

Design Management Using Dynamically Defined Flows[†]

Peter R. Sutton*, Jay B. Brockman** and Stephen W. Director*

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

**Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556

Abstract

Many CAD frameworks now use the notion of a design flow to help provide methodology management services. Most flow-based approaches are limited, however, in that they involve a fixed sequence of operations specified in advance, restrict designers to using only those flows, and “hardwire” specific tools to flows. To overcome this, we introduce the concept of “dynamically defined flows” as tool-independent flows that are built up, on demand, by designers. Dynamically defined flows 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. In our view, it is disadvantageous to specify the exact tool sequences that should be followed by the designer in a creative design process. The designer should be given the freedom to move about the flow as necessary in order to accomplish 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. We show how this approach gives the designer greater freedom in choosing how to go about the design, even within the bounds of a fixed methodology.

A second point that we wish to emphasize with this paper,

[†] This work is supported in part by the Semiconductor Research Corporation and IBM.

is the inherent power of a flow-based approach to design management for organizing design data. This power arises from the fact that all design objects are created through the execution of flows and that each design object may be uniquely identified according to the sequence of tool/data transformations used in creating that object. A consequence of this is that *if flows are properly defined, queries into the derivation history of design objects obviate the need for additional version management schemes*. Further, we will show that by associating a small amount of meta-data with each design object, indicating the immediate tool and data used in creating that object, the complete derivation history of a design may be stored.

The paper is organized as follows. Section 2 presents an overview of previous work in design flow management. Dynamically defined flows are then defined and described in Section 3. Section 4 outlines how dynamically defined flows have been implemented in the Hercules Task Manager, part of the Odyssey CAD Framework [2].

2 Previous Work

Over the past few years, numerous papers have appeared that address design management through flows. The definition and representation of flows as used in the JESSI Common Framework project [3] is fairly typical of these. JESSI uses the term *flow* to mean a predefined sequence of *activities*, where an activity represents a particular feature of a tool (taking specific input data and producing specific output data). Flows define the temporal interrelation between activities and can be specified with either linear or branched connections. Design work is then carried out by associating flows with *cells* in a design. Similar constructs may be found in the *flowmaps* used by Philips [4] and the Technical University of Delft [5], *task templates* by U.C. Berkeley [6], our older *task trees* [7] and the more generic *task graphs* introduced in this work. Casotto [8] has defined the term *trace* to refer to a historical record of a sequence of tool invocations and data transformations that have been performed during design. A trace may be considered one example of design meta-data – information about design data – that was described in [5].

One of the main issues in flow-based design management is deciding which data to assign to flows. The Philips “Data Flow Based Architecture for CAD Frameworks” [4] was the first work to suggest that a design derivation history based on flows may be sufficient for organizing design data without the need for additional constructs. By storing flow definitions and design

meta-data in a common database, van den Hamer and Treffers were able to easily process queries such as “find the simulations that were performed for this netlist.” Chiueh and Katz [6] use *activity threads* in a similar manner to assist designers in identifying needed data. For example, if a designer implemented a logic circuit using standard cells and then wished to re-implement the same circuit using a PLA, he or she could reposition a “cursor” to the appropriate point in the standard cell activity trace and create a new activity branch using a “create PLA” task.

A second issue in flow-based design management is deciding which activity to perform next. The most restrictive option is to confine designers to operate within predefined flows – which may lead to the “flow straight-jacket” described in [1]. While the approaches described in [4], [5], and [6] utilize predefined flows, in these systems users ultimately have control over deciding which flows are to be executed when, and thus may dynamically determine the sequencing of design activities. Casotto [8] avoids the problem of flow restriction entirely by merely capturing a trace of designer activity and allowing existing traces to be used as prototypes for new activities. The problem with this approach is that it provides no means for *enforcing* a particular design methodology (though one may be *defined*), nor does it provide a means for organizing and indexing traces in a more generalized fashion than with regard to specific design data files. As Rumsey and Farquhar [1] suggest, designers should have the option of accessing design activities from either a tool-, data- or flow-oriented viewpoint, with only the latter restricting the user to execute tools in a set sequence. In this case though, choosing the tool- or data-oriented views allows the designer to escape from any fixed methodology.

We now propose the notion of *dynamically defined flows* to maintain the advantages of a flow-based approach while giving the designer maximal control over the design process.

3 Dynamically Defined Flows

Before being able to build up dynamically defined flows, some knowledge of the kind of tasks that can be performed and how they can be strung together is required. A *task schema* may be used to represent this information. In this section, we present an overview of the concept of a task schema and then introduce dynamically defined flows. We then show how they can be used to support various framework services as well as multiple design approaches.

3.1 Task Schema Overview

A task schema (originally introduced in [7], and extended here) is a graph that specifies the dependencies between design entities (both tools and data). The dependency relationships described in a task schema serve two purposes. First, they state the construction rules by which tasks (tool independent design functions) can be built. Second, they specify the data schema for a database that stores the design derivation history. Tasks may be *primitive* – performing a single function, such as “extract a netlist from a layout”, or *complex* – e.g., “extract a

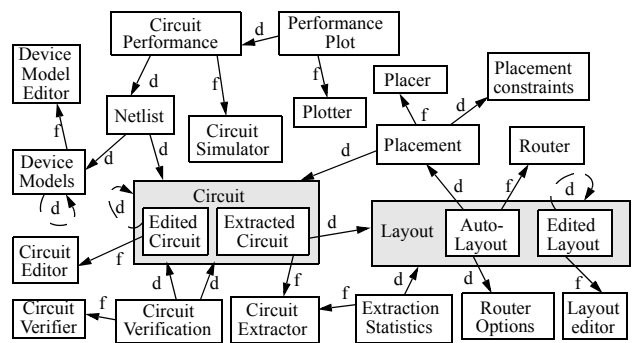


Fig. 1. An example task schema.

netlist from a layout and simulate its performance.”

A task schema can be expressed graphically as shown in Fig. 1.. Design entities (represented by rectangles) are connected to other entities by directed arcs labelled with **f** or **d**. An **f** indicates a functional dependency, e.g., a **Circuit Performance** is functionally dependent on a **Circuit Simulator**. A **d** indicates a data dependency. An entity can have at most one functional dependency and an unlimited number of data dependencies.

In cases where there is more than one way of generating a design entity, subtyping can be used to separate the different construction methods. For example, in Fig. 1., **Auto-Layout** and **Edited Layout** are two subtypes of entity type **Layout** that have different construction methods. This figure also shows that loops in the task schema are possible: an **Edited Layout** depends on a **Layout**. Loops like this are broken by considering the data dependency as optional (shown by a dashed arc).

Because tools and data are both treated as entities, it is possible to support tools that are created during the design process. An example of such a tool is the switch-level simulator COSMOS [10] which is compiled for a given netlist and can then be executed on different stimuli. Fig. 2. shows how a task schema can represent such a process.

Design decomposition can also be represented by extending the original definition of a task schema to allow an entity (called a *composed entity*) to have only data dependencies and no functional dependency. (For example, in Fig. 1., a **Netlist** is shown to depend only on the **Device Models** and **Circuit** entities.) From our earlier experience with the task schema, we have found it useful to allow designers to group together two or more entities that are commonly used together, such as circuit descriptions and device models. This concept is also useful for grouping entities which can be simultaneously produced by a single task.

Although composed entities have no explicit functional

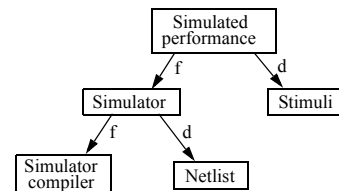


Fig. 2. Task schema subgraph for a tool created during a design.

dependency, they do have implicit *composition* and *decomposition* functions associated with them. Composition functions can be used, for example, to check for consistency between entities (e.g., can these device models be used with this circuit?). Decomposition functions can be used, for example, to split an entity instance's data into component parts¹. More complicated notions of design decomposition (such as a hierarchy of cells within a design) can be handled at a higher level of abstraction. In the Odyssey CAD Framework [2], this is the *Design Process Level* – implemented in the Minerva Design Process Manager [11].

3.2 Definition and Representation

A *dynamically defined flow* is a sequence of primitive tasks (forming a complex task) which is generated, on demand, by the user of the design system. Traditionally, flows have been represented by a bipartite flow diagram such as that shown in Fig. 3.(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 flow². An example of a task graph is shown in Fig. 3.(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. 3.(b) are shown in Fig. 4.. Note that the circuit in Fig. 4.(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,

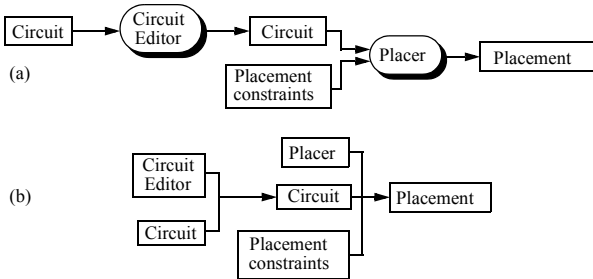


Fig. 3. Two possible representations of a dynamically defined flow (based on the task schema in Fig. 1.), (a) a traditional bipartite flow diagram, and (b) a task graph.

- In practice, we have found that this is rarely necessary. The design data is often stored separately anyway, with the composite entity storing pointers to the component parts.
- Our representation of a flow is analogous to the Lisp representation of a function, whereas a traditional flowmap is analogous to the C or Pascal representation. For example, we may write Fig. 3.(a) as:
 $placement \leftarrow placer(circuit_editor(circuit), placement_constraints)$
 whereas Fig. 3.(b) may be written as:
 $placement \leftarrow (placer, (circuit_editor, circuit), placement_constraints)$.
 We are treating the tool as just another parameter.

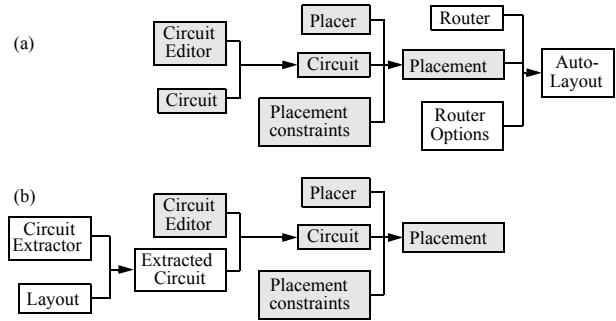


Fig. 4. Two possible expansions of the flow shown in Fig. 3.(b).

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. 5. shows a flow, based on the task schema of Fig. 1., involving the reuse of an entity in several subtasks 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 flows³ was restrictive in how entities were to be used. Construction was allowed from the goal entity only, entities could not be reused, and tasks could not produce multiple outputs [7].)

3.3 Design Management Support

Dynamically defined flows and the task schema provide support for various framework services. These services are outlined below.

Support for **design methodology management** is provided in the task schema itself. The schema describes the tasks that can be executed and how design entities can be used to form these tasks. While we cannot restrict a designer to a fixed sequence of tasks, we believe that this is advantageous: the designer should be able to perform any *allowable* task in any order.

Dynamically defined flows easily allow for automatic task

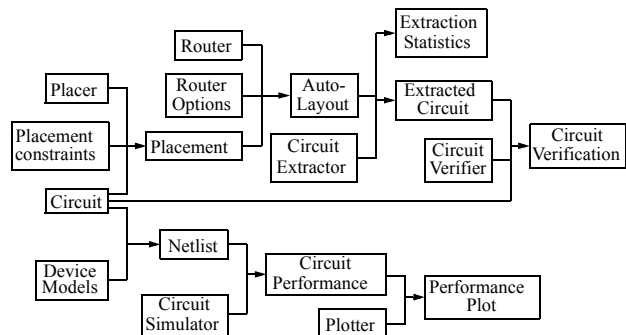


Fig. 5. A complex flow structure, based on the task schema of Fig. 1.

- The flows were called *task trees* and were true trees, looking like those in Fig. 4.. We have abandoned the name “task tree” as the flow structure is no longer restricted to being a tree.

sequencing (**flow automation**) because tool and data dependencies are specified in the task schema. It is also possible to support parallel task execution, wherein disjoint branches in the flow can be executed in parallel, possibly on different machines (see Fig. 6.).

As stated earlier, the task schema aids **design data management** by forming the data schema for a design meta-data (design history) database. Further details of this are given in [12]. As will be illustrated in section 4.2, this flow structure can also be used as the *form* or *template* for queries into the database.

Tools which can perform multiple functions are often handled by defining separate *activities* for each tool behavior. Using our approach, if the multiple behaviors correspond to differing entity types (e.g., one tool that can function both as a layout editor and a circuit extractor), the tool (or its encapsulation) can be instantiated for each of those entity types. For behaviors corresponding to the same entity type (e.g., automatic layout using different algorithms – specified by arguments to the tool), multiple encapsulations can be used to specify the differing arguments. Another approach is to define the options or arguments themselves as an entity type. An example is the entity **Router Options** in Fig. 1.. It is also possible to share encapsulation code among several tools. For example, we have encapsulated three statistical circuit optimization tools that take exactly the same input arguments and produce the same type of output using this technique.

Finally, tools themselves may serve as data input to other tools. For example, an optimization procedure may have a circuit simulator passed to it as an argument.

Design consistency maintenance (i.e., automatic retracing of a flow to update derived design data), is readily supported through the storage of the design history. Queries into the design history can quickly determine whether such retracing need occur. For example, a query such as “find the netlist that was extracted from this layout” could determine whether such an extraction had yet been performed, or whether the extracted netlist was out-of-date with respect to the layout.

Another aspect of design consistency maintenance is ensuring that different *views* of a design are in correspondence. Designers often think of a design in terms of different views such as a logic view, a transistor level view, or a physical view, as shown in Fig. 7. View management is typically seen as the responsibility of a data management system in a framework, separate from methodology management. If views of a design are associated with entities in a task schema, however, flows

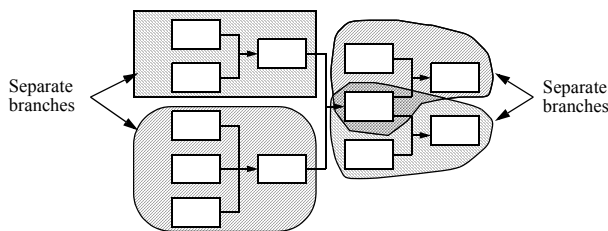


Fig. 6. Separate branches can be executed in parallel.

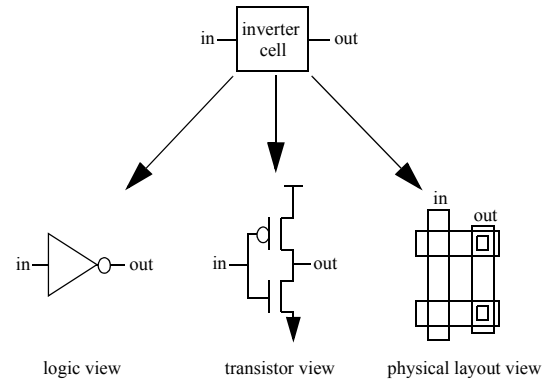


Fig. 7. Three views of an inverter cell.

can be used to represent the transformations between views. For example, given the task schema in Fig. 1, the flow in Fig. 8 (a) will synthesize the physical view of a circuit from the transistor view, while the flow in Fig. 8 (b) will verify that the physical view is consistent with the transistor view.

As can be seen, dynamically defined flows enable the provision of many framework services. Not only can they be used to support these services, they also make methodology maintenance easier by avoiding the requirement for the maintenance of a set of flows (only the task schema need be maintained), and by simplifying the incorporation of new tools. They allow designers greater freedom by not restricting them to work on a static flow, and, as will be seen in the following section, they also allow greater freedom in how a design is approached.

3.4 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

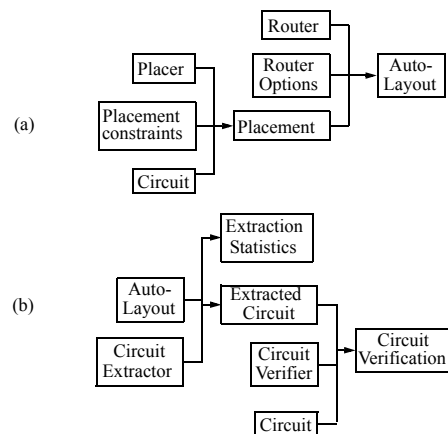


Fig. 8. Flows for (a) the synthesis of physical view of circuit, and (b) verification that physical view corresponds to transistor view.

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.

4 Implementation

The concepts described above have been implemented in the modified version of the Hercules Task Management System [7], part of the Odyssey CAD Framework [2].

4.1 Execution of a Simple Task

Importantly, as in [1], and as described in 3.4 above, Hercules allows users to be very flexible in their approach to a design task, being able to approach the task from multiple viewpoints⁴. In [1], this is done using different user interfaces for each approach. Hercules, however, 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. 9.. 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. If the task schema is the one shown in Fig. 1., then the entity-catalog will list all of the entities in Fig. 1.. 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. 9.(a)) described earlier (§3.2). (Expand operations would not be necessary if the user had chosen a predefined flow.) 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. 9.(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 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. Also, the selection made in the database browser need not be unique: it is possible to select more than one instance, or a set of instances – causing the task to be run for each data instance specified. (The relevant

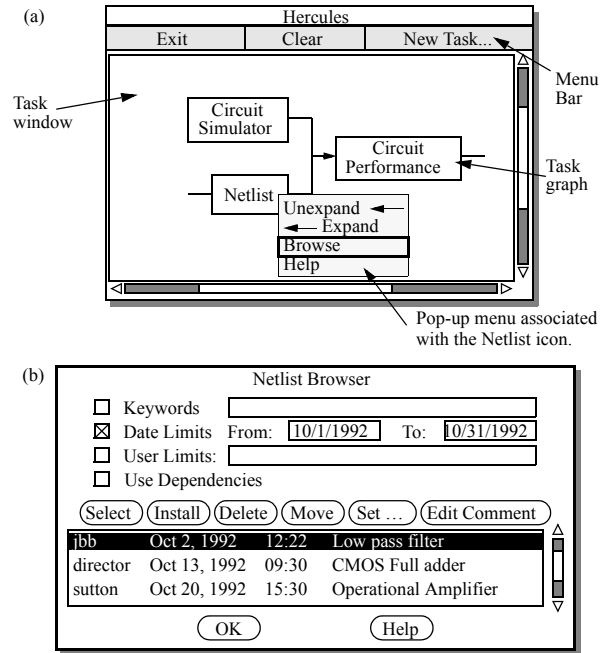


Fig. 9. Two parts of the Hercules User Interface: (a) the task window, (b) an entity instance browser.

encapsulation may cause the tool to be run for each instance selected or it may pass all of the data to a single call of the tool.)

As can be seen from the Netlist browser in Fig. 9.(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.

4.2 Using the Design History

The task graph can be used to formulate and return the result of queries into the design history database. Simple queries about an entity instance can be formed using the browser associated with each entity (see Fig. 9.(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. 10.). 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

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

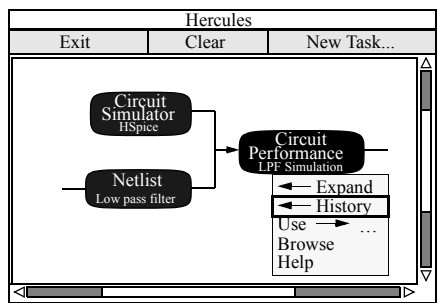


Fig. 10. Browsing the design history. Note that the Circuit Simulator and Netlist entities do not appear until after ‘History’ is chosen. The inverse display means that the icon represents a unique instance. The name of this instance appears within the icon.

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. 9.(b)).

As every design object created has its design history instance stored in the design history database⁵, it is possible to use this history to keep track of different *versions* of design objects. Versioning is closely associated with editing tasks which, in a task schema, are characterized by having a data dependency whose source and target are of the same entity type. (For example, in Fig. 1. an **Edited Circuit** depends (optionally) upon a **Circuit**.) Versioning of a design is traditionally represented by a version tree (see Fig. 11.(a)). Our representation – a *flow trace* – is a semantically richer superset of a version tree, not only showing the relationship between the data, but also showing the tools that were used in creating that data (see Fig. 11.(b)). A flow trace has the same form as a task graph and can be built up using the forward- and backward-chaining approaches described above.

5 Conclusions

We have introduced the concept of *dynamically defined flows* – tool independent flows (represented by a task graph) that are built up, on demand, by a designer subject to the rules contained in a *task schema*. Dynamically defined flows provide support for various framework services (including design methodology management and automatic task execution) in a tool independent manner. Design data management support is also provided: the task graph itself can be used to formulate and return the results of queries into the design history database. Dynamically defined flows also provide support for multiple design approaches – designers can choose a goal-based, tool-based, data-based or plan-based approach to solving their design problem.

5. Note that, although each instance of an entity (including different versions of the same design) has its own associated meta-data, it may share the actual (physical) data with other instances. For example, several design history instances could point to the same Unix RCS (Revision Control Software) or SCCS (Source Code Control Software) file, but have different version numbers stored in the meta-data.

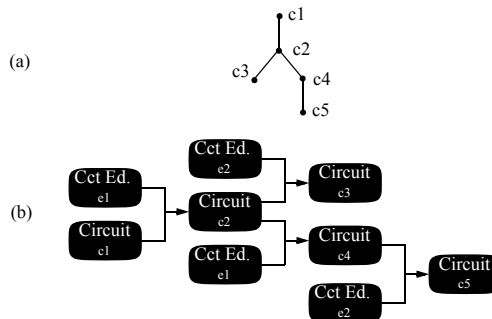


Fig. 11. Representations of version trees: (a) a traditional version tree, and (b) a *flow trace* as represented in Hercules. A flow trace shows the tool used to create each version.

The concepts presented here have been implemented in the latest version of the Hercules CAD Task Manager, part of the Odyssey CAD Framework.

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] D.C. Liebisich and A. Jain. “JESSI Common Framework Design Management – The Means to Configuration and Execution of the Design Process.” In *Proceedings of First European Design Automation Conference*, GI/ACM/IEEE/IFIP, 1992, pages 552-557.
- [4] P. van den Hamer and M.A. Treffers. “A Data Flow Based Architecture for CAD Frameworks.” In *Proceedings of the International Conference on Computer-Aided Design*, IEEE, 1990, pages 482-485.
- [5] K.O. ten Bosch, P. Bingley, and P. van der Wolf. “Design Flow Management in the NELSIS CAD Framework.” In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, ACM, 1991, pages 711-716.
- [6] T. Chiueh and R. Katz. “A History Model for Managing The VLSI Design Process.” In *Proceedings of the International Conference on Computer-Aided Design*, IEEE, 1990, pages 358-361.
- [7] 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.
- [8] A. Casotto, A.R. Newton, and A. Sangiovanni-Vincentelli. “Design Management based on Design Traces.” In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, ACM, 1990, pages 136-141.
- [9] P. van der Wolf, G.W. Sloof, P. Bingley, and P. Dewilde. “Meta Data Management in the NELSIS CAD Framework.” In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, ACM, 1990, pages 142-145.
- [10] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. “COSMOS: A Compiled Simulator for MOS Circuits.” In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, ACM, 1987, pages 9-16.
- [11] M.F. Jacome and S.W. Director. “Design Process Management for CAD Frameworks.” In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, 1992, pages 500-505.
- [12] J.B. Brockman and S.W. Director, “A Schema-Based Approach to CAD Task Management”, In *Proceedings of the Third IFIP WG 10.2 Workshop on Electronic Design Automation Frameworks*, Edited by T. Rhyne and F.J. Rammig, Elsevier Science Publishers, 1992.