On this basis, the transition from a higher to a lower level of abstraction is described. The design knowledge is given by design primitives, and by design algorithms, specified by the formal relationships between the design representations.

Beside these structured divisions of design knowledge, each level of data abstraction enables typical formal approaches to design manipulation.

### 2.2 The Requirements

This new synthesis approach leads to the following requirements for the software implementation:

- At each level of data abstraction, the design is represented by design graphs in the structural and in the behavioral domain. We need a way to enable the user to graphically control the design flow and to specify the input of the different abstraction levels.
- All the different tools should share the same understanding of the design data. Therefore we have chosen a specialized and interpreted graph as underlying data structure for each individual tool.
- In analog circuit design, specialized modeling approaches require unique data representations at different levels of abstraction. A flexible management of design graph types is needed.
- Formal design manipulation requires a mathematical system, allowing the description of the design step in a way the designer is familiar with.
- Also, the knowledge based parts of the design approach require efficient data management of the actual design data, the design primitives, and the design steps.
- We cannot keep all design data in memory, but the data which describes the various design levels are often closely related to each other. This forces us to provide efficient persistent storage of the design data.
- In general, many design alternatives have to be evaluated, and thus many data need to be accessed and modified during the design process. We want to allow the toolset to be used not only on a single machine, but also on a cluster of machines or a multiprocessor box, to perform the evaluation of design alternatives in parallel.

### 3 Design System Architecture

#### 3.1 Software Components

In Fig. 2, it is shown how the different parts of our architecture interact with each other, and the services we have in our system.

Using an architecture relying on the components we are using, the principle idea is component based software development, which means we can build a complex software system of small software entities, each individually performing a single task. Components have been tested, are robustly
constructed, and their usage and interface are well defined. An approach like this reduces the time required to build a complex system and the overall complexity of the project is reduced. The implementation details of the pieces are hidden for the purpose of understanding the system basics. In our case we have four main components:

- the visualization using a Java2D based Graph Editor,
- the graph search engine, which allows to search for already synthesized design data, the design primitive knowledge,
- the design entity storage which is an Enterprise Application Server,
- the synthesis controller which has been - up to now - a Mathematica process.

All these components interact with each other, and because of the heterogeneous character of components, the communication layer of choice is CORBA. In the next section we will discuss the underlying data structure of all our tools. We also transmit our data using this structure.

3.2 Graph Model

As we can see, our graph model differs from ordinary\(^1\) graphs in the way, that we have ports which are used to connect the edges on the nodes. These ports can have certain properties; the same is true for nodes and edges. Fig. 3 shows a reduced version of our graph model.

In the very beginning, we defined the persistent data format for this graph. By the use of the parser, we can save and restore the graph. Nowadays we would base the data format on XML (Extensible Markup Language) and check the consistency of the information using XML schemata [10].

In our toolset one part – the GraphBuilder – is responsible for doing this using the MetaInformation of the graph.

You could regard this MetaInformation as the Meta-Model of the UML. It defines the abstraction levels we have, the figure types and the types of connections and the ports being allowed in the different abstraction levels. Further, we need a check for the use of properties for each graph part.

The properties need to be human readable, able to be specified, and we have to be able to search for a given property.

3.3 The Synthesis Controller

As its name implies, this part of the system initiates and controls synthesis steps.

Up to now this process is implemented in Mathematica. Due to the support of symbolic expressions, Mathematica fits very well into this synthesis approach. However, we cannot store, manipulate, and forget huge amounts of data inside Mathematica, as is required.

So we provide a kind of CORBA access layer via the JLink\(^2\) interface for Mathematica. In this fashion, we can control the other parts of the system outside Mathematica. Using the JLink interface of Mathematica, we can work in two ways: a) interact with a Java Classes inside a JAVA VM, and b) directly communicate via CORBA with external processes.

---

\(^1\)By ordinary graphs we mean graphs containing only nodes and edges.

\(^2\)see http://www.wolfram.com/solutions/mathlink/jlink/
Choice a) enables to directly access the MetaTypeInformation and the Java based graph classes from within Mathematica. If, for example, the synthesis controller initiates a search in one synthesis library, the Mathematica design representation is transformed using the Java GraphBuilder classes into our Java Graph Model, which then can be used by the Java classes for firing a search on one synthesis library. Fig. 4 shows the interaction as UML sequence diagram.

Figure 4. Interaction between Mathematica and the synthesis library SynLib

As you can see, the VM machine will only be instantiated once. All subsequent calls will go through the external Java classes; the GraphBuilder and the Wrapper around the remote calls to the SynLib. For further interactions this saves the cost of starting up a new JVM.

3.4 The Synthesis Library

The synthesis approach requires the search for existing knowledge which is stored in an object oriented database in the graph format. The component which provides access to this database is called the synthesis library (SynLib). It provides search facilities for locating entries which contain the provided graph as a part. This means the semantics of the search requests are: Does any graph exist with the same type and with the at least same number of nodes (as well as the same types) containing the same number of edges, all with at least the same attributes as the graph provided? Give me a reference to these matching graphs, together with a table showing the correspondence between the nodes of the provided graph to the nodes of the graphs found.

This is a very difficult question, not only because our graph model contains ports, and the structure of the three parts – right port to the right node, connected with the right edges – must fit. Besides this, we have also certain properties which are less easy to find, such as range values. The SynLib can also find nodes with a fixed value property, when the search value is a range type. Figure 5 provides a more detailed view on the internals of the Synthesis Library.

Figure 5. Internal structure of the SynLib

The implementation follows a three tier approach. Tier 1 is the CORBA (Common Object Broker Architecture [11]) request layer, in which we encapsulate the usage from other components via CORBA IIOP (Internet Inter-ORB Protocol). We can use any CORBA 2.3 compliant ORB, but are currently using ORBACUS 4.0.5.

The 2nd tier contains the Business Logic, meaning in this case all the functionality to prepare the database entries and to control and execute the search queries. Our SynLib also stores the results for already finished queries, so we don’t need to store search results in another component. The search controller can control several search engines, which can run on independent workstations. A precondition for this scenario is that the ODBMS (Object Database Management System) supports this, which is not allowed by our low cost, freeware solution with ObjectStore PSE as the underlaying database. For our local final system we will shortly switch to Versant, which is both – multithreading and multisession capable.

So later versions will also improve the usage of the Portable Object Adapter in the CORBA layer, and allow us to execute independent search requests in separate threads. This will make the SynLib more scalable, but requires a more professional database.

Using Versant, we also want to attempt to improve the search itself. Currently the search engine works on 12 indices, which are rebuilt each time a new graph is added into

\footnote{new servant adapter introduced with CORBA 2.3}
the synthesis database. Using this index, the search subdi
divides into three phases:
- phase 1 - preselection using the indices for existing graphs,
nodes, ports and edges
- phase 2 - choice of a starting node and iteration over pres-
elected graphs from phase 1
- phase 3 - a tree search over all possible links beginning
from the starting node

The problem is, that for phase 3 we need to instantiate
the graph in memory. One future task is, to investigate
whether we can perform this with the ODMG 3.0 Object
Query Language or with the more vendor specific query
languages to search faster within the ODBMS. As the per-
formance of the SynLib is crucial for the whole Synthesis
approach, this needs further analysis.

3.5 Concurrent Access to Design Data

Our aim is also to build in everything which is necessary
to allow parallel User Interaction and parallel tool opera-
tion. To allow this, we need to deal with distributed trans-
actions.

Transactions give us a boundary around a set of opera-
tions. We can associate a set of resources and operations
with a particular transaction. For example, to perform a
more complex synthesis step can mean to modify design
graphs and data in various abstraction levels. These data
are stored in a persistent way, because they are held over a
single session. So we need to reactivate these data.

From a programmer’s perspective we will use transac-
tions to bracket code that should be executed or fail as a sin-
 gle unit. This characteristic of transactions is called atomic.
The set of operations will succeed completely or fail. We
cannot leave our design data in an inconsistent state with-
out knowing from where the breaks come.

We will simplify this by using Enterprise Java Beans
(EJBs). The aim of the Enterprise Java Beans specification
is to define — analogous to Java Beans — an interface for
server side components rather than application object-like
normal Java Beans.

Building larger distributed applications, sooner or later
system level services are required such as transaction man-
gement, security, client connectivity, and database access.
Using the Enterprise Application Server effort can be fo-
cussed on building the server side application logic. This
logic can be used in reusable server side beans, the so called
Enterprise Java Beans, whereas the EAS functions as both
— the middle-tier server, which provides the functionality
on the stored data and the datastore for the many different
data, which need to be held during a long running design
cycle.

Let us describe the role which a EJB can play in our ex-
ample application. I. e. we need as part of a synthesis step
a kind of merging functionality, meaning part of of a design
graph at a certain level which must be modified through a
graph which has already been saved as design knowledge in
a synthesis library for a special area.

In Fig. 6 it is shown how different EJBs interact with
each other and which services of the Enterprise Applica-
tion Server are used. The graphical view, or rather the user,
drives the action. In our case a session bean is used when
the server should provide a particular service to the client,
such as searching for a model in our synthesis library. Us-
ing the underlying entity bean which holds the data graph of
the graphical, the current graph is modified in a concurrent
way and saved in the DBMS used. We don’t refer to any
particular DBMS system, or specify if it is a RDBMS or a
OODBMS. For the EJB it only matters that a persistence
service is available in the EAS (in our case Visibroker IAS
with either JDataStore or a wrapper for Versant).

The task of the session bean is to send a search request
to the SynLib. This search request can be i. e. “give me the
possible realization principles for feedback using this and
this device with the specified parameters”. The SynLib will
reply with a number of results. Note that we maybe can
also embed SynLib as another EJB in the same server. After
performing both steps 1 (searching) and 2 (getting a single
result), the Session bean has to operate and merge with the
chosen result (model) on a stored or shared design graph.
Therefore it performs a task for the client using an entity
bean.

The advantages we wish to emphasize of using EJBs in
this simplified scenario are; first, we do not need to worry
about transactional issues relating to Entity Bean (which
may be shared); second the database handling which re-
lates to the Entity Bean save, update and activate is handled
by the Enterprise Application Server and therefore hidden
from the programmer. Or rather, in the case of “Container-
managed entity bean” the container is responsible for han-
dling the persistency of the data.
3.6 The Design Data Server

As already discussed in the last chapter, the Design Data Server will hold the data inside transaction aware Entity Beans, and we can provide tools which perform certain tasks on this design data. Using EJBs, we can plug in components at runtime using the whole infrastructure of the Enterprise Application Server. Fig. 7 shows which role the EJB-container plays in our architecture.

Figure 7. An Enterprise Application Server as design data server

As already stated, we want as much parallelism as we possible in performing the synthesis steps, as the amount of data which has to be modified can be very large. One problem will arise. The EJB specification explicitly forbids the usage of threads inside EJBs. So how can we then start modifying the data in parallel? Our solution for this is a collocated CORBA – EJB server.

What this means: We put a multithreaded CORBA server and an Inprise IAS container in one process. The CORBA server will then act via its multiple threads as a multiple client to the EJBs inside the container. The disadvantage of this approach is that the usage of transactions needs to be explicitly put into the code of the CORBA server. We need to use transactions via the CORBA Object Transaction Service (OTS). Inprise IAS provides an implementation for the OTS called ITS. Using ITS we can participate with the internal transaction of the EJBs from the outside (the CORBA servant).

4 Results and Conclusions

In this paper, we have presented a highly heterogeneous distributed toolset for the software support of analog high level synthesis.

We have shown how we tried to fulfill the requirements of the synthesis approach for the various components. Us-