Improved Tool and Data Selection in Task Management

John W. Hagerman and Stephen W. Director
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

Task management involves task creation and execution. These are facilitated using a *task schema* as exemplified in the Hercules Task Manager. Experience with Hercules has shown the task schema to be very useful for task creation, but less than ideal for task *resolution*, i.e., the selection of tool and data resources to be used in execution. Tool/data interactions often lead to resource selection constraints that cannot be captured using dependency relationships in the schema. We have addressed this by adding *conditions* to the task schema which use task-level meta-data to constrain resource selection. With examples we show that conditions are useful for handling a wide variety of real constraints.

1. Introduction

Task management is an important feature of design frameworks. A task is a set of steps leading from one set of data to another. For example, a task in the EDA domain might be to synthesize a circuit from a specification and then to use simulation to estimate circuit performance. Task management involves *task creation* (building a task tree), *task resolution* (selecting tools and data), and *task execution* (running the tools).

One approach to task management is to use a *task schema* which captures the relationships among tools and data. This approach is taken by the Hercules Task Manager [1]. Through Hercules we have found the task schema approach to be very useful in a wide variety of design situations. However, we have also found that the task schema does not handle the subtleties of task resolution very well. Specifically, the interactions among tools and data that must be considered during resolution often cannot be captured as simple dependency relationships. This is subtle because although the interactions can be captured as relationships, the relationships are not of the kind used in the original task schema formulation. We have addressed this by enhancing the task schema with the addition of *conditions*, which are Boolean expressions that use information attached to tools and data to constrain selection. In this paper we describe how we added conditions to the Odyssey framework [2] and show how conditions can be used to capture the constraints needed for tool ordering enforcement, multi-function tool management, tool/data matching, and data coherence.

We begin with a review of the Odyssey framework, Hercules, and task schemata. Then we describe the addition of conditions to Hercules, including the changes made to the task-level database, the schema, the resource manager, and the Hercules User Interface. Then we present results, and close with conclusions.

2. Background

Odyssey is a design framework test-bed developed at Carnegie Mellon University. In Odyssey, framework responsibilities are handled in a layered manner, as shown in Figure 1. Hercules is the task management layer. Tool and data details such as file locations and formats are hidden from Hercules by the Cyclops resource manager [3]. Tool and data resources are *encapsulated*, and Cyclops presents a uniform view of resources to Hercules. Thus, Cyclops mediates between encapsulations and Hercules.

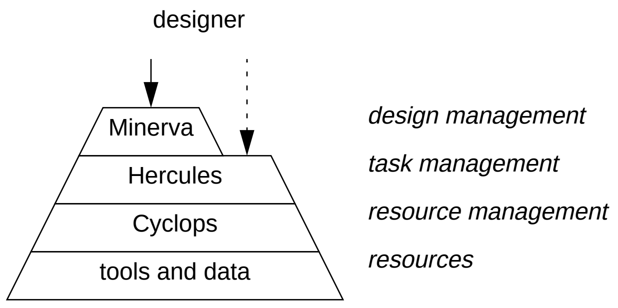

Figure 1: The Layers of the Odyssey Framework

A designer can interact directly with Hercules to manage tasks. Hercules also provides task management to the Minerva design process management layer currently under development [4]. The designer gives problem specifications and goals to Minerva, and Minerva helps the designer break the problem into sub-problems and map them onto tasks. This paper focuses on Hercules.

2.1 The Task Schema

In order to manage tasks, a framework must be aware of the relationships among tools and data. At the task level the unnecessary distinction between tools and data is removed by representing all resources as *entities*. (This is important for the accurate modeling of entities such as scripts that act as tools at some times and data at other times.) A hierarchy of entity *classes* (types) is defined. Each actual resource is an *instance* of some class. A *task schema* is an entity-relationship graph: vertices represent classes, and directed edges represent dependencies. An edge X→Y means that X depends on Y, that is, an instance of Y is needed to make an instance of X. The Hercules database holds static and dynamic information. The class hierarchy and the schema are static, while information about instances is dynamic. Information about an instance includes the mapping to the Cyclops encapsulation, and other *meta-data* such as creation time and creator username.

Consider the schema shown in Figure 3. Entity classes are drawn as boxes. Classes are hierarchical: a generic class is specialized to capture the methods by which instances are generated. In the figure, the generic class **specification** has two specializations: **analyzed-spec** and **transformed-spec**. The edges indicate, for example, that an **evaluation** has a *functional* dependence (“f”) on an **evaluator** and a *data* dependence (“d”) on a **specification**. Circular data dependencies may exist from a specialization to its generic class; two are shown in the figure.

A task is easily created from a schema. A task *tree* has vertices for entity classes and edges to indicate information flow. The task tree in Figure 5 could have been created from the schema of Figure 3. The root is at the bottom, and it should be clear how tree edges are obtained from schema dependencies. The goal of a task is to create an instance of the entity at the root of the tree; all the other vertices represent the tools and data to be used to achieve the goal. (Odyssey allows multiple roots; we will assume a single root and mention multiple roots only as necessary.)

2.2 Task Resolution

Tasks created from schemas are abstract, i.e., they are defined in terms of entity classes rather than specific instances. Therefore, before a task can be executed, specific tool and data instances to be used in the execution must be selected. This process is called task *resolution*. When instances have been selected for all leaves of a task tree the task is *resolved*. To execute a resolved task, the framework invokes the tools in an order that creates instances for the interior vertices as necessary to achieve the goal.

Task resolution is performed as follows. Figure 5 is the Hercules User Interface display of a task tree. At the top of each vertex box is the entity class name, and at the bottom is the selected instance name, if an instance has been selected. In the figure, an instance has been selected only for the **transformer** vertex. The user chooses a vertex to resolve and a browser appears which lists all the selectable instances for that vertex. The user selects an instance from the list, or selects *no* instance, and clicks an OK button. The user continues in this way, resolving vertices one at a time. When the task is resolved the user can request that it be run.

3. Selection Constraints

Experience with Hercules has shown that the task schema is very useful for creating tasks. We have also found, however, that tool/data interactions often imply resource selection constraints that cannot be captured using dependencies. For example, consider the analysis/transformation design flow shown in Figure 2. First, a specification is evaluated to determine whether it meets a set of goals. If the goals are met then design can proceed. Otherwise the specification is analyzed to determine what changes are necessary. The analysis tool annotates the specification to record the needed changes, and the transformation tool bases its operation on the annotations. Then the evaluation is repeated.

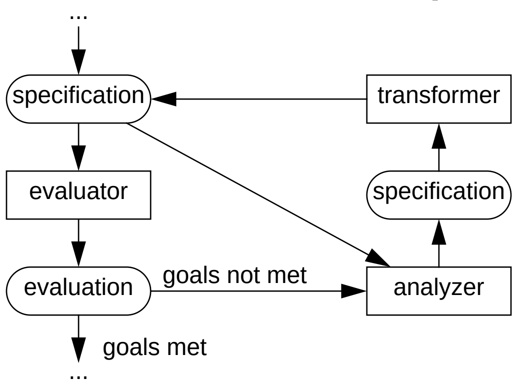

Figure 2: An Analysis/Transformation Design Flow

The data passed among the tools is all of the same type: a specification. Thus, the task schema illustrated in Figure 3 accurately captures the relationships among the tools and data.

This task schema indicates that a new specification is obtained by applying either an analysis or a transformation tool to an existing specification. However, an important relationship not captured in the schema is that analysis must be performed before transformation, since analysis adds information to the specification that is used in transformation. Thus we must track whether or not each **specification** instance has annotation information. We do this by attaching *attribute* meta-data to the instances, where an attribute is a name-value pair. Here we use an `annot` attribute to indicate whether a **specification** instance has been annotated. Then we add *conditions* to the task schema to test the attribute, as shown in Figure 4. The condition `!?annot@2` on the **analyzed-spec** entity means that **specification** instances at the other end of edge number 2 (“d2” in the figure) must *not* have the `annot` attribute to be selectable (`!?` is the “not-exist” operator). The condition `??annot@4` on **transformed-spec** means that **specification** instances at the other end of edge number 4 (“d4” in the figure) *must* have the attribute to be selectable. This shows the flexibility of using conditions to enforce orderings. The **transformed-spec** condition specifies only that the **specification** have an `annot` attribute. How and when that attribute is created is of no concern to the transformation tool, so this method permits *any* sequence of steps that leads to a properly annotated specification.

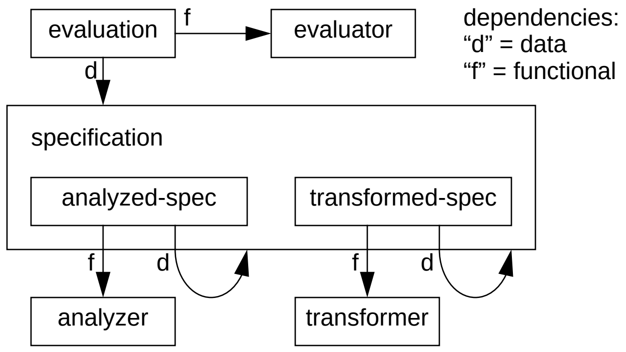

Figure 3: Analysis/Transformation Task Schema

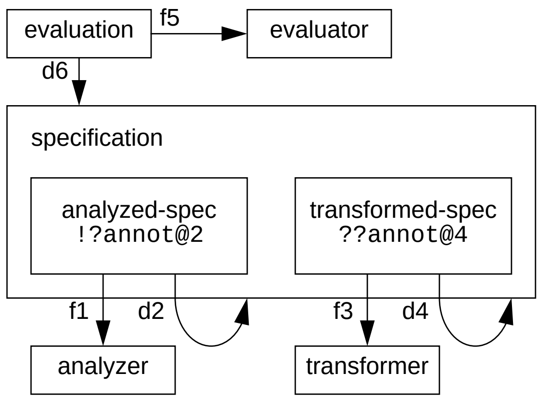

Figure 4: Previous Schema with Conditions Added

3.1 Condition Evaluation

In general, a condition is a Boolean combination of literals that test attributes attached to instances so as to constrain selection. Condition evaluation occurs as follows. The *selection state* of a task indicates the selection of instances at all vertices. It is with respect to this state that conditions are evaluated. For example, suppose the user is resolving the task of Figure 5, created from the schema of Figure 4, and suppose that the user has chosen to resolve the **specification** vertex. When resolving a tree vertex, it is the conditions at the *parent* of the vertex that must be satisfied. The parent of the tree vertex **specification** is **analyzed-spec**, which has the condition `!?annot@2`. To generate the list of selectable instances for the **specification** vertex, each appropriate instance is proposed in turn as being selected, the selection state is updated to reflect the proposal, and the parent’s condition is evaluated with respect to that state. (In a tree with multiple roots, some vertices will have multiple parents, and the conditions of all parents are evaluated.) The instance is added to the list being generated, along with a record of the condition evaluation result. If the condition failed then text describing the failure is stored (see Section 4.3). After the list is generated, the user chooses instances as usual. When the user chooses an instance for which a condition failed, the stored text is used to generate feedback.

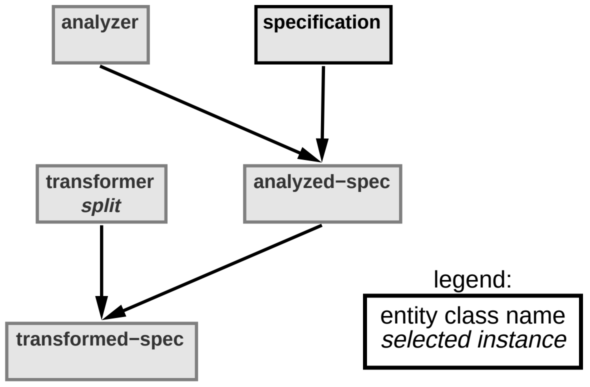

Figure 5: Hercules User Interface View of a Task Tree

There are two special cases involving missing instance selections for siblings of the vertex being resolved. The first case is where the sibling is a leaf, as when resolving the **specification** vertex in Figure 5, since no instance is selected for the **analyzer** leaf. To illustrate, assume that the **analyzed-spec** condition is changed to `!?annot@2 && ??annotator@1`, where an `annotator` attribute is attached to **analyzer** tool instances which annotate specifications. Even though no instance is selected for **analyzer**, it must be possible to evaluate the condition when resolving the **specification** vertex. Also, it is important not to over-constrain the list of instances. These requirements are met using *lenient* evaluation: condition literals that refer to unknowns evaluate to *true*, and thus those literals are effectively removed.

The second case occurs when an interior sibling has no selected instance. This will be the case when resolving the **transformer** vertex in Figure 5, since no instance is selected for the **analyzed-spec** vertex. It must be possible to evaluate the **transformed-spec** condition, and the evaluation must not be over-constraining. Again, these requirements are satisfied using lenient evaluation.

Note that no instance needs to be selected for the **analyzed-spec** vertex for the task to become resolved. When the task is run, the **analyzer** tool will be invoked first to create a new **analyzed-spec** instance, and then the **transformer** tool will be invoked to create a new **transformed-spec** instance using the new **analyzed-spec instance**. Since a lenient evaluation of the **transformed-spec** condition was performed, the condition must be evaluated again after creating the new **analyzed-spec** instance to check that the condition still holds — if it does not, then task execution must be aborted, and condition failure feedback must be given to the user.

4. Implementation

Changes were needed in the task database, the Cyclops encapsulation methods, the task schema, and the Hercules User Interface.

4.1 Task Database Changes

The main change to the task database was the addition of general attributes. Attributes have string values, there are no restrictions such as “instances of this entity class must have these attributes,” and strings are interpreted as numbers in numeric contexts.

4.2 Cyclops Changes

Attributes are meta-data for instances, with semantic meaning due to conditions. Thus, attribute creation must be considered carefully. There are two possibilities: attributes may be created by the task manager or by the tools. It might be possible for the task manager to create attributes in some cases. For example, for the schema in Figure 4 it might suffice for the task manager to attach `annot` attributes to new **analyzed-spec** instances. However, this will not work for tools such as editors, for which it may be impossible to know what attributes should be attached to new instances. It is better to have the tools (or, more specifically, the tool encapsulations) create attributes. This requires changes to Cyclops. When Hercules invokes a tool, Cyclops executes the tool’s encapsulation, and the encapsulation runs the tool. When the tool finishes, the encapsulation sends completion information back to Cyclops. We have added the ability for the encapsulation to send back a set of attributes for the new instance. This, then, is a very general way to capture attribute creation semantics in the framework, since encapsulations are “in” the framework.

4.3 Task Schema Changes

Above we stated that conditions are Boolean combinations of literals that test attributes. In fact, conditions are restricted to a sum-of-products form, as indicated by this pseudo-syntax:

condition	::=	*term* || *term* || ...
term	::=	*literal* **&&** *literal* **&&** ...
literal	::=	*exist-op attr-deref*
	::=	*str-operand str-op str-operand*
	::=	*num-operand num-op num-operand*
exist-op	::=	**??**, **!?**
str-op	::=	**eq**, **ne**, **gt**, **ge**, **lt**, **le**
num-op	::=	==, !=, >, >=, <, <=
str-operand	::=	**NAME@***number*
	::=	**USER@***number*
	::=	*attr-deref*
	::=	*string*
num-operand	::=	*attr-deref*
	::=	*number*
attr-deref	::=	*name***@***number*

Attributes are accessed using the *name@number* syntax. Edges leaving an entity class are numbered so they can be identified, as shown in Figure 4. The *name@number* syntax means “the value of attribute *name* on the instance at the end of the edge numbered *number*.” Built-in meta-data is accessed using special names: **NAME** for instance name and **USER** for creator username.

The sum-of-products (SOP) form has two advantages. First, it enables the simple generation of condition failure messages. An English description of the failure of a literal is easy to generate, and the SOP form makes it easy to produce a description for an entire condition (e.g., “`Condition failed for parent analyzed-spec: !?annot@2`”). Allowing general Boolean expressions would make failure isolation hard. Secondly, the SOP form enables the useful semantics of lenient evaluation, in that a literal that refers to an unknown can affect only one term.

4.4 User Interface Changes

Conditions cause instance selection at one vertex to be affected by selections at other vertices, so users should be allowed to see all instances, even those that cause condition failure. If the user tries to select such an instance then a feedback message is given which describes the condition failure and selection is rejected.

5. Results

The example of Figure 4 showed how conditions are useful for order enforcement. This example is also useful for illustrating the management of multiple tool instances. The real transformation tool of interest performs one of two operations, *split* or *merge*, so there are two encapsulations (and thus instances) of the tool. It is possible to define a single **transformer** class for both instances, even though they have different input requirements: split requires the specification to be annotated, but merge can be performed on any specification. This is captured using conditions as illustrated in Figure 6. A `split` attribute is attached to the **split** instance of the transformer tool. Then the condition on **transformed-spec** `!?split@3 || ??annot@4` means that either the **merge** tool is used (i.e., no constraint on the input specification) or, if the **split** tool is used, the input specification must be annotated. This allows the designer to define a "transformation" task, where the selection of the specific operation is deferred to task execution time, with tool input restrictions enforced by the condition.

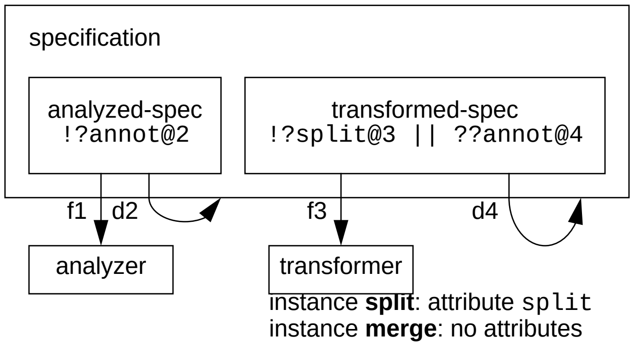

Figure 6: One Tool Class, Multiple Tool Functions

Conditions can be used to ensure that tools and data "match." This is illustrated by the example in Figure 7, where two **circuit-simulator** tools support different input format "levels": HSPICE supports input extensions not known by SPICE. This is handled by attaching a `lev` attribute to all **circuit-simulator** and **netlist** instances, where the attribute value indicates the input format level supported by the tool: 1 for **spice** and 2 for **hspice**. Then the condition `lev@2 <= lev@1` constrains selection so that the tool is able to handle the data. In the figure, the selection of the **spice** tool and the **low pow** data will cause the condition to fail.

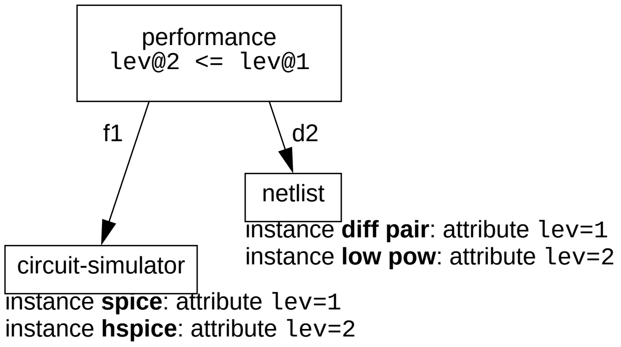
performance
lev@2 <= lev@1
f1
d2
netlist
instance **diff pair**: attribute `lev=1`
instance **low pow**: attribute `lev=2`
circuit-simulator
instance **spice**: attribute `lev=1`
instance **hspice**: attribute `lev=2`

Figure 7: One Tool Class, Multiple Tool Variants

Another example involves "data coherence." The design flow in Figure 8 shows the behavioral synthesis tasks of scheduling and allocation. The **alloc** tool data inputs must be *coherent*, i.e., the **control flow** must have been derived from the **value trace**.

A simple method of ensuring data coherence using conditions is illustrated in Figure 9. A `ver` attribute is attached to all the data

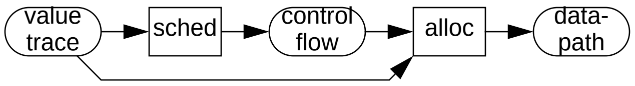

Figure 8: A Design Flow Requiring Data Coherence

instances to keep track of "versions." The scheduler tool encapsulation copies the version from the input **value trace** to the output **control flow**, and the condition `ver@1 == ver@2` ensures that the data instance inputs to the **allocator** tool are coherent.

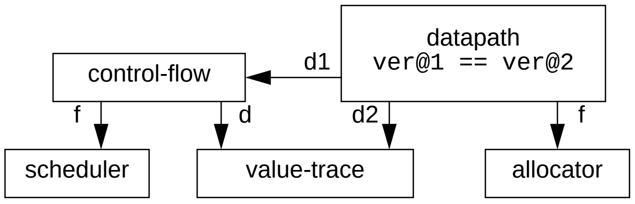

Figure 9: A Task Schema to Ensure Data Coherence

Finally, conditions are more general than the similarly-motivated capabilities described in [5]. The constraints described therein are tightly tied to the framework semantics, and are thus likely to have difficulty handling unanticipated constraint needs.

6. Conclusion

The task schema concept as originally embodied in the Hercules Task Manager was good for task creation but insufficient for task resolution, because resource selection constraints often cannot be captured as dependency relationships. We have overcome this by adding to task schemas conditions which test instance meta-data. Selection is constrained by evaluating the conditions for each proposed selection. Using a sum-of-products form ensures that instance selection is not overly constrained, and allows the easy generation of English feedback descriptions of condition failures. The result is a good balance of power and generality: conditions provide graceful solutions to real problems, and are also easy to use and understand.

Acknowledgments

The authors would like to thank the reviewers for their detailed and helpful comments. This research was supported in part by the Semiconductor Research Corporation, contract DC-068.

References

[1] J.B. Brockman and S.W. Director, "The Schema-Based Approach to Workflow Management," *IEEE Trans. on CAD*, vol. 14, no. 10, October 1995, pp. 1257-67.

[2] J.B. Brockman et al., "The Odyssey CAD Framework," *DATC Newsletter on Design Automation*, Spring 1992.

[3] T.F. Cobourn, "Resource Management for CAD Frameworks," Ph.D. dissertation, Carnegie Mellon University, July 1992, Research Report No. CMUCAD-92-39.

[4] M.F. Jacome and S.W. Director, "A Formal Basis for Design Process Planning and Management," *Int. Conf. on Computer-Aided Design*, November 1994, pp. 516-21.

[5] P. van der Wolf, O. ten Bosch, and A. van der Hoeven, "An Enhanced Flow Model for Constraint Handling in Hierarchical Multi-View Design Environments," *Int. Conf. on Computer-Aided Design*, November 1994, pp. 500–7.