Automatic Workflow Generation

Vladimir A. Shepelev*, Stephen W. Director**

*Research Institute for VLSI CAD Systems of Russian Academy of Sciences, Moscow, Russia
**Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA

Abstract

As the number and diversity of computer-aided VLSI design tools grows, there is an increasing interest in workflow management. In this paper we describe an enhancement to the task schema approach to workflow management that allows for the automatic generation of workflows. Such a capability can significantly enhance designer productivity. It has been implemented in the Dedal program.

1. Introduction

As the number and diversity of computer-aided design tools used by VLSI circuit designers continues to grow, there is an increasing need for tools that can help manage the design process. One such class of tools is known as workflow managers which help to organize, monitor, and automate complex processes. Through the use of *workflow management*, also known as *task management*, designers are freed up to think in terms of tool-independent procedures rather than in terms of specific application programs and data files.

Most approaches to task management are based on flow descriptions that must be prepared in advance thereby making them somewhat inflexible. Examples of such approaches are those based on flow maps to describe design tasks [2-4] and those that use extension languages to describe tasks [5, 6]. In [7-10], an alternate approach to task management was described. This approach, embodied in the Hercules Task Manager, employs the *task schema* to specify the rules by which tools and data may be combined to create a flow. It maintains the advantages of previous approaches while giving the designer maximal control over the design process. Through the realization of *dynamically defined flows* [9], designers gain significant freedom and flexibility resulting in a reduction of design cycle time.

In this paper we describe an enhancement to the task schema based approach to workflow management that allows for the automatic generation of workflows. This capability, which can significantly enhance designer productivity, has been implemented in the Dedal program.

2. The task schema

Our approach to automatic flow generation is based on a graph theoretic representation of the task schema that is employed in the Hercules Task Manager. We will show that such a representation allows us to develop an efficient algorithm for workflow synthesis. While a complete description of the task schema is beyond the scope of this paper, we need to introduce a few key concepts to facilitate our discussion. A more detailed description of the task schema can be found in [8] and [9]. Basically, the task schema captures the dependencies between all design entities (both tools and data) available to the designer and specifies construction rules to be used to create design flows that when executed realize design tasks. [8]. Fig. 1. is an example of a small task schema that we use here for illustrative purposes. Space limitations preclude inclusion of a more extensive task schema, of which what is shown here is a part.

Data entities may be either *simple* or *compound*. A compound entity is a set of simple entities that share some common properties. In the schema shown in Fig. 1., the entities labeled Edited Circuit and Extracted Circuit are simple entities while the entity labeled Layout is a compound entity.

A *simple design task*, which can be thought of as an elementary design function, consists of a target entity, which must be simple, and one or more input entities, which may be either simple or compound. The use of a compound entity as an input means that any of simple entities that constitute the compound entity can be used as inputs to the design task.

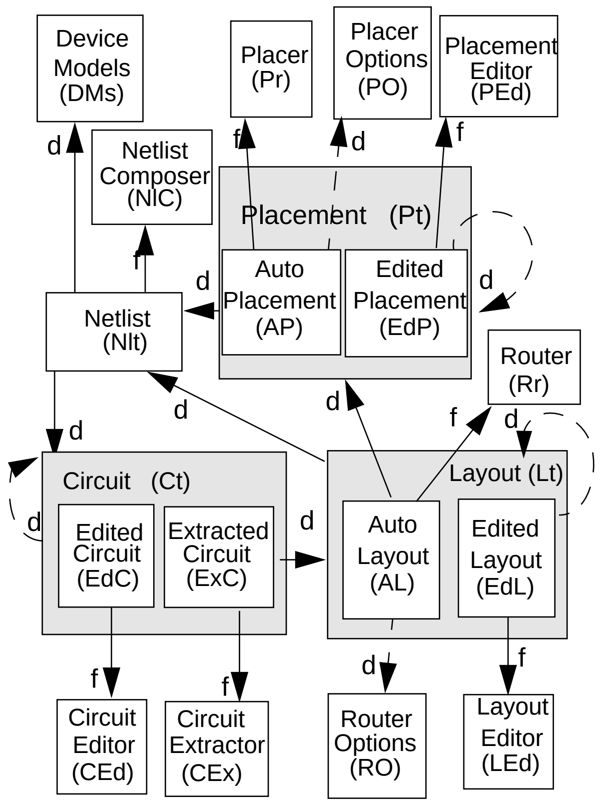


Fig. 1. An example of a task schema. The dependencies between entities is explicitly shown. The letter d denotes a data dependency and the letter f denotes a functional entity. The arrows point from resulting data entities to the entities that are used to create it. Note that a data entity may depend on itself, especially if it created with an editor.

During the course of design, *instances* of design entities are created, either as the result of task execution or the encapsulation of tool or data file. Each instance associated with a simple entity is described in terms of a vector of components. For a tool entity, this vector might contain the name of the tool, the computational expense associated with executing the tool, the expected quality of result produced by the tool, etc. For a data entity, this vector would probably contain the name of file, the size of file, the quality or precision of the data, etc.

The *compound entity graph* is a graph whose nodes correspond to a compound entity, a set of corresponding simple entities, and a set of instances associated with each simple entity. Arcs (directed edges) of the compound entity graph connect the compound entity with each of its simple entities and edges connect simple entities with their instances.

A *simple design task graph* is a partially directed graph with nodes corresponding to the input and functional entities, the target entity, and instances of the input and functional entities. Edges and arcs indicate the relations between design entities as well as between entities and their instances.

In what follows, we will only consider a that portion of the task schema that is shown in Fig. 2..The *task schema graph* is a graph assembled from the compound entity and simple design task graphs. As an example, Fig. 2.. shows the task schema graph associated with the task

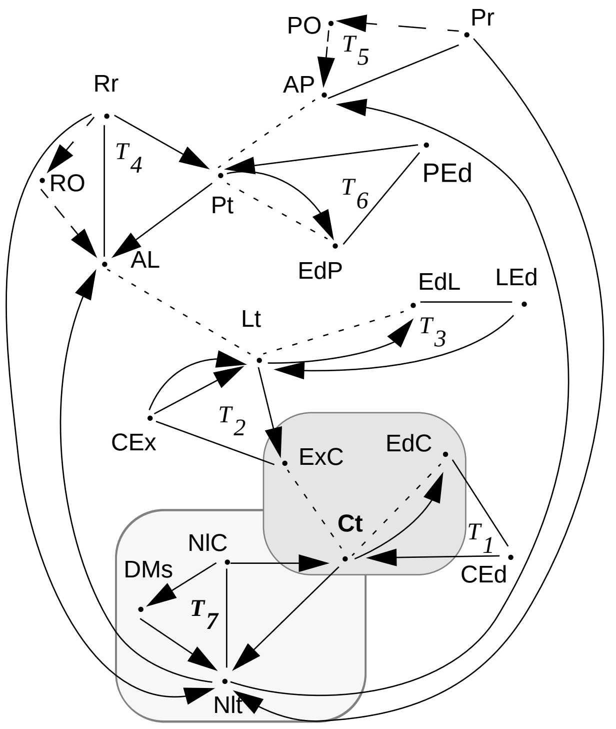


Fig. 2. Task schema graph

schema of Fig. 1.. For illustrative purposes, the compound entity graph associated with the compound entity Circuit (Ct) (dotted lines) and the simple design task graph associated with the simple design task T_9 , are marked in this figure. For the sake of simplicity, instances are not shown.

An entity is called a *defined entity* if it has at least one instance associated with it. A simple design task is *executable* if all of its input entities are defined input entities. An executable simple design task is said to be *exclusive* if each of its input entities has only one instance.

3. Automatic flow generation

Our method for automatic flow generation is based on the creation of an *expanded task schema graph* from the task schema graph, finding special *chains* in this graph, and then transforming these chains into a flow.

3.1. Expanded task schema graph

An *expanded task schema graph* is a task schema graph that has been modified so that it contains only simple entities. Removal of compound entities results in the creation of alternative simple design tasks. Fig. 3.

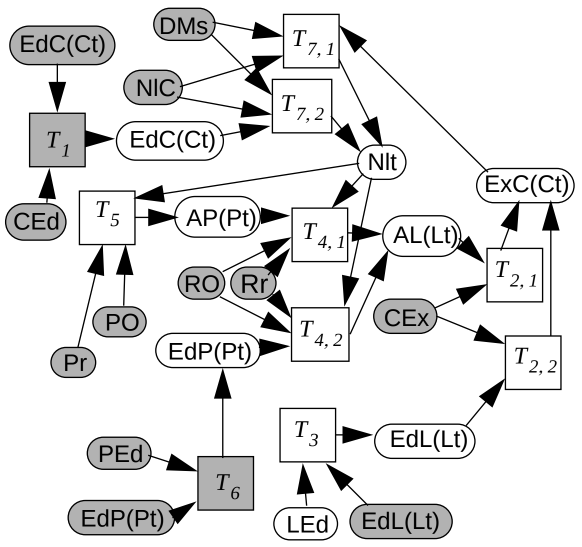


Fig. 3. The expanded task schema graph associated with the task schema graph of Fig. 2.

illustrates the expanded graph created from the task schema graph shown in Fig. 2. Note, the duplication of simple design tasks, as mentioned above. For example task T_2 in the original graph is transformed into the alternative tasks $T_{2,1}$ and $T_{2,2}$ in the expanded graph. The defined entities and defined simple design tasks are shown in gray. Note that T_1 is ready to execute. The *input leaf nodes* of an expanded graph are those nodes that are inputs to at least one simple design task but are not outputs of any simple design tasks.

A *flow* is a subgraph of an expanded graph that does not contain directed cycles, and if a simple design task belongs to the subgraph, all entities associated with the simple design task also belong to the subgraph. As with expanded graphs, we can identify the *input leaf nodes* of a flow. The *defined flow* is a flow in which an exclusive executable simple design task as been chosen for every simple design task and for which all entities that correspond to input leaf nodes, and only these entities, are defined entities.

A flow may be viewed as an alternating succession of entities and simple design tasks that connect a set of input leaf nodes to one or more target nodes. We refer to an alternating succession of entities and simple design tasks as a *chain*.

3.2. Flow construction method

We can now describe the flow generation algorithm.

Step 1: We begin with the identification of executable simple design tasks in the expanded task schema graph.

This is accomplished by considering each simple entity associated with each simple design task in the task schema graph and selecting an instance that satisfies some suitable criteria. For example an instance may be chosen, based on its characterization vector, to realize a "fast flow", an "economical flow," etc.

Step 2: Next we identify all chains that connect each of the target entities to executable simple design tasks

This may be accomplished by forming an *initial entities front* that includes all of the target entities of the future flow.

Then, we form the front of executable simple design tasks that has as its target entities in the entities in the initial entities front. At the same time we begin the creation of chains that connect these executable simple design tasks with the target entities. We also form pointers connecting every such executable simple design task with all its chains. If some executable simple design task is exclusive, we include all its chains into the initial set of chains. If all of the executable simple design tasks in the front are exclusive, we have completed Step 2 and proceed to Step 3.

Otherwise, we form the next entities front which is made up of the input entities associated with the current front of executable simple design tasks, and at the same time, create chains connecting these input entities with the target entities of the flow. Of course, we check the chains in order to avoid cycles.

This process is repeated until all of the executable simple design tasks in the front are exclusive.

Step 3: Transformation of the initial set of chains into minimal set.

To minimize the initial set of chains, we form the initial executable simple design tasks front which includes all exclusive executable simple design tasks from the initial set of chains. We then simulate the execution of these exclusive executable simple design tasks. Suppose, an exclusive executable simple design task has just executed. We then remove any alternative exclusive executable simple design tasks to realize a new (reduced) set of chains.

The next front is then formed by including all non exclusive executable simple design task from the current front and executable simple design tasks which have inputs connected to the exclusive executable simple design task from the current front.

If all target entities of the flow are not defined entities yet we analyze the actual front. If it does not contain any exclusive executable simple design task there is no flow which has to be created; otherwise, we simulate the execution of the exclusive executable simple design task, create the next front, and repeat this process.

If all target entities of the flow are defined entities we transform the current set of chains into a determining set: we analyse each chain beginning with the target entity, and, if some executable simple design task contains a defined input entity, we form the chain from the target to this entity.

3.3. An example

Again consider the expanded task schema graph shown in Fig. 3. and assume that we are interested in generating flows that will result in realizing the target entities $\mathrm{ExC}\,(\mathrm{Ct})$. To determine the initial set of chains we observe that the target entities are realized by the simple design tasks $\{T_{2,1}, T_{2,2}\}$; which in turn depend on the entities $\{\mathrm{AL}\,(\mathrm{Lt}), \mathrm{EdL}\,(\mathrm{Lt})\}$. Working backwards in this manner we can identify the following alternating sequences of simple design tasks and entities: $\{T_3, T_{4,1}, T_{4,2}\}$; $\{\mathrm{Nlt}, \mathrm{AP}\,(\mathrm{Pt}), \mathrm{EdP}\,(\mathrm{Pt}), \mathrm{LEd}\}$; $\{T_{7,1}, T_{7,2}, \boldsymbol{T_6}, \boldsymbol{T_5}\}$; $\{\mathrm{EdC}\,(\mathrm{Ct})\}$; $\{\boldsymbol{T_1}\}$, where executable simple design tasks are shown in bold. From this sequence we identify the initial set of chains:

$\cdot$ $\boldsymbol{T_1}, \mathbf{EdC}\,(\mathbf{Ct}), \boldsymbol{T_{7,2}}, \mathbf{Nlt}, T_{4,1}, \mathrm{AL}\,(\mathrm{Lt}), T_{2,1}, \mathrm{ExC}\,(\mathrm{Ct})$

$\boldsymbol{T_1}, \mathbf{EdC}\,(\mathbf{Ct}), \boldsymbol{T_{7,2}}, \mathbf{Nlt}, \boldsymbol{T_{4,2}}, \mathrm{AL}\,(\mathrm{Lt}), T_{2,1}, \mathrm{ExC}\,(\mathrm{Ct})$

- $\boldsymbol{T_1}, \mathbf{EdC}\,(\mathbf{Ct}), \boldsymbol{T_{7,2}}, \mathbf{Nlt}, \boldsymbol{T_5}, \mathrm{AP}\,(\mathrm{Pt}), T_{4,1}, \mathrm{AL}\,(\mathrm{Lt}), T_{2,1}, \mathrm{ExC}\,(\mathrm{Ct})$

$\boldsymbol{T_6}, \mathbf{EdP}\,(\mathbf{Pt}), \boldsymbol{T_{4,2}}, \mathrm{AL}\,(\mathrm{Lt}), T_{2,1}, \mathrm{ExC}\,(\mathrm{Ct})$

We now minimize the initial set of chains. To do this we begin with the initial set of executable simple design tasks $\{\boldsymbol{T_1}, \boldsymbol{T_6}\}$. Once these tasks are executed, the next set of simple design tasks that become executable are $\{\boldsymbol{T_{7,2}}, \boldsymbol{T_{4,2}}\}$ as seen in Fig. 4. If we simulate the execution of $\boldsymbol{T_{4,2}}$ we can remove chains containing $\boldsymbol{T_{4,1}}$ since it is an alternative simple design task for $\boldsymbol{T_{4,2}}$ (note that these are marked with a "minus" in Fig. 4).

We continue the minimization until the target entities are reached as shown below:

$\boldsymbol{T_1}, \mathbf{EdC}\,(\mathbf{Ct}), \boldsymbol{T_{7,2}}, \mathbf{Nlt}, \boldsymbol{T_{4,2}}, \mathrm{AL}\,(\mathrm{Lt}), T_{2,1}, \mathrm{ExC}\,(\mathrm{Ct})$

$\boldsymbol{T_6}, \mathbf{EdP}\,(\mathbf{Pt}), \boldsymbol{T_{4,2}}, \mathrm{AL}\,(\mathrm{Lt}), T_{2,1}, \mathrm{ExC}\,(\mathrm{Ct})$

We can then transform these chains into a determining set of chains connecting target with all defined entities. The flow corresponding to these chains is shown in Fig.4.

4. Dedal - a program for a task generation

The methodology discussed above for automatically generating flows has been implemented in the Dedal program which has been incorporated into the Odyssey CAD framework and interacts closely with the Hercules task management system [8]. Dedal can also work interactively with the designer to create specialized flows.

To illustrate the use of Dedal assume that the partial flow shown in Fig.5 exists and that we wish to generate the rest of the flow automatically. After choosing Generation from the main menu, the designer is able to select one or all leaf nodes as target entities using a pop-up menu. We are then ready to start flow synthesis.

If during flow generation Dedal encounters a compound entity for which no default has been specified, the user will be asked to choose among the simple entities associated with the compound entity. If no ambiguities exist, the flow will be created entirely automatically. The selection of a simple entity belonging to the compound entity Placement (Pt) is shown in Fig.6.

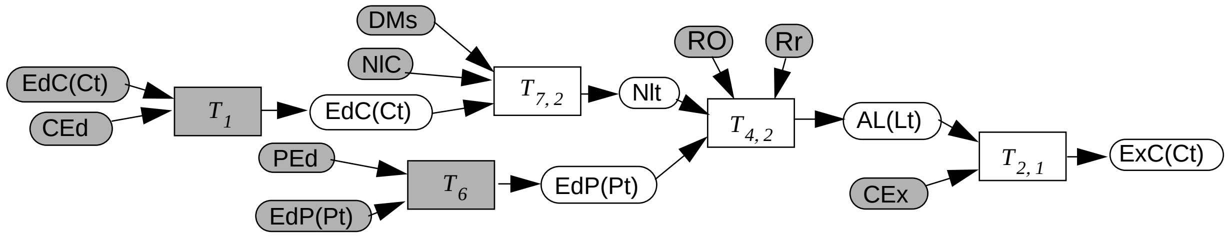


Fig. 4. The resulting flow

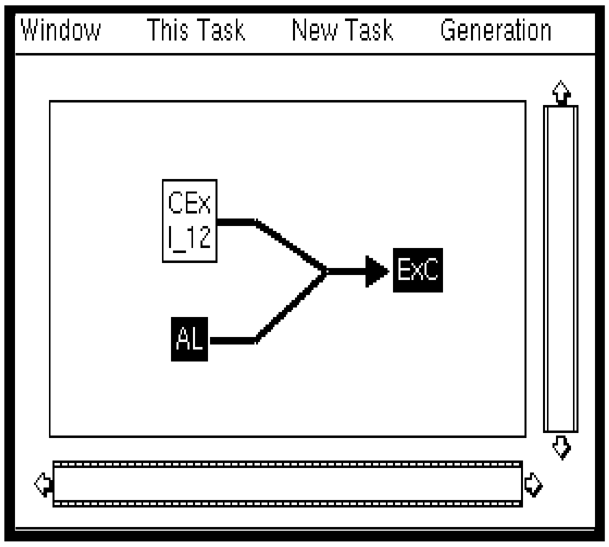


Fig. 5. The initial flow

Fig. 7 shows the flow that results after the user selected Edited Placement (EdP) belonging to the compound entity Placement (Pt).

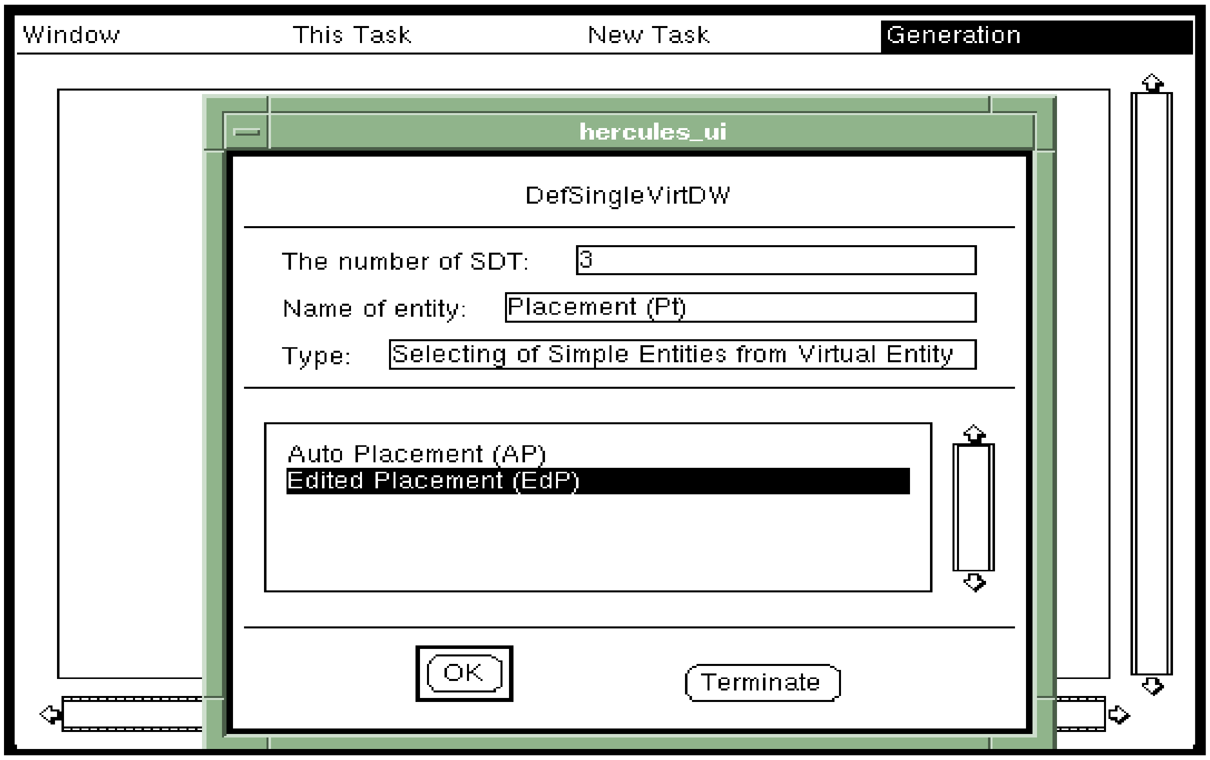
Dedal also has the ability to generate multi-target flows. Unfortunately, space limitations precludes an illustration of this capability.

5. Conclusions

We have presented a methodology for the automatic synthesis of design flows. The methodology is based on finding a determining set of chains in an expanded task schema graph. This methodology has been implemented in the Dedal computer program which has been incorporated into the Hercules task management system, a part of the Odyssey CAD framework. Thus users perceive Dedal as an extension of a task management system. Dedal can automatically generate and entire flow or allow the user to interact with it to create a specialized flow. While this paper only presents some initial results, we anticipate the development of optimal flow synthesis methods that take into account real characteristics of design instances.

6. References

[1] Timothy J. Barnes, David Harrison, A. Richard Newton, Rick L. Spickelmier. Electronic CAD Frameworks. - Kluwer Academic Publishers, 1992
[2] P. van den Hammer, M.A.Treffers. A Data Flow Based Architecture for CAD Framework. - Proceedings of the International Conference on Computer-Aided Design, IEEE, 1990, pp.482-485
[3] K.O. ten Bosch, P.Bingley, P.van der Wolf. Design Flow Management in the NELSIS CAD Framework. - Proceedings of 28th Design Automation Conference, 1991
[4] Gunnar Bartels, Peter Kist, Kees Schot, Mattie Sim. Flow Management Requirements of a Test Harness for Testing the Reliability of an Electronic CAD System. - Proceedings of EDAC-94, 1994
[5] W.Allen, D.Rosenthal, K.Fiduk. Distributed Methodology Management for Design-in-the-Large. - IEEE, 1990, pp.346-349
[6] W.Allen, D.Rosenthal, K.Fiduk. The MCC CAD Framework Methodology Management System. - Proceedings of 28th Design Automation Conference, 1991, pp.694-698
[7] J.B.Brockman, T.F.Cobourn, M.F.Jacome, and S.W.Director. The Odyssey CAD Framework. - IEEE DACT Newsletter on Design Authomation, Spring, 1992
[8] J.B.Brockman, S.W.Director. A Schema-Based Approach to CAD Task Management. - Proceedings of the Third IFIP WG 10.2 Workshop on Electronic Design Authomation Frameworks, Edited by T.Rhyne and F.J Ramming, Elsevier science publishers, 1992
[9] P.R.Sutton, J.B.Brockman, S.W.Director. Design Management Using Dinamically Defined Flows. - Proceedings of the 30th Design Authomation Conference, 1993, pp.648-653
[10] Eric W. Johnson, Jay B. Brockman. Incorporating Design Schedule Management into a Flow Management System. - Proceedings of the 32nd Design Authomation Conference, 1995



DefSingleVirtDW

The number of SDT: 3

Name of entity: Placement (Pt)

Type: Selecting of Simple Entities from Virtual Entity

Auto Placement (AP)

Edited Placement (EdP)

Fig. 6 The Selection of Simple Entities from a Compound Entity

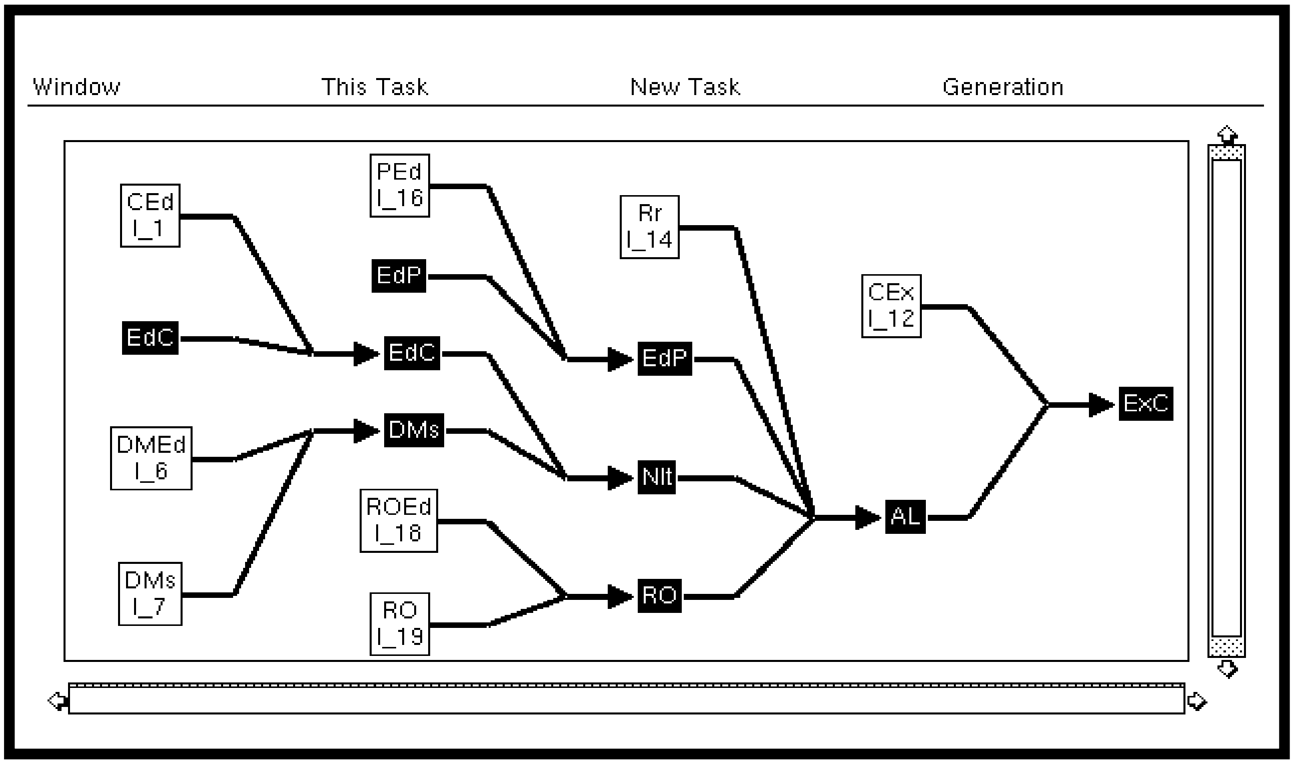


Fig. 7. The resulting flow