Integrating CAD Tools into a Framework Environment Using a Flexible and Adaptable Procedural Interface

Dr. Nick Filer Dr. Mike Brown Dr. Zahir Moosa

Department of Computer Science, The University of Manchester Oxford Road, Manchester, UK, M13 9PL

Abstract

Methodologies for integrating tools into CAD systems and CAD frameworks are examined. A fine grained procedural methodology for supporting inter-operability is introduced. This interface is based around a fairly small set of generic procedures. Flexibility is obtained by permitting the interface procedures to interpret conceptual models of the data. A mapping metaphor is introduced as a way of extending the flexibility and adaptability of the procedural interface.

1 Introduction

The continuing development of CAD environments and the current move towards the development and usage of framework technology is highlighting the need to support inter-operability among tools and systems. Inter-operability is dependent on the interface between components of an environment which are required to work together. For many CAD environments inter-operability is complicated by the need to frequently adapt to changes in the data representations used by applications and system components [1].

Tools are made to inter-operate through integration into a common environment. Where communication between tools is rare and only involves relatively small quantities of data, an ad hoc interface between the tools can suffice. However, in modern CAD systems communication between tools is frequent and involves very large quantities of data. Therefore, an ad hoc solution is not sufficient. In addition, the complex nature of the data and the fact that tools from different sources are frequently used to process the data means that attention must be paid to the structure and semantics of the data.

There are two extremes of interface which will be referred to here as coarse and fine grain. Some differences between coarse and fine grain interfaces are shown in Figure 1. Coarse and fine grain interfaces may be distinguished by examining the point at which the data is interpreted. A coarse grain interface usually delays interpretation of the data until after the data has been transmitted. Thus, typically large blocks of data are transmitted at one time. Much of this data may not be required by the application which consumes the data. A fine grain interface usually interprets the data prior to transmission. Therefore, the application need only receive the subset of the data which it specifically requires for its task.

The main difference in efficiency between coarse and fine grain interfaces occurs because the first does not interpret the data whilst the latter does. Not interpreting data permits coarse grain interfaces to transmit large bodies of data constrained only by the speed of the physical transmission devices. However, applications must then interpret the data in order to recover those entities which are of use to the application. Fine grain interfaces only transmit the specific data required by an application because the data is interpreted by either the data repository or by the interface. Therefore, in theory, it should be possible for fine grain interfaces to perform at a similar level of efficiency to coarse grain interfaces if caching and other optimisation techniques are utilised.

This paper describes an approach to fine grain interfaces which is designed to minimise the costs of changes to the representation of the data on either side of the interface. The interface is designed to be flexible and adaptable. This means that it is capable of transmitting any representation of data and adapting quickly to changes in the format of the data. Wherever possible, changes in the format of the...
data on one side of the interface are masked from the other side. This reduces the need for expensive modifications to applications, leading to greater flexibility in co-operative environments.

The paper starts by describing methodologies (coarse and fine grain) for integrating CAD tools together into systems environments and specifically into CAD frameworks. The design and implementation of a flexible procedural interface called “GPIC” (Generic Procedural Interface for CAD) [2] is described. An extension to the interface to provide greater flexibility and adaptability is then introduced. This extension enhances the ability of the interface to mask changes in schemas on one side of an interface from the other side. A generic methodology for supporting schema evolution and intra-schema mappings is used. The conclusions summarise the current status of the prototype implementation and describe further work to be carried out in the near future.

2 Tool Integration Methodologies.

Integrating different tools into a system or framework environment requires management of:

- The invocation, runtime support and termination of the tool.
- The user interface of the tool.
- The data requirements of the tool both as a consumer and as a producer.

Each of the above requirements of integration can be achieved in a variety of ways. Operating systems such as Unix\(^1\) provide the basic facilities needed to control a tool, to provide file based input plus output and usually a simple form of user interface. CAD frameworks such as JESSI-COMMON-FRAME, CADENCE, NELSIS and POWER-FRAME provide additional facilities to manage sets of integrated tools which may or may not be designed to work together in an integrated environment. These additional facilities provide amongst others:

- Management of complex projects.
- A common style for user interaction with both the framework and tools.
- Controlled access to the design data.
- Controlled invocation of tools.
- Facilities for capturing and responding to the data produced by tools.

The rest of this paper concentrates exclusively on the data integration of CAD tools into framework environments. The other requirements of integration are beyond the scope of this paper. Sections 2.1, 2.2 and 2.3 discuss the different types of integration methodology which are commonly used.

2.1 Encapsulation.

Multiple tools (and sometimes even complete CAD environments) can be integrated into a coherent system by a technique which is often called “encapsulation” [1]. Normally, encapsulation occurs at a very coarse level of granularity of data. For instance, the files read and written by a tool may be encapsulated by storing them within the framework’s database. The data in the files is not interpreted by the framework in any way. The framework has no conceptual representation (schema) of the structure and content of the files. The framework simply acts as a file management and control environment. It is possible to encapsulate a complete CAD system by using the framework to control access to the files that make up the CAD system’s database. An encapsulated tool can be invoked by releasing the files required from the framework and then starting the tool via a tool specific “wrapper” (often consisting of a command script) whose task is to ensure that the tool’s environment has been correctly initialised. On completion, the wrapper carries out any required termination actions for the tool. The framework can then capture the files of data back into its managed environment.

Tool encapsulation provides many facilities which enable greater control of design processes to be attained. However, encapsulation does not in itself facilitate the transmission of data between tools and other systems which use incompatible data representations.

2.2 Exchange of Data Using Languages.

The problem of sharing data between tools, and, between tools and the framework can be viewed at two levels:

- The physical level. This level is concerned with the mechanisms whereby data items are transmitted from place to place. This paper is not concerned with the exact mechanisms used but with the interface to these mechanisms.
- The conceptual level. This level is concerned not with the transmission of data items but with the structure of the data (described by a “data model”) and with the information content of the data (described by an “information model”). The interface provides a mechanism for the transmission of data between tools. The interface must maintain the semantics of the data which is passed through it.

At the data level, exchange of designs is often achieved using specially designed languages. Some languages, such as VHDL [3], are designed for use by both designers and CAD tools. Designers can describe their design using VHDL or they can use tools (e.g. for schematic capture, which are able to generate VHDL code. EDIF [4] is a language which is not designed for use directly by designers.

---

\(^1\) Unix is a registered trademark of UNIX System Laboratories, Inc.
but by design tools and systems. EDIF provides a language format for the interchange of electronic design data between tools, CAD systems and manufacturers. In order that the semantic interpretation of a design should not change when it is transmitted via a language, these languages are becoming standardised.

Tools and systems which incorporate a reader and writer for standardised interchange languages can use these language formats to provide files of data which can be interpreted and understood in other environments. These languages, therefore provide another mechanism for the coarse grain exchange of design data. However, unlike systems based only on encapsulation, the languages permit both parties to interpret and understand the data.

2.3 CFI DR PI

The CFI DR PI Version 1.0.0 [5] is a standard for the fine grain interchange of some types of CAD data (namely netlist data). The standard specifies a large number of procedures by which electronic CAD data can be interchanged. This interface is designed to enable fine grain integration of CAD tools and frameworks. Whilst it represents a major step forward in standardisation of interfaces between CAD tools and systems it has several disadvantages:

1. It has a very large number of separate procedures.
2. Any changes to the set of procedures must (in order to maintain backwards compatibility) always add many new procedures because new data types along with their associated attributes will inevitably be introduced.
3. The cost of adding a CFI DR PI to any given tool or environment is very high as each procedure must be recoded.
4. The representation captured by the CFI DR PI is specific to the domain in which it is designed to operate.

A simple way to improve the CFI DR PI would be to reduce the number of procedures. This could be achieved, for instance, by specifying the data type and attribute as parameters to the procedures instead of being encoded into the name of each procedure. This then results in an interface which is simpler to modify and is therefore more generic. Whilst this reduces the effects of points 1 and 2 above and may also reduce point 3, it has no effect on the wider applicability of the interface. Added functionality always requires further code to be added to the procedures which in turn requires that all applications which use the procedures must be reintegrated with the procedures.

3 Flexible Procedural Interface.

An alternative methodology for the implementation of procedural interfaces is to make the interface procedures generic rather than being specific to a particular set of application domains. Young [2] identifies three general functional requirements that have to be provided by access procedures:

1. Manipulation of data objects and establishment of relationships between them.
2. Manipulation of meta-data and establishment of relationships between meta level objects.
3. General housekeeping functions, e.g. logins.

He also identifies three main approaches to the way data objects are manipulated by access procedures:

1. Operation-orientation. A procedure is provided for each operation and the object type is passed as a parameter, e.g. get object(cell, ...), get object(port, ...).
2. Object-orientation. A procedure is provided for each object and the operation is passed as a parameter, e.g. cell op(get, ...), port op(get, ...).
3. A combined object and operation-oriented approach. A procedure is provided for each allowed combination of object and operation, e.g. get cell(...), get port(...).

The CFI DR PI

IREEN [6], SDAI [7] and the GPIC interface described here all adopt the first approach. The main advantage is that the number of operations (such as get, create, delete) is small in comparison with the number of object types (such as cell, library and port). This results in a small number of relatively static access procedures which are independent of conceptual model evolution. The style of the interface mirrors the structure of the underlying conceptual model.

Both object-oriented approaches result in interfaces with an unstable number of procedures which must change as the conceptual model changes. The CFI-DR [5] is an example of an object-oriented approach. Its main advantage is that structural constraints can be enforced by strong typing in the implementation language as opposed to runtime checks with the more generic approaches, e.g. checking if an attribute class belongs to a particular object type. There is therefore a trade-off between the increased complexity of the interface and the advantage (performance) of a strongly typed interface.

The GPIC interface has multiple layers of representation. The occurrence data (instances) for the domain is accessed via a set of data procedures which access the occurrences using a conceptual model. These describe the objects which occur in the application domain.

The conceptual model is defined as data of a meta-schema (based on a meta-conceptual model) which describes the objects and concepts from which the conceptual model is constructed. The conceptual model for applications using GPIC is represented in CMDL (CAD Meta-Data Language,
For data access using a database, CMDL is implemented via a database schema and provides a representation layer in between the database (physical storage) and the access procedures (GPIC). The lowest layer contains domain specific data such as occurrences of “ports” and “cells”. These in turn are examples of domain specific objects (types) which form the middle layer which is the conceptual model. The top layer contains meta-concepts such as “object”, “attribute” and “relationship” and is the meta-conceptual layer. There is a clear specialisation of concepts from the top layer through the middle layer to the bottom layer.

The conceptual model is interpreted by the data access procedures in order to find paths through the data representation for the storage and retrieval of occurrence data in a data repository [7]. Flexibility in the interface is achieved as access paths are not fixed. This means that if the conceptual model is changed then the interface, rather than the tools using the interface, can determine actual paths to the desired data.

In order that the conceptual model can be interpreted, the schema is not hard coded into the data access code. Rather, it is stored as meta-data which can be used to: Find which entities are represented by the schema, find relationships between entities in the schema, and determine access paths to the data.

### 3.1 Examples Using GPIC

Figure 3 shows three levels of representation. At the top is a simplified version of the conceptual model of CMDL in figure 2. This top model represents the domain independent meta concepts introduced in the previous section. The middle layer represents a schema for domain specific concepts. Figure 3 shows both a graphical and a textual view of the domain specific schema. The textual view of the middle schema is specified directly in CMDL. The bottom layer of figure 3 represents the occurrence structure on the left for the EDIF style data on the right. Each successive layer is implemented using the structures defined at the level above. Both the bottom and the middle layers utilise schemas which are stored as data by the environment. A traditional database implementation of a tool would have hard coded the schema of the middle layer instead of leaving it as meta-data.

Figure 4 outlines two examples using the GPIC procedures to access and manipulate the models shown in figure 3. The top example shows: A search of the schema for an object of type “view” called “VO”, a search of the contents of the identified view occurrence for any one “interface” entity, and a search of the contents of the identified interface occurrence for any “port” occurrence called “PO”.

The bottom example of Figure 4 shows the stages as an application uses the GPIC procedures to interpret the middle schema of figure 3 to create an occurrence structure similar to the bottom layer of figure 3.

### 3.2 Prototype Flexible Procedural Interface Implementation

GPIC [2] has been re-implemented for use within the JESSI-COMMON-FRAME environment [8]. Currently GPIC contains 55 procedures of which 26 manipulate occurrence data, 18 manipulate schema data, 9 manipulate meta-schema data and the rest are used for system management purposes.

A printed circuit board (PCB) routing tool [9] has been used to test the prototype GPIC interface. The router was designed to read in netlist (connectivity) data from a file and to produce its results in a similar format, again as a file. The router represents the data in its own internal data structures. In order to integrate this tool (via the GPIC access routines) to a fine grain database containing the netlist data, a mapping must be derived between the conceptual model of the database and the conceptual model representing the data structures of the routing tool.
In GPIC, the schema information about the global view is represented using CMDL. CMDL is also used to specify schemas for each local database. Evolution of local database schemas results in a series of schema versions. Mappings can be used to describe the transformations which occur when a schema evolves. These mappings describe removal, addition and structural changes to entities within a schema.

Pun [11] has shown that methods for mapping between quite different models of a reality can be developed provided...
a consistent abstract model of the reality can be derived. Versions of a schema each have a model of the reality which is particular to that schema. A bijective function maps between each schema's reality and the abstract reality. Because of the multiplicative nature of functions, it follows that if two versions of a schema have a bijective function mapping to the abstract reality then a single bijective function can be derived to map between the two versions of the schema.

Pun has constructed a prototype system to aid in deriving mappings between declarative knowledge (e.g. technology specific design rules) and the data structures of applications written in either “C” or “PASCAL”. This mapping engine plus the data dictionary/thesaurus concept introduced above provide the mechanisms for aiding tool/data integrators in deriving and then maintaining mappings between different data models.

Using mappings between schemas GPIC can be used to insulate tools against changes in the framework’s database schemas. Similarly, mappings can be used to insulate the framework against changes in the schemas used by tools. In addition, using mappings means that the external schemas of tools and of the framework’s database may themselves be different from their internal schemas. The GPIC interface is thus responsible for carrying out intra- as well as inter-schema transformations when accessing data.

5 Conclusions

A prototype GPIC interface has been built on top of the OMS database, part of the JESSI-COMMON-FRAME. This interface has been demonstrated as the data access component for a routing tool. The demonstration has shown that the interface can insulate the tool from simple changes in the database schema.

Currently the interface is being extended in order to implement the support for schema evolution using the mapping metaphor described in section 4 above.

The biggest problem for a fine grain interface is efficiency. The current implementation works by dynamically interpreting the conceptual models. Future work will involve: Compilation of data access routes, further implementation of the mechanisms for schema evolution, investigation of caching and clustering techniques in order to improve the efficiency, and benchmark testing with respect to other approaches.

References