

Dynamically Defined Flows: An Improved Method for Task Management in CAD Frameworks

Peter R. Sutton and Stephen W. Director

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

This paper introduces the concept of “dynamically defined flows” for design methodology management. Such flows are built up, on demand, by designers and can be used to provide a semantically rich means for browsing the design history database as well as to provide support for multiple design approaches, such as goal-based, tool-based, data-based and plan-based design.

1 Introduction

As CAD Frameworks evolve from managing only tools and data to managing the *design process itself*, the concept of a *design flow* has become an increasingly popular foundation on which to build such a framework. In general, a *design flow* describes a sequence of operations required to achieve design goals. Most approaches to design flow management, however, have a restricted view of a flow. Flows are often static, specifying a fixed sequence of design activities that the designer is often required to follow step by step (described in [1] as a “flow straight-jacket”). Flows are also usually hardwired to specific tools, and hence require modification whenever tool changes are made or new tools are added to the system. The disadvantage of this approach is that it limits the designer’s freedom in accomplishing his or her design goals.

In this paper, we introduce the concept of a *dynamically defined flow* which is constructed from a schema representing the dependencies between tools and data. Dynamically defined flows maintain the advantages of flow-based approaches while giving the designer maximal control over the design process.

2 Dynamically Defined Flows

Dynamically defined flows are tool-independent flows that are built up on demand, by designers. The designer is not restricted to operating within the bounds of hard-wired flows that are specified in advance, but may build up any flow - subject to some rules. These rules are contained in a *task schema* which specifies the dependencies between tool and data types, for example, to produce a “circuit performance” you need to run a “circuit simulator” on a “netlist”. The task schema is described in [3].

2.1 Representation

Traditionally, flows have been represented by a bipartite flow diagram such as that shown in Fig. 1.(a). We have chosen an alternative representation called a *task graph*, that emphasizes the similar manner in which tools and data may be handled in a dynamically-defined flow. An example of a task graph is shown in Fig. 1(b).

A task graph is a directed acyclic graph, with each node in the graph corresponding to an entity in the task schema, and each edge corresponding to a dependency. A dynamically defined flow (*represented* by a task graph) is a temporary structure that can be built up by the designer as desired (subject to the rules in the task schema). *Expand operations* can be used to incorporate further primitive tasks into a flow. For example, two possible *expansions* of the flow in Fig. 1(b) are shown in Fig. 2. Note that the circuit in Fig. 2(b) was *specialized* to an **Extracted Circuit** before expansion. (*Specialization* is the selection of an entity *subtype* so that an expand operation can be performed.) When the flow is built up as the designer desires, the entities can be *instantiated* (an instance selected for each leaf node) and the task executed. Alternatively, the designer could run each task as a primitive task, using the result of that to build a new flow and so on.

More complex flow structures are also possible. Fig. 3 shows a flow involving the reuse of an entity in several sub-tasks and the production of multiple outputs, including multiple outputs from the same sub-task. This flow could be constructed by starting at any one of the entities present and performing expand operations until the flow was built up. (Previous use of the task schema to construct flows¹ was restrictive in how entities were to be used. Construction was allowed from the goal entity only, entities could not be reused,

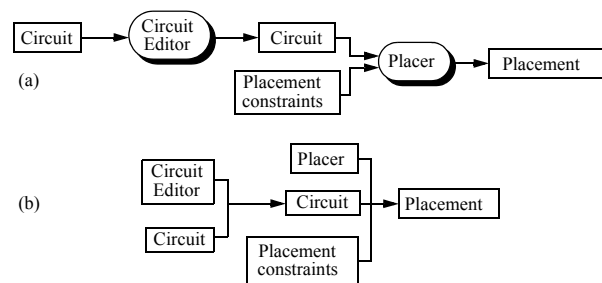


Fig. 1. Two possible representations of a dynamically defined flow: (a) a traditional bipartite flow diagram, and (b) a *task graph*.

and tasks could not produce multiple outputs [3].)

2.2 Design Management Support

Dynamically defined flows and the task schema provide support for a variety of framework services. The task schema provides support for **design data management** by providing a classification of tools and data. It also forms the data schema of a design history (or design meta-data) database. Dynamically defined flows can also be used as a query template (or form) with which queries into the design history database can be made. Queries into the design history database can be used to support **design consistency maintenance**, i.e. retracing of a flow to update derived design data. As tool and data dependencies are known, **flow automation**, that is, the automatic sequencing of tasks is possible. It is also possible to support the parallel execution of tasks. **Design methodology management** is supported by providing at least some rules about how flows can be built up. **Methodology maintenance** is simplified by avoiding the requirement to maintain a set of flows, only the tool-independent task schema need be maintained.

As can be seen, dynamically defined flows enable the provision of many framework services, and as will be seen in the following section, they also allow greater freedom in how a design is approached.

2.3 Multiple Design Approaches

Dynamically defined flows allow designers to be very flexible in their approach to solving a design problem. Any one of four different approaches may be selected. In the *goal-based* approach, designers identify a task by first selecting the goal entity of the task from the task schema. The *tool-based* approach allows users to initially select either the tool-entity or the tool-instance that they wish to work with. In the *data-based* approach users initially select an existing piece of data that they wish to work with. The *plan-* or *flow-based* approach allows designers to choose from a set or library of flows that they (or another user) have built up previously. This approach would normally be used when repeating a common design activity.

Allowing the designer to approach the problem in any way they choose provides the greatest possible designer flexibility.

3 Implementation

The concepts described above have been implemented in the modified version of the Hercules Task Management System [3], which is part of the Odyssey CAD Framework [2].

3.1 Execution of a Simple Task

Hercules allows users to be very flexible in their approach to a design task, being able to approach the task from multiple viewpoints². In [1], this is done using different user interfaces for each approach. Hercules, however, now uses the same user interface for each approach.

A visualization of a task graph forms the basis of the Hercules user interface, as shown in Fig. 4. Suppose, for example, that the designer wishes to obtain a circuit performance from an existing netlist. To start the task, the designer may select a predefined flow from the *flow-catalog*, a design entity type from the *entity-catalog*, a tool from the *tool-catalog*, or a piece of data from the *data-catalog*. In this example, suppose that the user elects to start by choosing an entity from the entity-catalog. The user can select either the goal type of the task (**Circuit Performance**), the entity corresponding to the tool to be used (**Circuit Simulator**), or the entity corresponding to the netlist to be simulated (**Netlist**). An icon representing this entity then appears on the screen. Using a pop-up menu associated with the icon, the user can then build up the flow simply and easily using *expand operations* (see Fig. 4(a)) described earlier (§2.1). Once the flow is built up, the user can select and fill in instances for the leaf nodes of the flow by using the browser associated with each leaf node entity (see Fig. 4(b)). Once instances have been selected for the leaf nodes, the non-leaf nodes become executable and may then be run.

Flows of any complexity can be created. Flows can be

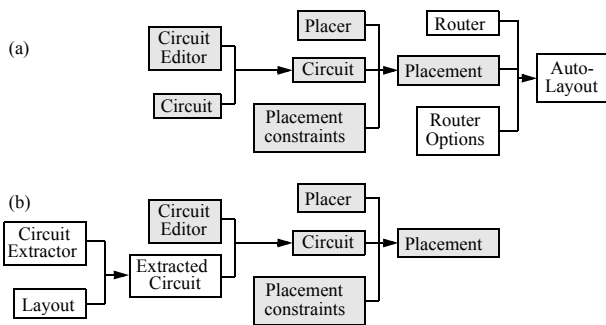


Fig. 2. Two possible expansions of the flow shown in Fig. 1.(b).

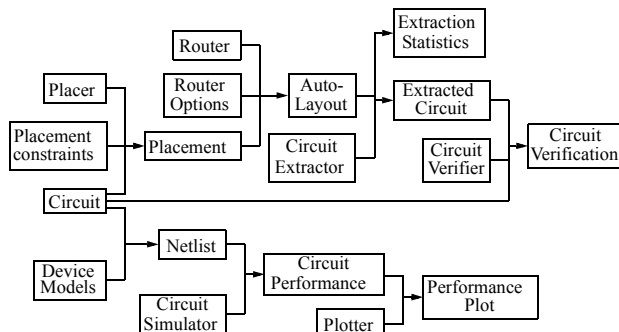


Fig. 3. A complex flow structure

1. The flows were called *task trees* and were true trees, looking like those in Fig. 2.. We have abandoned the name “task tree” as the flow structure is no longer restricted to being a tree.

2. The original version of the Hercules Task Manager supported only the goal based approach.

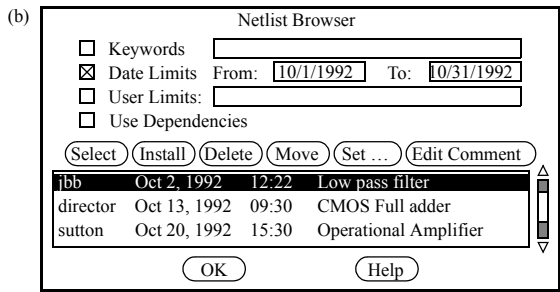
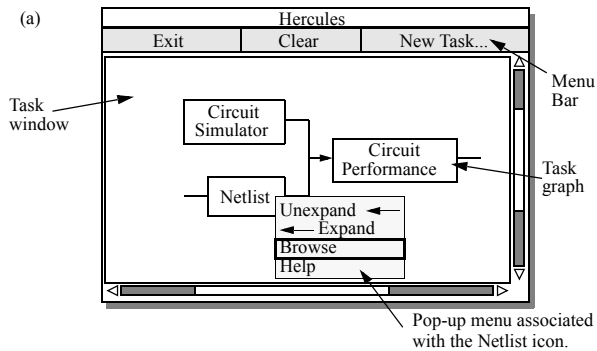


Fig. 4. Two parts of the Hercules User Interface: (a) the task window, (b) an entity instance browser.

expanded in either direction and can be of any depth. A sub-flow may be run at any stage – as long as its dependencies are satisfied – independently of the remainder of the flow. Instance browsing (and selecting) also need not wait until the complete flow has been generated; it may be done at any time that an icon for the given entity exists on the screen.

As can be seen from the Netlist browser in Fig. 4(b), meta-data such as user-id and creation time-stamp are recorded. The user is also able to annotate entity instances – providing both a name and a more detailed textual description if desired. This annotation can be used to help document the design steps taken. An instance’s most important meta-data is its design history which records the entity instances used to create that instance. This design history can be queried, using the flow itself as a query template. It also allows previously executed tasks to be recalled, possibly modified, and executed. Details of this are given in the following section.

3.2 Using the Design History

The task graph can be used to formulate and return the result of queries into the design history database. Simple que-

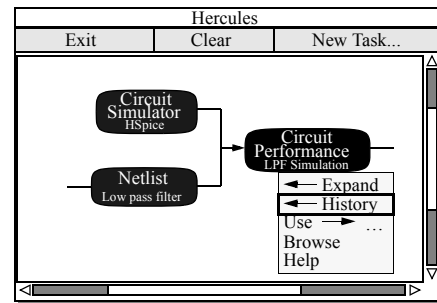


Fig. 5. Browsing the design history. Note that the Circuit Simulator and Netlist entities do not appear until after ‘History’ is chosen. The inverse display means that the icon represents a unique instance. The name of this instance appears within the icon.

ries about an entity instance can be formed using the browser associated with each entity (see Fig. 4(b)). More complex queries involving *backward-chaining* and *forward-chaining* through the design history can also be formed.

Backward-chaining queries into the design history database are used to find an instance’s derivation history. For example, after selecting a unique instance of an entity, a designer may select “History” from the icon’s pop-up menu to reveal the instances used to create it (see Fig. 5). The other meta-data (user, time-stamp, comment) can be examined by browsing on the revealed instances.

Forward-chaining queries are used to find the entity instances which depend upon a given instance, or instances, e.g., finding all of the circuit performances derived from a given netlist. This is achieved by constructing a task graph, specifying the known instances, and then browsing on the parent entity using the “Use dependencies” option in the browser (see Fig. 4(b)).

References

- [1] M. Rumsey and C. Farquhar. “Unifying Tool, Data and Process Flow Management.” In *Proceedings of First European Design Automation Conference, GI/ACM/IEEE/IFIP*, 1992, pages 500-505.
- [2] J.B. Brockman, T.F. Cobourn, M.F. Jacome, and S.W. Director. “The Odyssey CAD Framework.” In *IEEE DATC Newsletter on Design Automation*, Spring 1992.
- [3] J.B. Brockman and S W. Director. “The Hercules CAD Task Management System.” In *Proceedings of the International Conference on Computer-Aided Design*, IEEE, 1991, pages 254-257.