AN ENVIRONMENT FOR MODELING AND MANAGING THE PROCESS DESIGN PROCESS

S. GONNET, G. MANNARINO, H. LEONE [1] * and G. HENNING [2]

[1] *INGAR (CONICET) - GIPSI (Fac. Reg. Santa Fe, Universidad Tecnológica Nacional), Avellaneda 3657, 3000 Santa Fe, Argentina, hleone@ceride.gov.ar*
[2] *INTEC (Universidad Nacional del Litoral – CONICET), Güemes 3450, 3000 Santa Fe, Argentina, ghenning@intec.unl.edu.ar*

Keywords: Process Design Process Environment; Task Modeling Language.

Abstract
A three-layer object based environment architecture (*Client*, *Application Server* and *Server*) to support the modeling and management of the process design process is presented. It guides the design process and explicitly maintains the evolution of the design knowledge as a consequence of each design decision that is adopted during a particular design project. The *Client* encapsulates the interface of the designer to the design environment. The *Application Server* is responsible for managing the designer requirements by resorting to the Server data. It manages the definition of generic design tasks and the administration of the scenarios that are generated when a particular design process is carried out. Design tasks are defined according to the *Coordinates* language syntax. Basically, a design process is viewed as a set of tasks that are linked by temporal relationships and/or a set of resources they share. Generic design tasks can be instantiated and used in a specific design project. The *Server* contains the data that are created, used or modified by the Application Server.

1. Introduction

Computer-aided environments that support design processes have to (i) provide tools to structure the knowledge that is iteratively acquired, to administrate the design decisions that are adopted and their rationale, etc., (ii) integrate different design tools, (iii) coordinate the collaborative work among design experts, (iv) explore different design alternatives, etc. As there is not a unique design methodology, but a set of standard tasks that are usually carried out in a design process, a design environment should provide task patterns that each design team could combine according to their practices. There is very little previous work in this area in chemical engineering. Westerberg *et al.* (1997) have developed design support tools based on the *n-dim* environment. They have focused on supporting the management of information, a critical issue when addressing cooperative design. Bañares-Alcántara *et al.* (1995) have also made contributions to support the process design process. Their Epée environment makes possible the representation of the designer's intent and allows model traceability.

This paper presents a process design environment architecture whose goal is to support the execution of design projects, according to different design methods. Therefore, two object based components are defined: (i) a language for representing design methodologies; i.e., the design phases, the products to obtain at each stage, and the way the phases are linked by producing and consuming those products, and (ii) a conceptual tool for supporting the scenarios that are generated for a particular design problem.

The Coordinates language (Mannarino *et al.*, 1999) is used both to represent the different design methods and the outcomes of their application, i.e., process models representing a particular design. The version concept proposed by Gonnet *et al.* (1998) is employed for managing the evolution of a design process. Basically, the environment supports the explicit representation of the states through which a particular process model evolves during its design.

The environment proposed in this paper is supported by a three-layer architecture: the *Client*, the *Application Server* and the *Server* layers. Basically, the *Client* layer supplies the interface of the design environment to the designers; the *Application Server* manages the services the Client requires, such as the definition of generic design tasks or the implementation of the decisions adopted during a particular design project by making use of the data stored in the *Server*.

2. The Task Modeling Language

The different design tasks that can be carried out during a particular design process are represented according to the Coordinates language syntax. Coordinates is based on a three-layered architecture (Fig. 1). The Metamodel layer contains metaclasses representing the basic design modeling concepts, such as the notion of design task and resource. It also specifies the protocol required for defining and manipulating the different classes that inhabit the Model Layer. The Model Layer defines different design tasks and resource templates such as the ones that represent the structure of a flowsheet, an equipment item or the functionalities of an

optimization module. Tasks such as Solve-Material-Balance, Select-Reaction-Method, or Define-Recycle-Structure as well as the resources they require and transform during their execution, are identified at the Model Layer. This layer also defines the protocol for manipulating the entities of the Instance layer. Finally, the Instance Layer encompasses those entities generated by instantiating the Model Layer classes. Figure 1 shows instance relationships linking the three layers.

The *Coordinates* language describes a domain through different perspectives: *Task*, *Domain* and *Dynamic*. *Task* Models are used to represent a domain from a functional point of view. *Domain* Models put emphasis on the domain entities and their static relationships. Finally, *Dynamic* Models focus on the way the domain resources evolve during their lifecycle and how they interact in order to achieve the domain goals.

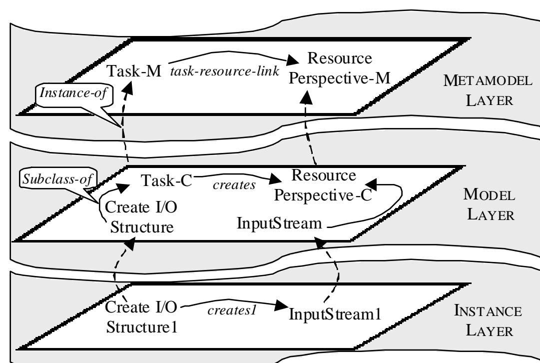

Fig. 1. Coordinates language

The *Task Metamodel* components (Fig. 2) can be used to prescribe design tasks. A *Task* metaclass (*Task-M*) represents an activity to be carried out. It makes use of different resources during its execution. A *Resource-M* is a conceptual or physical entity that can provide different services. It represents a piece of information, a design expert, or any computer design tool. Not every service a *Resource-M* can supply is of interest to every *Task-M*. Therefore, the resource perspective (*ResourcePersp-M*) concept is introduced in order to filter those aspects of a *Resource-M* that are relevant in a given context. *Tasks-M* are linked to *ResourcePersps-M* by resorting to the *task-resourcePersp-link-M*. The meaning of the *task-resourcePersp-links-M* is expressed by making use of the *state* concept (*ResourcePerspState-M*). A state represents a “snapshot” of a *ResourcePersp-M* at a certain moment. The *task-resourcePersp-link-M* is specialized into the *uses*, *employs*, *modifies*, *creates* and *eliminates* relationships. It is said that a *ResourcePersp-M* is *used* if the state it needs to be in order to participate in the *Task-M* is the same to the one it assumes when the *Task-M* has finished. The *creates* relationship represents the fact that a new *ResourcePersp-M* appears in the domain as a consequence of the *Task-M* execution. The link *eliminates* is the inverse of the *creates* one. Finally, the *modifies* link indicates that the *ResourcePersp-M* suffers a change of state due to its participation in the *Task-M*.

As seen in the previous paragraphs tasks relate among themselves indirectly by means of the resources they operate on. However, two tasks can be directly linked through explicit temporal relationships (Allen, 1983). The fact that a Task may have different endings is explicitly represented by resorting to the *TaskMode* (*TaskMode-M* language element) concept. Moreover, tasks can be described at different abstraction levels, according to the complexity of the design activity that is being modeled. Hence, a task can be decomposed into subtasks. However, as there may exist alternative ways of disaggregating a given task, the Task Decomposition concept is introduced. A *TaskDecomp-M* encapsulates a particular way of decomposing a *Task-M*, under a specific *TaskMode-M*. Each *Task-M* precisely identifies when it can assume the structure specified by each associated *TaskDecomp-M*. Actually, a set of task preconditions identifies the states its associated *ResourcePersps-M* need to assume for the *Task-M* to be executed as specified by each Task Decomposition.

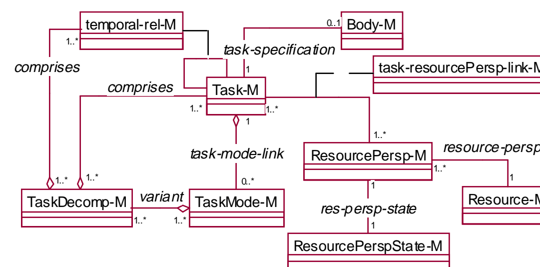

Fig. 2. Task Metamodel.

3. The design process environment architecture

As it was introduced in the previous section, the *Task Metamodel* can be used to define the generic design tasks associated to a particular synthesis and design methodology. Thus, at a class level (Model Layer), generic design models are represented. However, when the design activities of a particular project are to be modeled, generic tasks have to be instantiated and executed. The instantiation process produces specific occurrences of the different classes specified at the Model Layer (Fig. 1). Basically, a chemical process is envisioned as a set of *ModelVersions*, where each Model Version represents a snapshot of the design process at a specific moment. In other words, the design environment naturally shows how a set of *ResourcePersps* (at the Instance Layer) evolves from initial to final states. The proposed environment makes explicit a process design history in terms of the decisions that are adopted at

each design step, and the effects of those decisions over each participating *ResourcePersp*. This description is modified each time a design task is executed. The model version management approach introduced in this paper provides an explicit mechanism to administrate the different *ModelVersions* being generated during the course of a design project, as design tasks are executed.

In order to manage the evolution of the states of the *ResourcePersps*, the Version Administration System (VAS) proposed by Gonnet *et al.* (1998) was specialized.

The term software architecture is usually employed in the software engineering jargon to describe an abstract representation of the components of a software system and their relationships. Therefore, a three-layered architecture approach was adopted to represent the VAS system (Fig. 3).

Fig. 3. The design process Architecture.

3.1 Client Tools component

The *Client* Side provides the required user interface for both (i) specifying the different design tasks and (ii) interacting during a particular design project by instantiating some tasks and resources or transforming some existing resources. At any moment, the Client can get information such as when a given *ResourcePersp* was created, which adopted decision transformed a particular *ResourcePersp*, why some *ResourcePersp* were generated in the model, etc.

3.2 Application Server layer

The *Application Server* provides the functionality for managing the *Client* Side requirements. Basically, this layer provides two main services: (i) the definition of design tasks classes and (ii) the administration of design process scenarios.

3.2.1 Design Tasks classes administration

A use case (Jacobson *et al.*, 1994) represents a given functionality a system provides which is specified in terms of a set of collaborating objects by resorting to the so-called Sequence diagram. In order to define the different design tasks, various use cases are identified (Fig. 4)

The Task Metamodel shown in Fig. 2 specifies how design tasks classes are defined and manipulated. One of the basic use cases is the creation of a generic design Task class, such as *Create AFD* (Abstract Flow Diagram), *Select Reaction Alternatives*, etc. The Sequence Diagram that appears in Fig. 5 shows that a task class creation is responsibility of the *Task-M* metaclass. The generated *Task* requires a duration and is also subclass of the *Task-C*, which defines the protocol for creating and manipulating the tasks at the Instance Layer. It can be seen that every class is a direct or indirect subclass of *Task-C*.

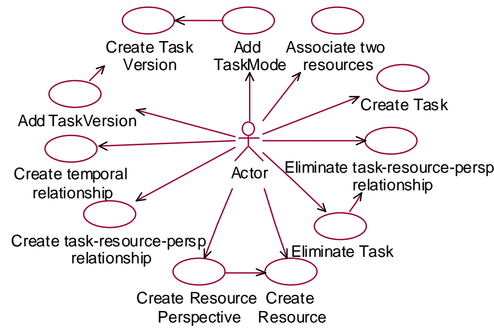

Fig. 4. Use cases associated to the Task Metamodel.

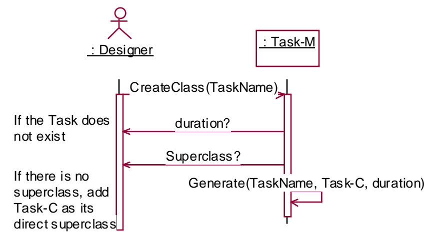

Fig. 5. Design Task class creation.

The Sequence Diagram that appears in Fig. 6 shows the process of creating an *InputStream ResourcePersp*. As a *ResourcePersp* filters only those services of a *Resource* class relevant in a given context, it is a responsibility of a *Resource-M* the definition of a new *ResourcePersp* class. Therefore, the generation of *InputStream* is responsibility of the *Stream* class.

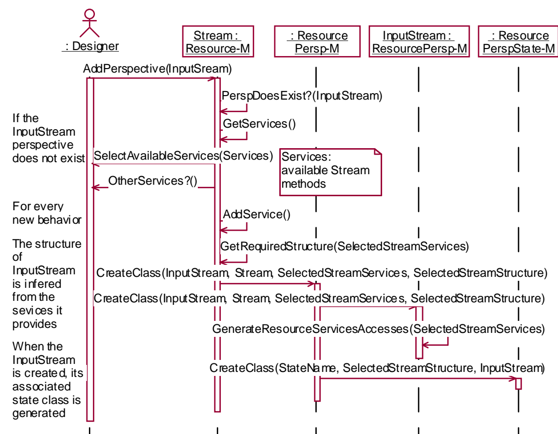

Fig. 6. An InputStream ResourcePersp is associated to the Stream Resource.

The services the *InputStream* provides are derived from the set of *Stream* services together with the new ones that may be added during the *ResourcePersp* creation. In this particular case, the *InputStream* views, among other characteristics, the destination unit and the flow value of its associated Resource (*Stream*). The definition of each *ResourcePersp* method consists of delegating the required service to its associated *Resource-C*. That means, for example, that whenever the message *GetDestinationUnit* is sent to the *InputStream*, it will be delegated to the *Stream* Resource, which actually contains the structure. Finally, the *InputStreamState* class is generated. This class represents a copy of the structure of the *Resource* (as view by the *ResourcePersp*) whose evolution will be documented. When a particular design project is executed, this class will give rise to a new instance each time the associated *InputStream* changes its state.

3.2.2 The Scenario support

The design environment structure that supports the administration of the process design process, shown in Fig. 7, is defined at the Model layer. Note that some of the classes that appear in the figure are instances of the metaclasses specified by using the *Task Metamodel*. More specifically, we make reference to the classes *Task-C, ResourcePersp-C, State-C, Resource-C* and *task-resourcePersp-link-C*.

A *ModelVersion-C* represents a snapshot of the chemical process being designed. More specifically, the products of the design process are encapsulated in *ResourcePersp* instances. Those products suffer different transformations, as expressed by the various states they can assume. Therefore, a *ModelVersion-C* explicitly stores the different resource perspective states (*State-C)* of the *ResourcePersps* instances involved in a particular design process. Figure 7 not only shows that design information is maintained, but also information relevant for (i) navigating among the different *Model versions* and (ii) documenting the transformations of the *ResourcePerps*. The basic links that are currently used to specify how a given chemical process evolves are the *addition*, *elimination* and *redefinition* links. The *addition* (*elimination*) link represents the fact that a *ResourcePersp* instance is generated (eliminated) as a consequence of executing an instance of *Task-C*. The *addition* and *elimination* links relate a specific *ModelVersion* instance with the added or deleted resource perspective state. The *redefinition* relationship links two resource perspectives states (instances of *State-C*) and not only expresses that the associated *ResourcePersp* instance suffered a change of state but also which was the actual change

The *Versions, Design Tasks and Repository Packages* that appear in Fig. 7 capture different aspects of a design project. The *Versions Package* encapsulates the evolution of the design knowledge in terms of the different states the involved resource perspectives assume. The *Task Package* makes explicit the design decisions that are adopted at each design step. Finally, the *Repository Package* identifies the resources and therefore the perspectives that can evolve in the design process scenario.

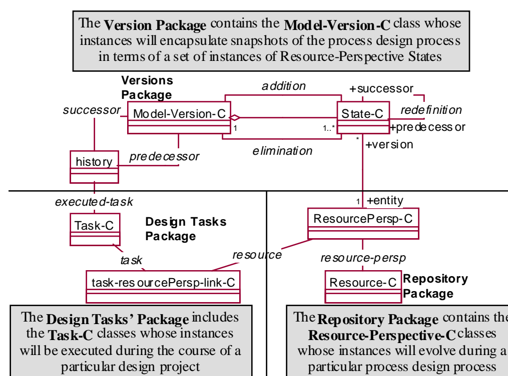

Fig. 7. The VAS structure

A design process scenario is carried out by iteratively executing the following two tasks: *Select the next task to execute* (Fig. 8) and *Execute the chosen task* (Fig. 9). Let us assume that some instances of *task-C* have been generated together with the *ResourcePersp* they either use, modify or eliminate. Only atomic tasks are considered in the interaction diagrams. The *Planner* entity that appears in the following figures is responsible for managing a particular design process

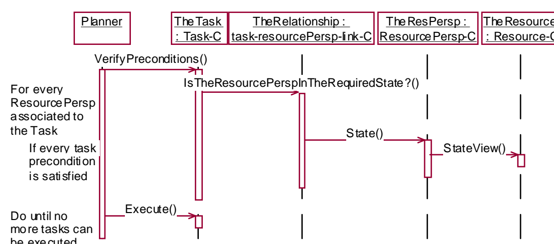

Fig. 8. Selection of the next Task to be executed

In order to guide the designer in a chemical process creation, the *Planner* has to select among the instantiated tasks the next one that can be carried out. The planner therefore, "suggests" the next step to take. Nevertheless, at any moment, the designer can instantiate new tasks or propose a different design activity. As mentioned before, a task is linked to a set of *ResourcePersps* by *task-resourcePersp-links*. Each link specifies the state the *ResourcePersp* needs to assume in order to participate in the associated task. Therefore, in order to determine whether a given task can be executed, a set of preconditions has to be verified. Figure 8 shows that each *task-resourcePersp-link* has to contrast the current resource perspective

state with the prescribed state. Each resource perspective state is generated by accessing in the associated *Resource* the values of the slots of interest. Therefore, the message *State()* that retrieves the resource perspective state is delegated to the associated resource by invoking the *StateView()* method.

Once the next step to take is chosen (*TheTask* from now on), it has to be executed. But before that, a new *ModelVersion-C* instance (*NewVersion*) has to be created so as to register the changes to be made, and thus maintaining the chemical process history. The *Execute()* message is sent to the *TheTask*, which delegates the responsibility to its associated *Body*, which encapsulates the actual task behavior. During the task execution, the involved resource perspectives exchange messages which are ultimately delegated to each corresponding *Resource*

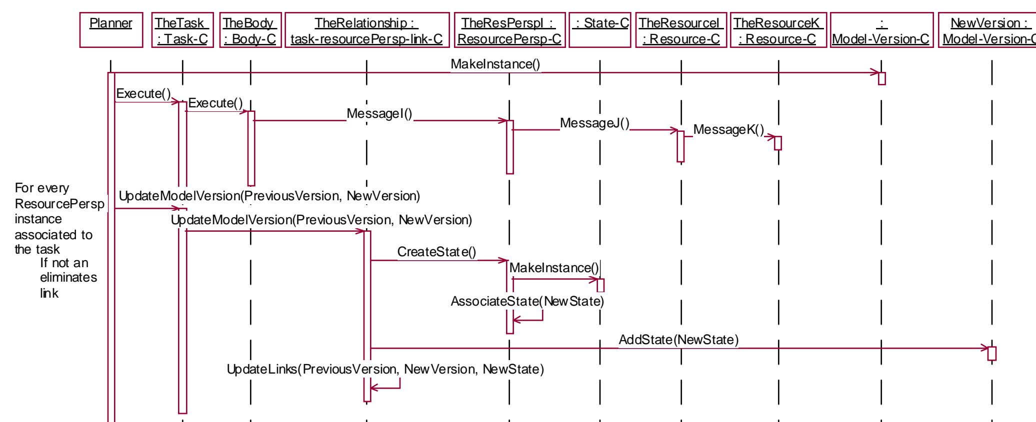

Fig. 9. Execution of the chosen task

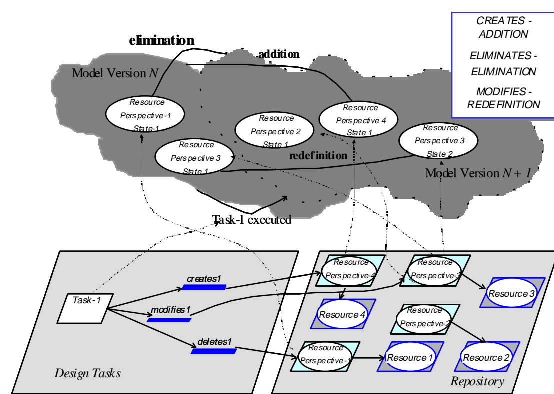

Fig. 10. Relationship among task-resourcePersp-links and VAS operations

Once the task finishes its execution, the design knowledge changes have to be documented. As the *task-resourcePersp-links* are the ones that describe the effects of the tasks over their associated resource perspectives, they are the ones responsible for updating the previous *ModelVersion*. Each *task-resourcePersp-link* gives rise to a new *ResourcePersp state* instance (*NewState*), with exception of the *eliminates* links. The created *NewState* instance is stored in the *NewVersion* Model Version and finally, the links between the previous Model Version (*PreviousVersion*), the *NewVersion* and the *NewState* are updated by invoking the *UpdateLinks* method. Figure 10 explicitly shows the relationships among the *task-resourcePersp-links* specified in the task model and the VAS history links. The semantics is the following: (i) the existence of a *creates* relationship in a task model gives rise to an *addition* link, (ii) an *eliminates* relationship derives in an *elimination* relationship and (iii) a *modifies* link generates a *redefinition* relationship

3.3 Data Access Server (DAS)

The *Server* side is responsible for storing and retrieving the information generated in every design project as well as the design tasks specifications.

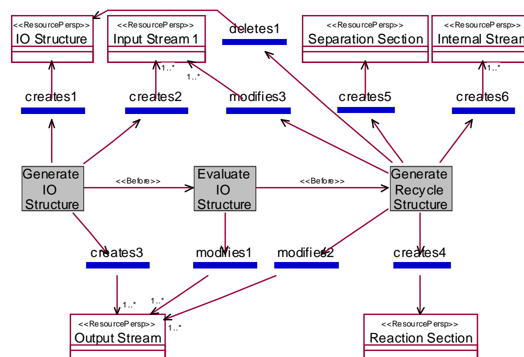

Fig. 11. Tasks involved in the example scenario

4. Example

In order to illustrate the scenario administration environment a small scenario is considered. Figure 11 shows the generic design tasks involved in the scenario as well as their associated *ResourcePersps*. The tasks

represent the initial steps taken during the creation of a flowsheet.

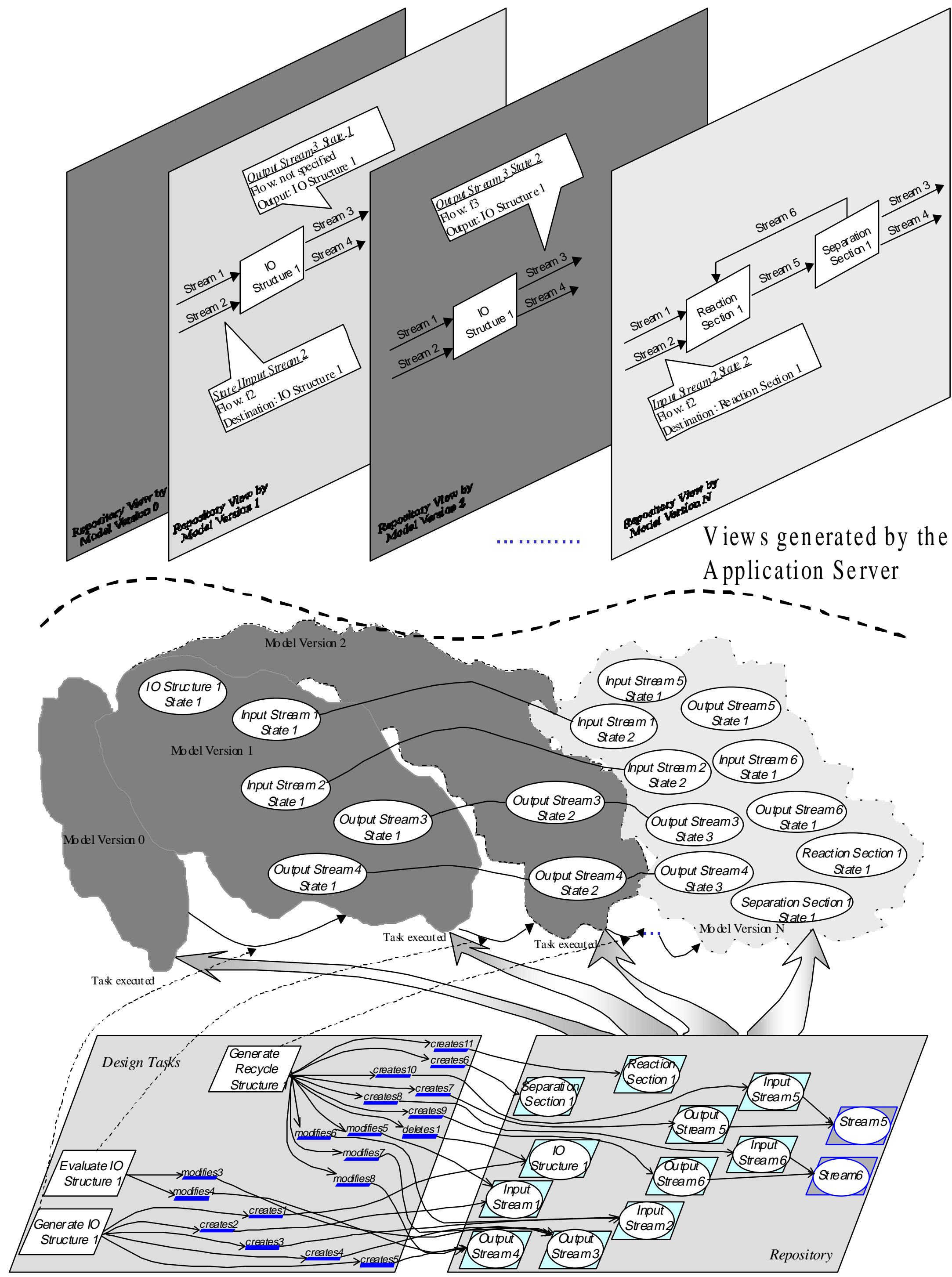

Fig. 12. Process design execution

Initially, an empty model Version (*Model Version 0*) is created (Fig. 12). An instance of task *Generate IO Structure1* does exist. As the task does not have any preconditions, it can be executed. *Model Version 1* is created. Task *Generate IO Structure1* is executed and the *IO Structure1* is created with two input streams

(ResourcePersps *Input Stream1*, *Input Stream2*) and two output streams (ResourcePersps *Output Stream1* and *Output Stream2*). The *Input Stream2* perspective only knows the destination and the flowrate (f2) of the *Stream2* Resource. Similarly, *Output Stream3* filters the origin and the flowrate of *Stream3*. The flowrate of *Stream3* has not been specified yet. The results of executing *Task Generate IO Structure 1* are graphically shown in the figure as "Repository View by Model Version 1". When the *Input Stream2* perspective is created an instance of the *State-C* class is generated (*Input Stream2 State1*) and stored in *Model Version 1*. *Input Stream2 State1* represents a snapshot of *Stream2* as viewed by the *Input Stream2* perspective. The remaining streams are similarly treated.

In order to capture the process history, a reference to the task *Generate IO Structure 1* is stored in the *history* relationship that links *Model Version 0* with *Model Version 1*. The relationship *addition* between *Model Version 0* and *Input Stream2 State1* is created. It makes explicit how the design knowledge evolved after executing task *Generate IO Structure 1*.

Task *Evaluate IO Structure* is then executed in order to specify the Output Streams flows

References

Allen, J. F., "Maintaining knowledge about temporal intervals", *Communications of the ACM* **26**, 832-843 (1983).

Bañares Alcántara, R., J. King and G.H. Ballinger, "Extending a Process Design Support Systems to Record Design Rationale", *AIChE Symposium Series No.* **304**, 332-335 (1995).

Gonnet, S., R. Holzer, H. Melgratti and H. Leone, "Administración de Versiones de Modelos en una Herramienta de Soporte para el Análisis y Diseño Orientados a Objetos". *Proceedings of the IV CACIC*, Neuquén, Argentina (1998).

Jacobson, I., Christerson M., Jonsson P. and Övergaad G., *Object-Oriented Software Engineering. A Use Case Driven Approach*, Addison-Wesley (1994).

Mannarino, G., G. Henning and H. Leone, "Coordinates: A Framework for Enterprise Modeling. Information Infrastructure Systems for Manufacturing II, J.J. Mills and F. Kimura (Eds.) Kluwer Academic Publishers (1999).

Westerberg, A., E. Subrahmanian, Y. Reich, S. Konda and the n-dim group, "Designing the Process Design Process", *Computers chem. Engng*. **21**, Suppl, S1-S9 (1997).

Received September 2, 1999.
Accepted for publication March 14, 2001.
Recommended by A. Bandoni.