

This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

<http://www.elsevier.com/locate/permissionusematerial>

A model for capturing and representing the engineering design process

Silvio Gonnet ^{a,*}, Gabriela Henning ^b, Horacio Leone ^a

^a CIDISI – INGARI/UTN – CONICET, Avellaneda 3657, 3000, Santa Fe, Argentina

^b INTEC (CONICET – UNL), Güemes 3450, 3000, Santa Fe, Argentina

Abstract

This paper presents a Collaborative Model for capturing and representing the engineering Design process (CoMoDe). CoMoDe is a deductive object-oriented model that, in relation to an engineering design process, is able to capture the different elements that participate in a design process in an integrated fashion. In particular, it is able to represent (i) the activities, operations, and actors that have generated each design product, (ii) the imposed requirements, and (iii) the rationale behind each decision. Furthermore, it also offers an explicit mechanism to represent and trace the different model versions that have participated in the design process. On such a basis, this proposal introduces specific procedures to handle various situations appearing in cooperative environments. They are: (i) different design teams perform independent concurrent activities on “a priori” independent parts of the artefact being designed and afterwards their results need to be made consistent; (ii) distinct teams concurrently work on slightly coupled parts of the artefact being designed and conflict handling must be addressed along their “parallel” course of actions.

© 2006 Elsevier Ltd. All rights reserved.

Keywords: Design process support; Collaborative design; Version management; Situation calculus; Deductive object base

1. Introduction

Engineering design, be it the design of a chemical process plant, an information system, or a mechanical piece, needs knowledge from the specific domain. However, it is possible to identify a common denominator for the design, its complexity and the resolution of problems being not well defined at all. Development of products in many engineering disciplines is a challenging task. Even for quite different types of products, development processes have strong common characteristics and features, such as the following:

- Design problems are inherently ill defined; therefore, the structure of the design process is not known in advance. It starts with a small set of requirements that include goals and constraints and evolves through subsequent

stages of increasing complexity in a non-linear manner. In most cases, there is a lack of a fully articulated methodology so that there is no clear distinction between solution stages. Furthermore, there might be a need for a backtracking process to change previously adopted decisions.

- During a design process, various models of the artefact being designed are generated. They differ in granularity, complexity, and associated assumptions; therefore, there is an explicit need for properly managing model versions.
- Once a design project is complete, what remains is mainly the “design product” (the models that were generated, detailed specifications of the resulting artefact, drawings, sketches, etc.). However, there is no explicit representation of how that product was obtained. More specifically, there is no trace of:
 - Which activity/ies originated a given product.
 - Which requirements were imposed.
 - Which actors performed a given activity.
 - Which is the underlying rationale behind a decision-type activity.

* Corresponding author. Tel.: +54 342 4534451; fax: +54 342 4553439.

E-mail addresses: sgonnet@ceride.gov.ar (S. Gonnet), ghenning@intec.unl.edu.ar (G. Henning), hleone@ceride.gov.ar (H. Leone).

- Due to their size and complexity or specific expertise needs, design problems are rarely tackled by individuals; design teams are the usual choice. Thus, teams of human experts in conjunction with computer-aided tools solve intricate problems by interacting cooperatively, sharing resources of various types and intermediate design products (i.e. models that were generated, detailed specifications of resulting artefacts, drawings, sketches, etc.).

Depending on the domain and on the problem being addressed, design methodologies can vary. Boyle (1989) proposes a classification that splits design into three broad methodologies: analytical, procedural, and experimental design. The concepts behind this classification are those of object, attributes, and operations as well as the different roles that are assigned to humans and machines in these classes of design methods. The three categories proposed by Boyle can be summarized as follows:

- Analytical or attribute-centered design, in which the attributes of the objects are used to determine the appropriate design actions. A design solution is automatically synthesized from the object attributes and the design objectives.
- Procedural or operation-centered design is based on using procedures to perform operations on an object with the aim of transforming it into one having the desired attributes.
- Experimental search or object-centered design involves working through an available set of objects in order to find one whose attributes best match the design objectives.

In this paper the focus is on procedural design, which is the most frequent case in engineering design. Design is viewed in a way similar to that of Boyle (1989) and other authors (Brown & Chandrasekaran, 1989; Mittal & Araya, 1992) who consider it as an iterative process that operates under a generate–test–analyze–advise–modify paradigm. As design artefacts are generated, they are checked against design objectives.

This scenario increases the need for information sharing and exchange. Accordingly, several proposals have arisen to tackle this issue. Most of them try to address the multiple design data representation problem that has appeared due to the various design tools used by designers during a design process (Marquardt & Nagl, 2004). Furthermore, engineering data management systems (EDM), product data management systems (PDM), and software configuration management systems (SCM) provide assistance in managing products of a development process (i.e. models, diagrams, documentation files, etc.) along their life cycle. Several models have been proposed in literature (Carnduff & Goonetillake, 2004) to enhance database system facilities that are used to group mutually consistent component versions together into useful configurations. As Westfechtel (1999) has pointed out, EDM, PDM, and SCM systems

focus on the products of development processes, neglecting the representation of the activities that have generated them. In consequence, they do not satisfy the need for keeping consistency, navigability, and traceability among models (and model's components) identified along the design process.

Once a design stage is complete, what remains is mainly the design product but there is no explicit representation of how this product was obtained. Even if such knowledge is found in documents, it is often scattered around technical domains and improperly categorized (Kitamura & Mizoguchi, 2003). Thus, most design knowledge still rests in the minds of experienced designers and is not available to be shared or to be employed (for specific advice) as needed (Liao, 2005). This issue has been dealt with by several authors (Madow & Pérez-de-la-Cruz, 2004; Roda, Poch, & Bañares-Alcántara, 2000; Westerberg, Subrahmanian, Reich, Konda, & the n-dim group, 1997) who acknowledge that the history of a design process, captured in a useful way, can form the basis for learning and reuse.

Consequently, to overcome such troubles, not only the data and their dependencies but also the design process where the information is created and used need to be thoroughly understood. Therefore, the pivotal element in developing any design support environment is modelling the design process itself in order to understand and articulate it. To do so, it is necessary to recognise the design activities that are performed to evolve from the initial design specifications to the final engineering design; at the same time, it is crucial to identify the design decisions associated with each activity, along with their corresponding assumptions, simplifications, and underlying rationale. In fact, it is this design experts' knowledge that has to become explicit.

This contribution addresses this issue by introducing CoMoDe, a Collaborative Model for capturing and representing the engineering Design process. CoMoDe is an integrated deductive object-oriented model that captures and represents the products being designed, the activities taking place in the design process, their associated contexts, and the adopted decisions at two different granularity levels. The proposed model is introduced in the next section and described more in detail in Section 3. Furthermore, CoMoDe also offers an explicit mechanism to capture and trace the different versions that have participated during engineering design processes. This mechanism is described in Section 4.

Throughout this contribution, we will consider design processes in different engineering disciplines. We will restrict the discussion to software and process systems engineering because our work has been mainly focused on these disciplines. The ideas presented in Sections 3 and 4 are exemplified by modelling the design of a chemical plant to produce polyamide-6.

On the other hand, cooperative tools and methodologies are currently available to face complex activities undertaken by designers, such as the ones that belong to the field of computer supported collaborative work (CSCW). Most

of them focus on communication (messaging) and coordination features (approval forms, workflow tools, video-conference tools) (Gzara Yesilbas & Lombard, 2004). However, there is an absence of tools to support the conflict management process (CMP), which takes care of conflict detection and resolution tasks.

One of the difficulties associated with the way CMP is usually tackled is related to one of the points outlined above. Once a design stage is complete, there is no explicit representation of how the design products were obtained. Therefore, on the basis of the knowledge represented by CoMoDe, Section 5 provides specific procedures to handle different situations that may arise as the result of working in a cooperative environment.

Finally, Section 6 emphasizes the main contributions of the model.

2. Representing how the design process is performed

Designers react contextually according to the domain knowledge they manage. In consequence, the integrated model here proposed aims to relate the context where a design activity is performed to the activity itself; otherwise, some information about the activity would be lost. Therefore, CoMoDe aims to capture not only the activities performed during the design process but also why (their underlying rationale) and when these activities were done and who (actors) executed them. However, at a more detailed level, activities operate on the results or products that are created along the design process, called *design objects*, by generating, deleting, modifying, and/or using them. These design objects also need to be considered in the model.

The previous analysis indicates that contexts of different granularity need to be handled by CoMoDe. On the one hand, there is an *activity context*, described in Section 3, which captures those tasks that are responsible for establishing how the design process advances by means of either implicit or explicit decision-making activities. On the other hand, there is an *operation context*, presented in Section 4, which is responsible for capturing how products under development are transformed along the design process. This process originates new contexts, based on which new activities will take place.

The proposed model is based on a hybrid approach that combines object-oriented technology and first order logic. Moreover, both object-oriented technology and situation calculus (Reiter, 2001; Scherl & Levesque, 2003) are employed for modelling the evolution of the design objects. The model has been formally specified in the O-Telos language (Jarke, Jeusfeld, & Quix, 2004), which successfully combines object-oriented and first order logic properties, and has been implemented in ConceptBase (Jarke et al., 2004), a deductive object-oriented data base manager. ConceptBase integrates techniques from deductive and object-oriented databases in the logical framework of the O-Telos' data model, a dialect of Telos (Mylopoulos, Borgida, Jarke,

& Koubarakis, 1990). Telos is a conceptual modelling language for representing knowledge about information systems. It is based on the core concepts of object-oriented technology, integrity constraints, and deductive rules. Telos resorts to the constraints and deductive rules modelling constructs to enforce model consistency and to provide deductive capabilities. In consequence, the object-oriented concepts supported by O-Telos are employed to map the objects defined in CoMoDe into a formal representational language. The rules and constraints expressed as first order formulas are used to encode the axioms of CoMoDe. Therefore, the design process model specified in O-Telos and implemented in ConceptBase allows the obtention of a formal model that is internally consistent and that can be used to derive other properties of the system specified.

3. Activity context

Engineering design processes are inherently complex. Therefore, any model aiming to capture them cannot be simple. From the point of view of supporting the design process, various essential aspects of the process need to be modelled, as described in the previous section. All the required design process elements cannot be included in just one model view, since such representation would be quite obscure. To avoid this problem, in CoMoDe, the *activity context* is modelled across the following representation spaces:

- *Process representation space*, representing the activities performed during a design project and their associated operations,
- *Requirement representation space*, modelling the imposed requirements,
- *Actor representation space*, keeping track of the actors who performed a given activity,
- *Artefact representation space*, representing the designed artefact,
- *Decision representation space*, capturing the underlying rationale behind each design-related decision.

Each representation space encapsulates a particular view of the design process model and has been formulated under the minimal ontological commitment design criteria (Gruher, 1995). Therefore, a representation space just focuses on the main concepts of its associated perspective. This allows the model to be specialised and further extended with the concepts of a particular design domain. Fig. 1 presents an overview of the activity context model, including the spaces mentioned above and the concepts belonging to each space.

The core concepts that are included in CoMoDe are the following:

- *Activities*: They are the tasks that are carried out during design processes, such as proposing a given separation

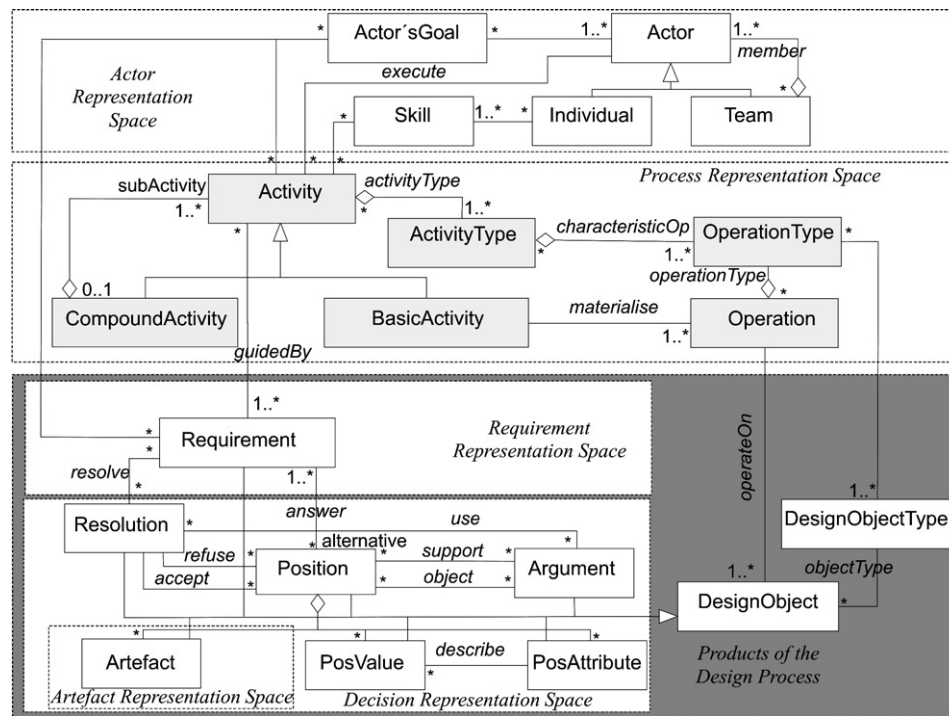


Fig. 1. Design process representation: activity context.

structure, analysing whether such structure satisfies the separation targets that were imposed, evaluating its economic potential, deciding among alternative separation schemes if the process systems engineering domain is considered.

- **Design objects:** They represent the different products of the design activities. Typical design objects are models of the artefact being designed, specifications to be met (i.e. stream purity specs, products' throughput for process system engineering, quality attributes such as modifiability or performance for software engineering), variable values (e.g. reflux ratios, number of stages of a separation unit, operating temperatures and pressures, etc). Naturally, these objects evolve as the design process takes place, giving rise to several *versions*, which in turn may comprise one or more *model versions*.
- **Requirements:** They specify design objectives or goals, as well as the functional and non-functional characteristics that the design products must satisfy. Then, they can be regarded as the driving force of a design process. Since activities operate on requirements, they are represented as design objects. They may also evolve as the design project proceeds.
- **Operations:** They are the ones that actually materialise the execution of design activities. In particular, operations materialise the so-called elemental or basic activities, the ones that cannot be further decomposed into subactivities. By doing this, operations transform design objects (i.e. by creating, deleting, or modifying them), thus allowing the evolution of their associated *versions*.
- **Actors:** They conduct the execution of activities and operations by having specific purposes, or pursuing cer-

tain objectives, while attempting to satisfy one or more requirements.

- **Model version:** Each model version represents a set of design objects within the context in which a given design activity is carried out. Thus, a model version is like a snapshot, providing a description of the state of the design process at a given moment.

Despite the separation of the activity context in different views (see Fig. 1), special attention is given to the relationships among concepts that pertain to distinct views, just as the *guidedBy* link existing between the *requirement* and *activity* concepts. The links that explicitly integrate the model elements are preserved when concepts are specialised. For instance, any specialisation of *requirement* will inherit the links it has with other concepts that are defined in CoMoDe.

3.1. Process representation space

As shown in Fig. 1, this space represents the activities being performed during a design process. The process representation space tries to capture the fact that the design process is carried out by a set of *activities* executed either by designers or by automated tools. Design activities may be described at various levels of abstraction. Thus, an *activity* may be decomposed into a set of *subactivities*, which may be organized according to a schedule or performed without a specified order. The relationship between an *activity* and its *subactivities*, which is depicted in Fig. 1 by an aggregation link, can be represented using first order logic by the predicate $subActivity(a_i, a_k)$, which means that

a_i is a subactivity of a_k . Taking into account this relationship, activities are classified into *basic* and *compound activities*. The first ones are represented in Fig. 1 by the *BasicActivity* class. It specialises the *Activity* class (expression (1)), and expression (2) imposes the fact that a *basic activity* cannot be further decomposed.

$$(\forall a_b) \text{basicActivity}(a_b) \Rightarrow \text{activity}(a_b) \quad (1)$$

$$(\forall a_b) \text{basicActivity}(a_b) \Leftrightarrow \neg(\exists a : \text{Activity}) \text{subActivity}(a, a_b) \quad (2)$$

A *compound activity* cannot be a leaf node in the activity hierarchy; thus any activity that is regarded as a compound activity is not a basic one (expression (3)). Fig. 1 represents compound activities by the *CompoundActivity* class, a specialisation of the *Activity* concept (expression (4)).

$$(\forall a : \text{Activity}) \text{compoundActivity}(a) \Leftrightarrow \neg \text{basicActivity}(a) \quad (3)$$

$$(\forall a_c) \text{compoundActivity}(a_c) \Rightarrow \text{activity}(a_c) \quad (4)$$

As indicated by expressions (5)–(7), the *subActivity* relationship is transitive, irreflexive, and asymmetric.

$$(\forall a_1 : \text{Activity}, \forall a_2, a_3 : \text{CompoundActivity}) \text{subActivity}(a_1, a_2) \wedge \text{subActivity}(a_2, a_3) \Rightarrow \text{subActivity}(a_1, a_3) \quad (5)$$

$$(\forall a : \text{CompoundActivity}) \neg \text{subActivity}(a, a) \quad (6)$$

$$(\forall a_1, a_2 : \text{CompoundActivity}) \text{subActivity}(a_1, a_2) \Rightarrow \neg \text{subActivity}(a_2, a_1) \quad (7)$$

Moreover, an activity cannot be a subactivity of two or more distinct activities that are not related by means of a *subactivity* relationship, as prescribed by expression (8).

$$(\forall a_1 : \text{Activity}, a_2, a_3 : \text{CompoundActivity}) \text{subActivity}(a_1, a_2) \wedge \text{subActivity}(a_2, a_3) \Rightarrow a_2 = a_3 \vee \text{subActivity}(a_2, a_3) \vee \text{subActivity}(a_3, a_2) \quad (8)$$

The recursive decomposition of *subactivities* leads to an overall activity structure. The activity structure bottoms out in *basic activities*. They are the ones that are actually executed in a design process. This idea is conceptualised in the model by introducing the *operation* concept. An *operation* is the basic transformational primitive action, representing an action on *design objects*. Indeed, operations prescribe how the design domain is changed by operating on design objects along the design process. These concepts are described more in detail in Section 4. Therefore, *operations* are responsible for materialising *basic activities*. This fact is represented by the predicate *materialise*(ϕ, a) (expression (9)), where ϕ represents the sequence of operations that materialises the *basic activity* a .

$$(\forall a) \text{basicActivity}(a) \Leftrightarrow (\exists \phi : \text{SequenceOfOperations}) \text{materialise}(\phi, a) \quad (9)$$

A *sequence of operations* ϕ is defined by resorting to a recursive expression (10), where *op* is an operation, λ the empty sequence and \bullet represents the concatenation between an *operation* and a *sequence of operations*.

$$\phi = \begin{cases} \lambda \\ op \bullet \phi \end{cases} \quad (10)$$

From the previous definitions, it can be inferred that an activity performs a certain operation if such operation is part of the sequence of operations that materialises the activity. This fact is represented by resorting to expressions (11.a) and (11.b). The first one corresponds to the case in which the activity is a basic one, while expression (11.b) corresponds to compound activities.

$$(\forall op : \text{Operation}, a) \text{basicActivity}(a) \wedge \text{perform}(a, op) \Leftrightarrow (\exists \phi : \text{SequenceOfOperations}) \text{materialise}(\phi, a) \wedge op \in \phi \quad (11.a)$$

$$(\forall op : \text{Operation}, a) \text{compoundActivity}(a) \wedge \text{perform}(a, op) \Leftrightarrow (\exists a_i : \text{Activity}) \text{subActivityOf}(a_i, a) \wedge \text{perform}(a_i, op) \quad (11.b)$$

From the previous expressions, it can be deduced that an activity a operates on various design objects, denoted by *do*. This relationship is not represented in the class diagram shown in Fig. 1 but is inferred by means of the predicate *operateOn*(a, do) defined in expression (12).

$$(\forall a : \text{Activity}, do : \text{DesignObject}) \text{operateOn}(a, do) \Leftrightarrow (\exists op : \text{Operation}) \text{perform}(a, op) \wedge \text{operateOn}(op, do) \quad (12)$$

As it is shown in Fig. 1, it is possible to identify different types of activities during a design process. This is so in the process system engineering domain, where some contributions are focused on activities that operate on design objects (Eggersmann et al., 2003) and propose three characteristic kinds of activities: *Synthesis*, *Analysis*, and *Decision*.

Whereas *Synthesis* activities create design objects, like process flow diagrams (PFD), reaction pathways, or mathematical models, *Analysis* activities are in charge of generating data about the design artefact itself, providing values for its distinctive attributes. Generally, an *Analysis* activity is used to predict the performance of a prospective design for its intended use from some lumped interpretation of detailed calculations. Then, based on the available information about the design artefact, a *Decision* activity may decide whether such artefact is adopted, rejected, or kept under consideration as a possible alternative to be further explored.

Though in certain cases it is possible to distinguish between the main types of activities that participate in a design process, in others some sort of aggregation may appear. Taking into account the classification proposed by Eggersmann et al. (2003), synthesis and analysis can

be lumped and carried out all at once, thus generating different alternatives along with all the necessary information to decide on the most convenient one. Therefore, a decision activity can be performed afterwards. In some other cases, the three types of activities could be aggregated. This occurs when a computer-aided tool like a mathematical programming based software implicitly generates, evaluates, and decides among different alternatives, as in the approach proposed by Floudas (1995) to synthesise heat exchanger networks.

Consequently, CoMoDe splits up the *activity type* concept from the *activity* one. This is done in order to propose a flexible model that could represent a particular activity classification adopted during a given design project. Furthermore, the model presented in Fig. 1 allows representing the aggregation of activity types by adopting a multiple cardinality (1...*) in the activity type role of the *activity-Type* relationship. An activity type has associated some characteristic operations, which are the ones that can be executed when the activity is carried out.

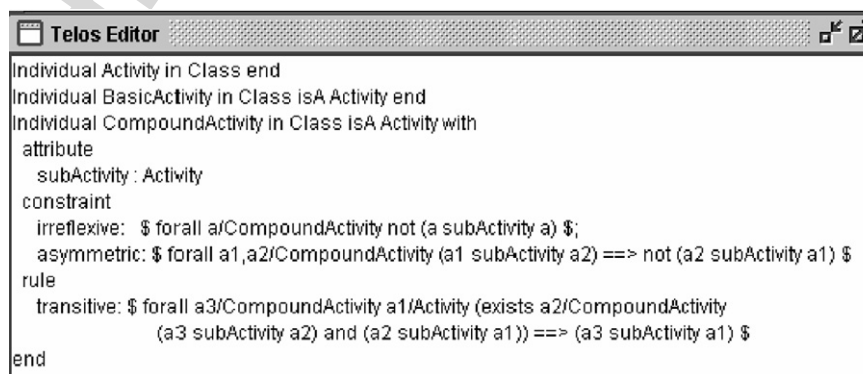
3.1.1. Retrieving knowledge about the process representation space

This section presents a case study focused on the process representation space. This case study is a well-known example in the process system engineering community. It was developed within the CRC 476 IMPROVE project (Nagl & Marquardt, 1997) and used with different purposes by several authors (Eggersmann, 2004; Eggersmann et al., 1999; Heller, Jäger, Schlüter, Schneider, & Westfechtel, 2004; Nagl, Westfechtel, & Schneider, 2003). It involves a design of a chemical plant to produce 40000 tons of polyamide-6 per year. In this work, its execution is represented by the proposed model using the specification provided by the O-Telos language and their implementation on ConceptBase. As introduced in Section 2, the O-Telos language was employed to formally specify the proposed model. This specification integrates the object-oriented model and the associated axioms. Furthermore, it makes it possible to obtain a deductive object base, implemented in ConceptBase, which enables to check and validate the proposed model.

The specification is obtained by means of the following rules:

- From the object-oriented model, a static representation is specified. This knowledge is part of the extensional database. Fig. 2 shows a partial specification of the *activity* concept.
- This specification is completed with the knowledge represented by the axioms, forming the so-called intentional database. Axioms that allow deducing new facts, such as the one represented by expression (5), have been defined as *rules* (see *transitive* rule in Fig. 2); and the ones that introduce constraints on the model, such as expressions (6) and (7), have been realized as *constraints* (see *irreflexive* and *asymmetric* constraints in Fig. 2).

The recursive decomposition of *subactivities* leads to an overall activity structure, as exemplified in Fig. 3. This figure represents the activities performed in the design of the polyamide-6 plant at several abstraction levels. At the most abstract one, the plant design was carried out by the *Design-Polyamide6Plant* activity. Obviously, it is a compound activity and it was performed by the following subactivities: *PrepareRequirements*, *DesignReaction*, *DesignSeparation*, *DesignCompounding*, and *DecidePlantDesign*. Initially, *DesignPolyamide6Plant* carried out the *PrepareRequirements* compound activity, where the problem to be solved was defined. This activity was carried out by a sequence of activities that identified the need for manufacturing 40000 tons per year of the Polyamide-6 product. Thus, the requirements *PlantCapacity* and *Polyamide-6ProductToManuf* were created during the execution of the *Define-Requirements* activity. After that, a literature study was carried out (*LiteratureResearch* activity), where the *Polyamide-6ProductToManuf* requirement was modified with the new gathered information. Furthermore, the polyamide reaction system was studied. In order to select between the anionic and the hydrolytic reaction mechanisms, a more detailed description about their advantages and disadvantages was needed. In this case, the two alternatives to be considered were the two reaction mechanisms. The decision technique was based on the polyamide-6 destination and



```

Telos Editor
Individual Activity in Class end
Individual BasicActivity in Class isA Activity end
Individual CompoundActivity in Class isA Activity with
attribute
  subActivity : Activity
constraint
  irreflexive: $ forall a/CompoundActivity not (a subActivity a) $;
  asymmetric: $ forall a1,a2/CompoundActivity (a1 subActivity a2) ==> not (a2 subActivity a1) $
rule
  transitive: $ forall a3/CompoundActivity a1/Activity (exists a2/CompoundActivity
    (a3 subActivity a2) and (a2 subActivity a1)) ==> (a3 subActivity a1) $
end

```

Fig. 2. Activity specification using O-Telos.

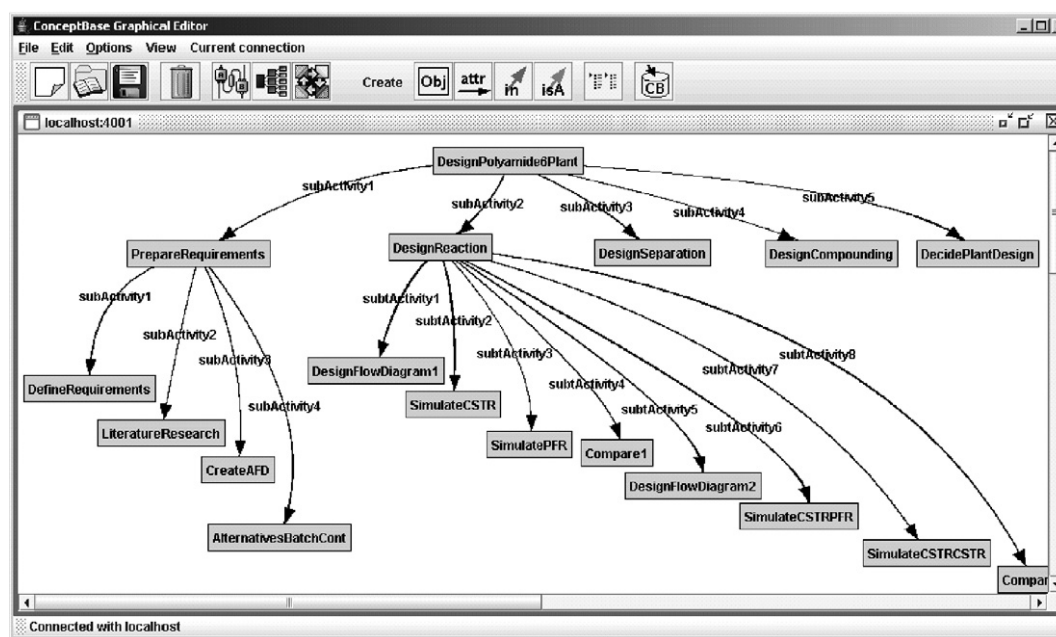


Fig. 3. DesignPolyamide6Plant activity decomposition.

plant size criteria. Then, the initial abstract flow diagram (AFD) was generated. The AFD specifies the overall function of the chemical process and decomposes it into subfunctions and connections between the subfunctions (Jarke, List, & Weidenhaupt, 1999). The next decision to adopt was the plant operation mode. The relevant information for selecting the operation mode was gathered by resorting to evaluate the *Batch-Operation-Mode-under-Hydrolytic* and evaluate the *Continuous-Operation-Mode-under-Hydrolytic* activities, subactivities of the *AlternativesBatchCont* activity, for the sake of simplicity of the figure these subactivities are not shown.

After the start of the project, different alternatives were evaluated based on the AFD. Reaction (*DesignReaction* activity), separation (*DesignSeparation* activity) and plastic processing systems (*DesignCompounding* activity) were developed concurrently. After the completion of these activities, a plant concept was established (*DecidePlantDesign* activity).

Fig. 3 only shows the decomposition of the *DesignReaction* activity. Since different alternatives exist for both the separation and the reaction phase, the *DesignFlowDiagram1* activity revealed that two types of reactors could be used: (i) a continuous stirred tank reactor (CSTR) and (ii) a plug flow reactor (PFR). Therefore, two alternatives were proposed: (i) realization of the reaction by a CSTR; and the (ii) realization of the reaction by a PFR. Consequently, the simulation of the CSTR (*SimulateCSTR* activity) and then the simulation of the PFR (*SimulatePFR* activity) were performed. The aim was to find the best operating conditions for the reactor. After these activities had been completed and the CSTR and PFR alternatives elaborated, both alternatives were compared (*Compare1* activity). Since none of them performed satisfactorily,

new reaction alternatives had to be considered: (i) realization of the reaction by a CSTR \rightarrow CSTR pair; and (ii) realization of the reaction by a CSTR \rightarrow PFR tandem (*DesignFlowDiagram2* activity). Therefore, new simulation activities were carried out (*SimulateCSTRPFR* and *SimulateCSTRCSTR* activities) and the most suitable alternative was selected.

The activity structure bottoms out in activities that are not further decomposed and are, therefore, characterized as basic activities. As it was defined in the process representation space, basic activities are materialised by means of a sequence of operations that are performed on *design objects*; for example, operations like *propose*, that introduces a *position* with the aim to *answer* a given *requirement*. In this case, the operation that posed the first reaction alternative (*PositionReactionByCSTR*) was performed while a subactivity of *DesignFlowDiagram1* was carried out. The various operations that were executed while addressing the case study are described in the Operation Context Section, in which the case study is discussed at the operation context granularity level.

3.2. Requirement representation space

As it is depicted in the class diagram of Fig. 1, the execution of an activity is guided by one or more *requirements*. This is represented by the *guidedBy* relationship. Therefore, the design process can be interpreted as a series of *activities* guided by requirements. If the *guidedBy(a,r)* predicate is defined (activity *a* is guided by requirement *r*), then it is possible to infer the potential requirements that could have guided the subactivities of *a*. This fact is represented in expression (13).

$$\begin{aligned}
& (\forall a_i:Activity, r:Requirement)(\exists a_c:CompoundActivity, \\
& r_i:Requirement) \text{ guidedBy}(a_c, r) \wedge \text{subActivity}(a_i, a_c) \\
& \wedge [r = r_i \vee \text{subRequirement}(r_i, r)] \\
& \Rightarrow \text{potentialGuide}(r_i, a_i)
\end{aligned} \quad (13)$$

The representation of requirements shown in Fig. 1 is very general and would need to be specialised and extended to model any particular domain. As it is specified in expression (13), a requirement may be decomposed into a set of subrequirements, which specify in a more concrete form the functional and non-functional properties that the designed product should meet. The adopted requirements' representation is similar to the goal representation employed in the actor model proposed by Eggersmann, Henning, Krobb, Leone, and Marquardt (2001). Therefore, a requirement can have zero or more possible structures (*ReqStructure*), which are related to one another by an implicit OR relationship, meaning that these structures are alternatives (Fig. 4). The class *ReqStructure* defines the relationship between a requirement and its subrequirements. It does have a type, which may be either AND or OR, or in turn, it can assume other types that could be specified by the user, allowing an extension of the concept of *ReqStructure*. These categories are defined as subclasses of the *StructureType* class. As shown in Fig. 4, the *AND* type determines that all subrequirements have to be met; the *XOR* (Exclusive OR) type states that exactly one of the subrequirements has to be satisfied to fulfil the superrequirement; and the *OR* (Inclusive OR) type indicates that at least one subrequirement has to be satisfied to meet the superrequirement. The satisfaction of the subrequirements does not guarantee the fulfilment of their superrequirements. For enhanced flexibility and modularity in the construction of complex requirements, the model allows a requirement structure (*ReqStructure*) to be composed of requirements (by means of the *consistOf* relationship) and/or structures (by means of the *subStructure* relationship). Moreover, the model allows a requirement to represent the functional and non-functional aspects that a design product must meet. It can be seen that the requirement concept is modelled as an aggregation of (possibly empty) conditions about products, referred as *ProductReq*, the ones that allow specifying an *Artefact*, and textual descriptions called fuzzy requirements (*FuzzyReq*).

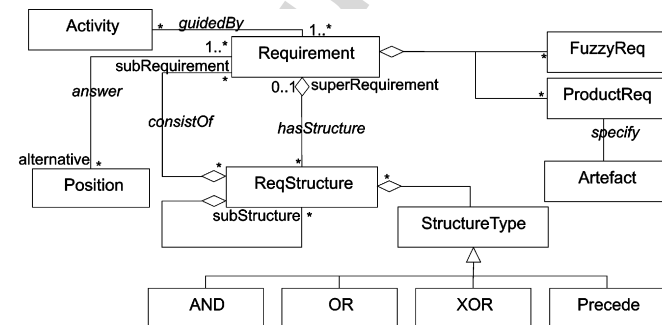


Fig. 4. Requirement model.

For this requirement representation, the requirement decomposition is not straightforward; then, it is necessary to know the subrequirements r_i of r that define the predicate $\text{subRequirement}(r_i, r)$. Expression (14) defines the axiom that makes it possible to know the subrequirements by resorting to the *hasStructure* and *structureConsistOf* predicates. *hasStructure*(r, st) means that requirement r has an associated structure st , and the *structureConsistOf*(st, r) is true if the requirement r is part of the structure st or one of its substructures, as defined in expression (15).

$$\begin{aligned}
& (\forall r_i, r_j:Requirement) \text{subRequirement}(r_i, r_j) \\
& \iff (\exists st:ReqStructure) \text{hasStructure}(r_j, st) \\
& \wedge \text{structureConsistOf}(st, r_i)
\end{aligned} \quad (14)$$

$$\begin{aligned}
& (\forall r_i:Requirement, st_i:ReqStructure) \text{structureConsistOf}(st_i, r_i) \\
& \iff \text{consistOf}(st_i, r_i) \\
& \vee [(\exists st_j:RequirementStructure) \text{subStructure}(st_j, st_i) \\
& \wedge \text{structureConsistOf}(st_j, r_i)]
\end{aligned} \quad (15)$$

Often, requirements may not be stated explicitly or with enough details at the beginning of the design process (Brown & Chandrasekaran, 1989). In general, they are refined and specified more precisely as a greater comprehension of the design problem is reached (Boyle, 1989; Cameron, Fraga, & Bogle, 2005; Goel, 1994). Then, it is very important to represent how requirements evolve during a project execution. This is described in Section 4, in which a requirement is represented as a *design object* having a life cycle.

3.2.1. Retrieving knowledge about the requirement representation space

As shown in the class diagram depicted in Fig. 1, the execution of an activity is guided by one or more requirements, fact that is represented by the *guidedBy* relationship. Indeed, the proposed model allows abstracting the requirement concept, focussing on: (i) the relationships among requirements and the design activities guided by them and (ii) the different design alternatives that arise as answers to requirements. This knowledge can be retrieved from the model implemented in ConceptBase by querying, navigating the knowledge base by *guidedBy* relationship, and its inverse, *guide*. Fig. 5 illustrates these relationships and other links as *PotentialGuide*, which enables to know the requirements that could have guided an activity (expression (13)). Furthermore, Fig. 5 illustrates that an activity can operate on a requirement, because this is a *design object*.

As regards the requirement concept, Fig. 6 shows the main requirement that has guided the case study. It is represented as an instance of the model introduced in Fig. 4. Therefore, *Requirement1* has associated a requirement structure (*ReqStruct1*) that specifies a set of requirements must be fulfilled (they are linked by an AND connector). Some subrequirements specify conditions on the product

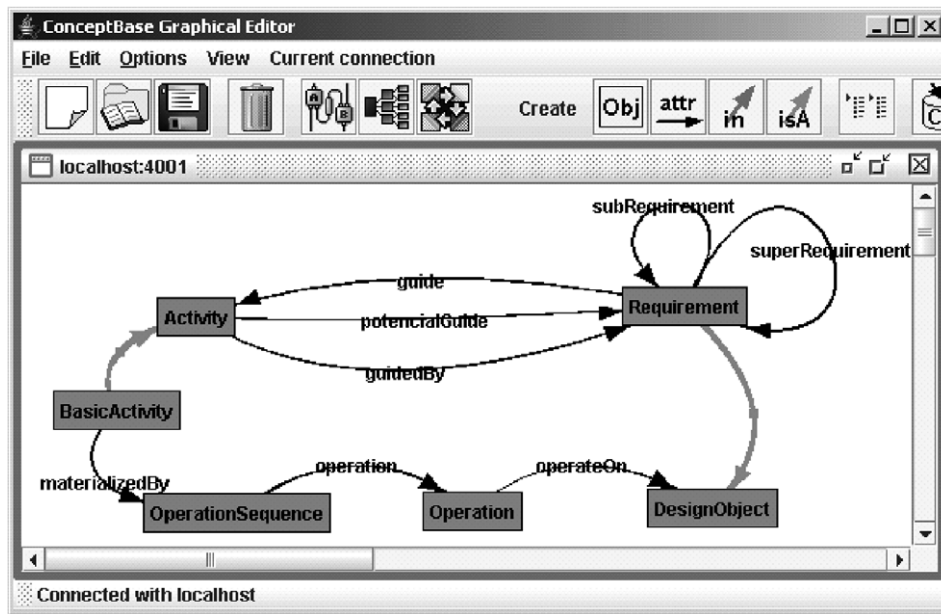


Fig. 5. A view of activity and requirement relationships.

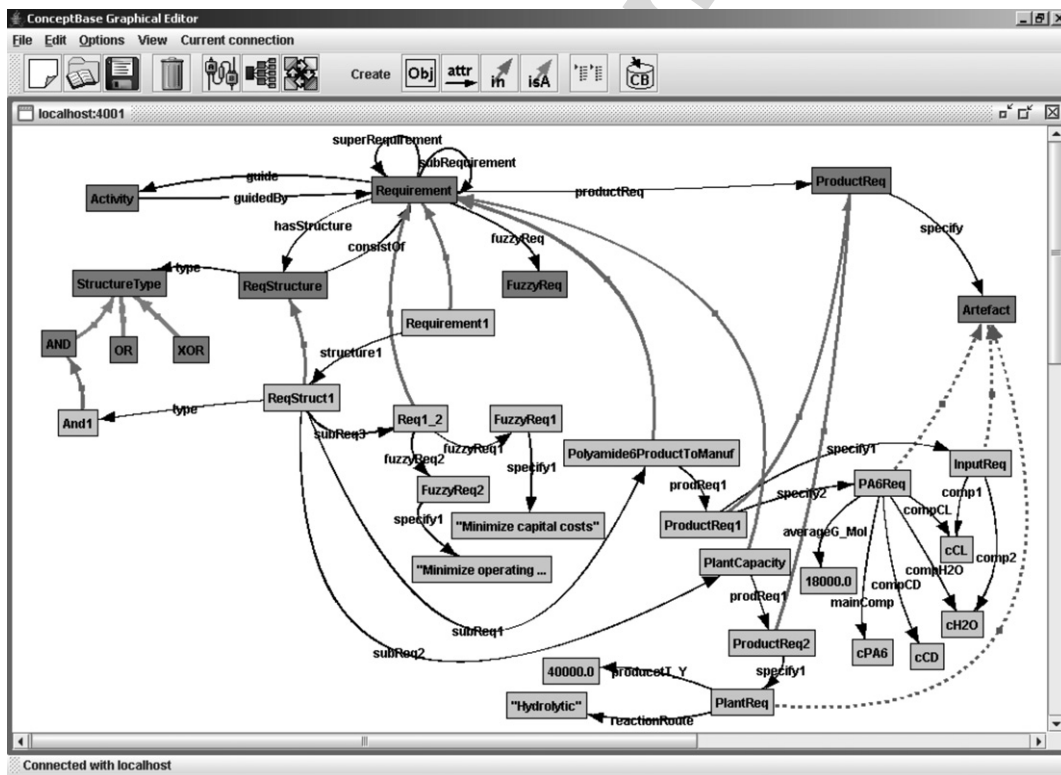


Fig. 6. A partial view of requirement model and its instances.

being designed; for that reason, they are linked with a *ProductReq* instance. For example, the *PlantCapacity* and *Polyamide-6ProductToManuf* requirements that were introduced by *DefineRequirements* activity. *Polyamide-6ProductToManuf* specifies some properties on the input (*InputReq*) and the output (*PA6Req*) of the plant, as the ϵ -caprolactam (*cCL*) residue must be less than 0.1%. Fig. 6 introduces non-functional requirements, as *FuzzyReq*

Req1 and *FuzzyReq2*, requirements related to the minimisation of capital and operating costs.

An observation worth mentioning is related to the various requirements that were posed while the case study was carried out and when those requirements appeared. Indeed, requirements might be created during every type of activity. This can be seen from an examination of the case study. Though the main requirement was to manufacture

40 000 tons per year of the Polyamide-6 product, fact that was represented by the *PlantCapacity* instance, the artefact to be designed also had to comply with other objectives that appeared during the design process, such as the one requiring the product to be manufactured via the hydrolytic route (*PlantReq*). This requirement arose as a result of the *LiteratureResearch* activity where an analysis of two reaction alternatives was carried out.

3.3. Actor representation space

As introduced in the process representation space, *activities* are executed either by designers or by automated tools, thus, actors. The *actor representation space* presented in this paper extends the actor model introduced by Eggersmann et al. (2001). This extension is made with the aim of proposing an integrated model and answering questions such as “who performed a given activity?”, “which requirements did the actor try to meet?”.

The main concepts are shown in Fig. 1 where it is possible to observe that each *activity* is related to the *actor* who executes it (*execute* relationship in Fig. 1). An *actor* may be either an *individual* (expression (16)) – a human being or a computational program, or a *team* (expression (17)). The team concept allows modelling a set of actors.

$$(\forall ac_i) \text{ individual}(ac_i) \Rightarrow \text{actor}(ac_i) \quad (16)$$

$$(\forall ac_i) \text{ team}(ac_i) \Rightarrow \text{actor}(ac_i) \quad (17)$$

Thus, a *team* may be decomposed into a set of *actors*. The relationship between a team and its members is captured by an aggregation link (Fig. 1). This relationship is represented in first order logic by the *member*(*ac*, *ac_i*) predicate, which means that *ac* is a member of the *ac_i* team. As the *subActivity* relationship, *member* is transitive (expression (18)), irreflexive (expression (19)), and antisymmetric (expression (20)).

$$(\forall ac_1 : \text{Actor}, ac_2, ac_3 : \text{Team}) \text{ member}(ac_1, ac_2) \wedge \text{member}(ac_2, ac_3) \Rightarrow \text{member}(ac_1, ac_3) \quad (18)$$

$$(\forall ac : \text{Team}) \neg \text{member}(ac, ac) \quad (19)$$

$$(\forall ac_1, ac_2 : \text{Team}) \text{ member}(ac_1, ac_2) \Rightarrow \neg \text{member}(ac_2, ac_1) \quad (20)$$

It is important to remark that an *actor* can be *member* of zero, one, or more *teams*, as it is represented in Fig. 1, by the multiplicity * (zero or more) in the *member* relationship. However, an individual cannot be considered as a team (expression (21)), thus, it cannot have members (expression (22)).

$$(\forall ac : \text{Actor}) \text{ team}(ac) \iff \neg \text{individual}(ac) \quad (21)$$

$$(\forall ac_i) \text{ individual}(ac_i) \iff \neg (\exists ac : \text{Actor}) \text{ member}(a, ac_i) \quad (22)$$

Each *individual* has defined a set of *skills* (expression (23)). This relationship is represented by the link between

the *Individual* and *Skill* classes in Fig. 1, and it represents the knowledge of a particular actor. The *Skill* class, which encapsulates information about the abilities of actors, is quite general. It can be specialised and extended to represent the special characteristics of a particular domain. For example, it may be enriched to represent the competency concept proposed by Harzallah and Vernadat (2002).

$$(\forall ac) \text{ individual}(ac) \Rightarrow (\exists s : \text{Skill}) \text{ hasSkill}(ac, s) \quad (23)$$

The team concept allows representing the compound *skills* that are needed for performing many design activities. They are not organizational units because the present work is focused on process support/process tracing. Expression (24) enables the skill inference of a given *team*.

$$(\forall ac_i : \text{Team}, s : \text{Skill}) (\exists ac : \text{Actor}) \text{ member}(ac, ac_i) \wedge \text{hasSkill}(ac, s) \iff \text{hasSkill}(ac_i, s) \quad (24)$$

Moreover, the model shown in Fig. 1 represents the fact that activities are executed by those actors having the necessary *skills* to carry them out. As it has been shown, the *actor*, *activity*, and *skill* classes are connected to one another in order to represent both the actors' skills and the skills needed to carry out the various activities. Expression (25) prevents the execution of one activity by one actor without the needed skills. The predicate *performedBy*(*a*, *ac*) means that the *activity* *a* was performed by the *actor* *ac* (*execute* relationship in Fig. 1) and *need*(*a*, *s*) represents the *skill* needed to perform the *activity* *a* (relationships between *Activity* and *Skill* class in Fig. 1).

$$(\forall ac : \text{Actor}, s : \text{Skill}, a : \text{Activity}) \text{ need}(a, s) \wedge \text{performedBy}(a, ac) \Rightarrow \text{hasSkill}(ac, s) \quad (25)$$

As previously specified, the *activity* concept represents *basic* and *compound activities*. Therefore, if an actor performs a compound activity, then the subactivities would be carried out by the actor itself or by a member of the team represented by the actor (expression (26)).

$$(\forall ac_k : \text{Actor}, a_k : \text{Activity}) (\exists ac : \text{CompoundActivity}) \text{ subActivity}(a_k, ac) \wedge (\text{performedBy}(ac, ac_k) \vee (\exists ac_i : \text{Team}) \text{ performedBy}(ac, ac_i) \wedge \text{member}(ac_k, ac_i)) \Rightarrow \text{performedBy}(a_k, ac_k) \quad (26)$$

Each *actor* has goals (*actor's goals*) that express the actor's intentions and desires. The actor's decision of executing a given activity for reaching one or more goals with the final aim of satisfying a set of requirements is represented by the *promote* links among the *activity*, *actor's goal*, and *requirement* entities (Fig. 1). These links reflect the actor's intention.

3.3.1. Retrieving knowledge about the actor representation space

The model shown in Fig. 1 allows the representation of the actor (individual or team) that has executed a given activity. This knowledge is explicitly modelled by the

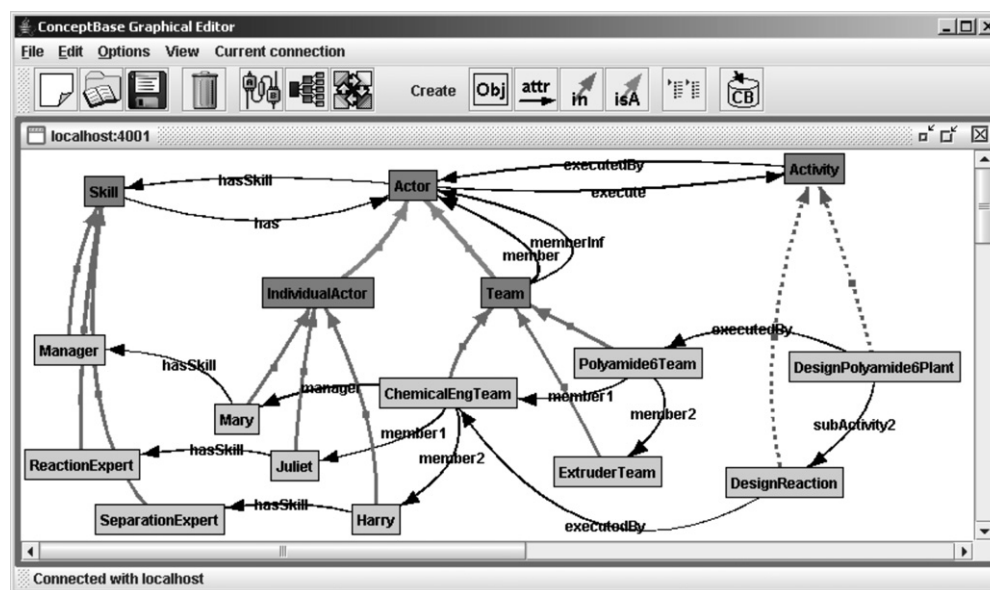


Fig. 7. Some actors involved in the execution of the case study.

execution relationship between the *Activity* and *Actor* classes. As seen in the case study, the actor performs a given activity with the aim of meeting a given goal (stated by means of one or more requirements). This knowledge is structured in the model shown in Fig. 1 by using: (i) the *execute* relationship, (ii) the *Actor'sGoal* class, which captures the goals of the actor; and (iii) the *promote* relationships that link both the requirement to be met with the activity performed to attain it and the activity with the sought goal. For example, Fig. 7 shows that the *DesignReaction* activity was carried out by the *ChemicalEngTeam* team and the members of this team were *Mary*, *Juliet* and *Harry*. Each actor is shown with his/her skills.

3.4. Artefact representation space

The main product of the design process is the artefact being designed. With the aim of proposing a domain independent model, the representation of the design product is encapsulated in the *Artefact* class, a subclass of *DesignObject* (see Fig. 1). Thus, the same modelling criterion used in the definition of the requirement concept is once again adopted. The *Artefact* class encapsulates relationships with the main concepts included in CoMoDe. Thus, as any *Artefact* is a design object, the operations that materialise basic activities may operate on it.

On the other hand, the *Artefact* class can be specialised and/or extended to represent the design products that pertain to a particular domain. For example: (i) in the software engineering domain it is possible to specialise the *Artefact* class with the UML concepts (Booch, Rumbaugh, & Jacobson, 2005), or with the software architecture domain concepts (Bass, Clements, & Kazman, 2003; Rolán, Gonnet, & Leone, 2005); (ii) in enterprise engineering, it is feasible to specialise the artefact concept with the

building blocks of enterprise modelling languages as Coordinates (Mannarino, 2001), TOVE (Fox & Gruninger, 1998), or CIMOSA (computer integrated manufacturing-open systems architecture) (Vernadat, 1996); (iii) and in the process systems engineering domain, the *Artefact* class can be enriched with the concepts of a chemical process modelling language, as MODEL.LA (Stephanopoulos, Henning, & Leone, 1990).

3.4.1. Retrieving knowledge about the artefact representation space

As it was previously indicated, the *Artefact* class encapsulates the main design product in a generic, domain independent way. Therefore, such concept needs to be specialised. This can be done by resorting to any language, such as CLiP (Bayer & Marquardt, 2004), suited to represent models employed at a conceptual design level. In this case study, the MODEL.LA (Stephanopoulos et al., 1990) language was employed. Therefore, the *Artefact* representation space was enriched with the MODEL.LA building blocks. Fig. 8 shows a partial view of this particular extension of the artefact space. The model incorporates the concept of generic unit (*MLAGenericUnit*), which is a subclass of the *Artefact* class, and other classes are used to complete the definition, such as *MLAPort* class, representing ports. Additionally, Fig. 8 shows the *Polyamide6Process* instance and its input and output flows (*MLAInputStream* and *MLAOutputStream*, respectively), generated by the *CreateAFD* activity.

3.5. Decision representation space

With the aim of representing the rationale associated with the execution of a given activity, the IBIS model (Kunz & Rittel, 1970) has been adopted and extended.

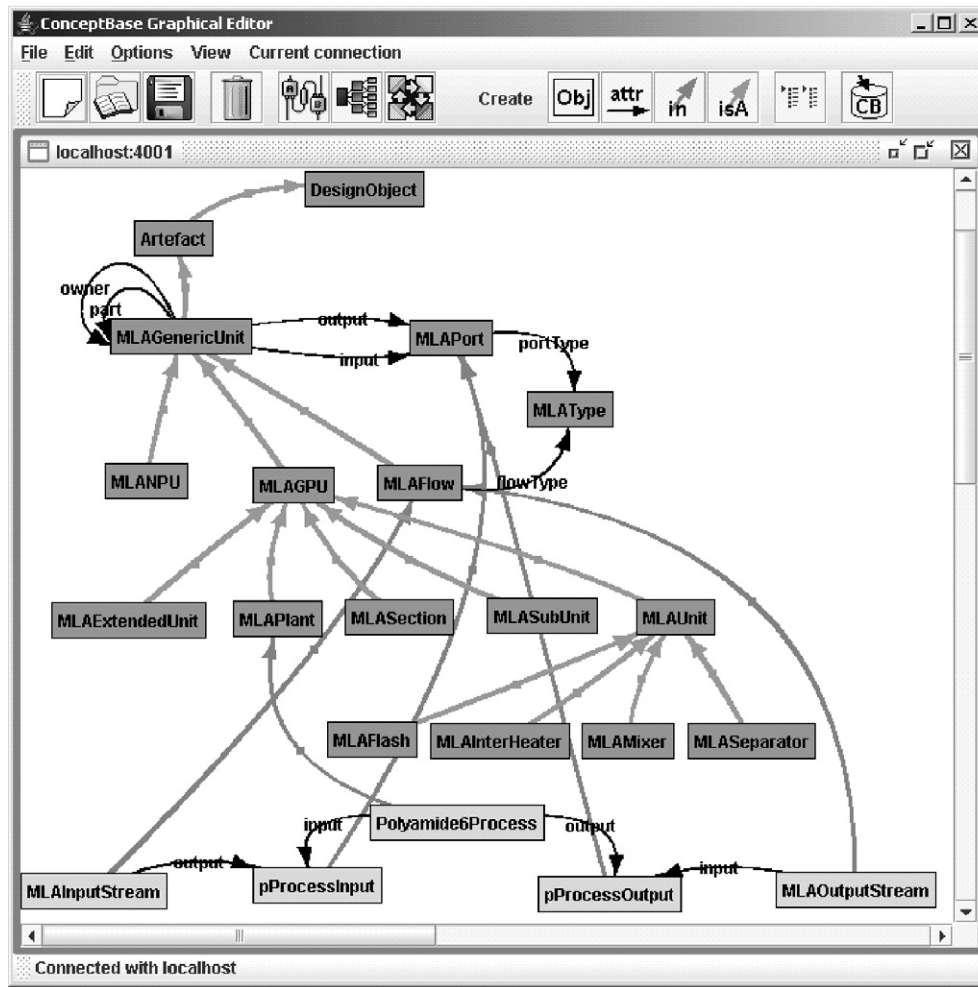


Fig. 8. A partial view of the MODEL.LA based extension of the artefact representation space.

The IBIS model focuses on articulating key design issues. An issue is a question to be answered and a position is an alternative that exists for solving such issue. CoMoDe extends this idea by (i) introducing *requirements*, which specify issues, (ii) decomposing *positions* into *artefacts*, *attributes* (*PosAttribute*), and *values* (*PosValue*), and also by (iii) adding *resolutions*, which represent the selection (or refusal) of an alternative with the aim of resolving a *requirement* (see Fig. 1).

Therefore, a *position* encompasses: (i) a design *artefact*, such as a software architecture, a flowsheet structure, a mathematical model describing part of the chemical process being considered, the geometry of a piece of equipment, etc., its (ii) *attributes* and (iii) corresponding *values*. As it was indicated, an *artefact* represents the product that it is being designed, whereas the *attributes* and *values* characterise the *position*. Then, the different alternative products that arise in the design process are represented by the *position* concept. A *position* is qualified by one or more *arguments* and addresses at least one *requirement* (expression (27)). An *argument* either *supports* or *objects* a *position*. It allows testing whether the position is capable of fulfilling

the prescribed requirements by means of the *answer* relationship.

$$(\forall p) \text{ position}(p) \iff (\exists r: \text{Requirement}) \text{ answer}(p, r) \quad (27)$$

A *position* p is accepted to answer a *requirement* r (predicate $\text{accepted}(p, r)$) by a *resolution* res , if res tries to *resolve* the *requirement* r and *accepts* p as a valid answer (expression (28)). Expression (29) presents the refusal of a position by a resolution.

$$(\forall p: \text{Position}, r: \text{Requirement}) \text{ accepted}(p, r) \iff (\exists res: \text{Resolution}) \text{ accept}(res, p) \wedge \text{resolve}(res, r) \quad (28)$$

$$(\forall p: \text{Position}, r: \text{Requirement}) \text{ refused}(p, r) \iff (\exists res: \text{Resolution}) \text{ refuse}(res, p) \wedge \text{resolve}(res, r) \quad (29)$$

Positions, *artefacts*, *attributes*, *values*, *resolutions*, and *arguments* evolve during the execution of a design project and their various states are fundamental for representing the different contexts where activities are performed. Then, they are modelled as *design objects* (see Fig. 1), and the representation of their evolution is described in the Operation Context section.

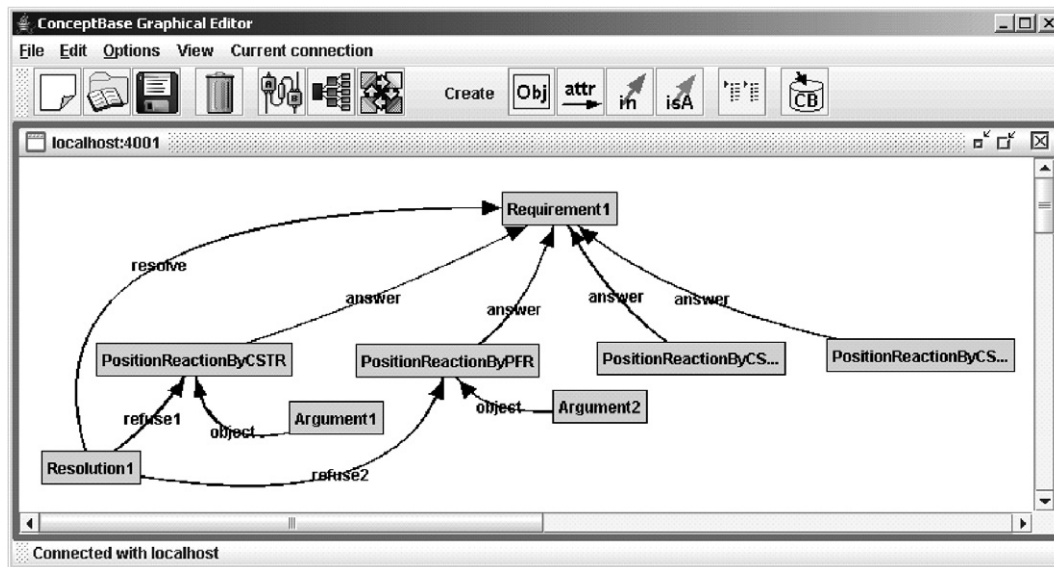


Fig. 9. Positions answering Requirement1.

3.5.1. Retrieving knowledge about the decision representation space

While performing the case study, different positions have been proposed with the objective of satisfying the requirements that were specified. Since it is desirable to know why a position was refused or selected, it is necessary to document the actor's decisions and to link the various design alternatives with the arguments that support or object them. Fig. 9 presents a partial view of how the information associated to the underlying rationale that was captured during the case study has been structured. It shows that the four positions that were proposed provide an answer to the *Requirement1* requirement. It also shows that the first two proposed positions (*PositionReactionByCSTR* and *PositionReactionByPFR* representing the reaction by CSTR and PFR units respectively) were considered inadequate (there is a refuse link between *Resolution1* and *PositionReactionByCSTR* and another refuse link between *Resolution1* and *PositionReactionByPFR*) as a result of *Compare1* activity. The decision was based in the *Argument1* and *Argument2* arguments. This latter fact is represented by the *object* links between *PositionReactionByCSTR* and *Argument1*, and *PositionReactionByPFR* and *Argument2*.

As seen, the IBIS extension incorporated in CoMoDe allows for representing in an integrated form: (i) the design alternatives that arose as answers to the imposed requirements (for simplicity reasons, Fig. 9 just shows only one requirement), (ii) the arguments that support or object the different positions, and (iii) the resolutions that were adopted in each case.

4. Operation context

Each *basic activity* performed during a design process is represented, at the lower level of granularity, in the *Operation Context* through the execution of a sequence of oper-

ations, which transforms the products of the design process (*design objects*). While a design process is carried out, *design objects* evolve into multiple *versions*. Consequently, this granularity level focuses on representing (i) the various states (versions) of the design objects along their life cycle, and (ii) how these states are derived.

As it was previously introduced, *activities* operate on the outcomes or products of the design process, called *design objects* (Fig. 1). A *design object* represents any entity that can evolve during a design project. It is represented at two levels, the *repository* and the *versions level* (Fig. 10). The *repository* level keeps a unique entity for each *design object* that has been created and/or modified due to model evolution during a design project. This object is regarded as a *versionable object* (*o*). Furthermore, relationships among the different *versionable objects* are maintained in the repository. These relationships correspond, according to the notation being used, to the rules that allow associating objects in order to develop syntactically valid models.

On the other hand, the *versions level* keeps the different versions of each *design object*. These are called *object versions* (*v*). The relationship between a *versionable object* and one of its *object versions* is represented by the *version* association and the *version(v,o)* predicate. Therefore, a given *design object* keeps a unique instance in the *repository*, and all the versions it assumes along the design process in different model versions belong to the *versions level*.

At a given stage during the execution of a design project, the states assumed by the set of relevant *design objects*, from now on called *model version*, supply a snapshot description of the state of the design process at that point. According to the proposed representation, a new model version m_n is generated when an activity *a* (a *basic activity*) is executed. A basic activity *a* is materialised by a sequence of operations ϕ and the new model version m_n is the result of applying such sequence ϕ to the components of

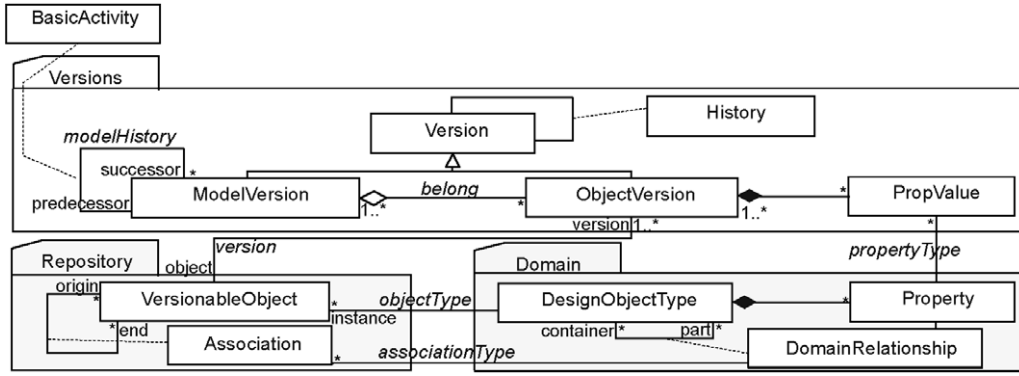


Fig. 10. Version administration model (operation context).

a previous model version m_p . The *predecessor model version* m_p constitutes the *context* where the activity a was performed and the *successor model version* m_n represents the *resulting context*. This model evolution is posed as a history made up of discrete situations. The situation calculus (Reiter, 2001; Scherl & Levesque, 2003) is adopted for modelling such version generation process. Therefore, the new model version m_n is achieved by performing the following evaluation: $apply(\phi, m_p) = m_n$.

The *apply* function is defined in expression (30), where *SequenceOfOperations* is the set of all possible operation sequences ϕ (defined in expression (10)) and *ModelVersion* is the set of possible model versions m .

$$apply: SequenceOfOperations \times ModelVersion \rightarrow ModelVersion \quad (30)$$

The primitive operations that were proposed to represent the transformation of *model versions* are *add*, *delete*, and *modify*. By using the *add(v)* operation, an *object version* that did not exist in a previous *model version* can be incorporated into a successor one. Conversely, the *delete(v)* operation eliminates an *object version* that existed in the previous *model version*. Also, if a *design object* has a version v_p , the *modify*(v_p, v_s) operation creates a new version v_s of the existing *design object*, where v_s is a successor version of v_p . Thus, an *object version* v is *added* after applying the sequence of operations ϕ to *model version* m when the new version v is created by means of an *add* or *modify* operation (expression (31)). On the other hand, the expression (32) represents the fact that an *object version* v is *deleted* after applying the sequence of operations ϕ to *model version* m when the version v is deleted by the *delete* or *modify* operation.

$$(\forall \phi: SequenceOfOperations, v: ObjectVersion, m: ModelVersion) \begin{aligned} &add(v) \in \phi \vee (\exists v_p: ObjectVersion) \\ &modify(v_p, v) \in \phi \Rightarrow added(v, apply(\phi, m)) \end{aligned} \quad (31)$$

$$(\forall \phi: SequenceOfOperations, v: ObjectVersion, m: ModelVersion) \begin{aligned} &delete(v) \in \phi \vee (\exists v_s: ObjectVersion) \\ &modify(v, v_s) \in \phi \Rightarrow deleted(v, apply(\phi, m)) \end{aligned} \quad (32)$$

From these definitions, and by using the format of the successor state axioms proposed by Reiter (2001), a formal specification of the cases in which an *object version* belongs to a *model version* is presented. In expression (33), the predicate *belong*(v, m) is true when the *object version* v belongs to the *model version* m . Thus, an *object version* v belongs to a *model version* that arises after applying sequence of operations ϕ to *model version* m , if and only if one of the following conditions is met:

- (i) v is added when the new *version* is created; or
- (ii) v already belonged to previous *model version* m (*belong*(v, m)) and it is not deleted when ϕ is applied to it.

$$(\forall \phi: SequenceOfOperations, v: ObjectVersion, m: ModelVersion) \begin{aligned} &belong(v, apply(\phi, m)) \\ &\iff belong(v, m) \vee added(v, apply(\phi, m)) \\ &\wedge (\neg deleted(v, apply(\phi, m))) \end{aligned} \quad (33)$$

From this expression, the *object versions* that belong to a certain *model version* can be determined. Then, it is possible to reconstruct a *model version* m_{i+1} by applying all the sequences of operations from the initial *model version* m_0 (expression (34)).

$$\begin{aligned} m_{i+1} &= apply(\phi_{i+1}, m_i); m_i = apply(\phi_i, m_{i-1}); \dots; \\ m_1 &= apply(\phi_1, m_0) \\ m_{i+1} &= apply(\phi_{i+1}, apply(\phi_i, apply(\dots apply(\phi_1, m_0) \dots))) \\ m_{i+1} &= apply(\phi_1 \bullet \dots \bullet \phi_i \bullet \phi_{i+1}, m_0) \end{aligned} \quad (34)$$

Once the versions belonging to a *model version* are defined, the relationships existing among *object versions* have to be specified. First, it should be noted that in this proposal, *object versions* belonging to a *model version* are not explicitly associated with other versions belonging to the same model version. These links are represented at the repository level (see Fig. 10). Consequently, the relationship existing between two object versions must be inferred from the relationship established between the objects that have been versioned by them. This fact is

represented in expression (35), in which an association a_k is inferred between two object versions, namely v_1 and v_2 , belonging to the same model version m (*inferredAssociation*(a_k, v_1, v_2, m)) if and only if there exists an association a_k between the two versionable objects o_1 and o_2 (*association*(a_k, o_1, o_2)), of which v_1 and v_2 are versions, respectively (*version*(v_1, o_1) and *version*(v_2, o_2)).

$$\begin{aligned}
 &(\forall v_1, v_2: \text{ObjectVersion}, m: \text{ModelVersion}, a_k: \text{Association}) \\
 &\quad \text{inferredAssociation}(a_k, v_1, v_2, m) \\
 &\quad \iff (\exists o_1, o_2: \text{VersionableObject}) \\
 &\quad \text{belong}(v_1, m) \wedge \text{belong}(v_2, m) \wedge \text{version}(v_1, o_1) \\
 &\quad \wedge \text{version}(v_2, o_2) \wedge \text{association}(a_k, o_1, o_2)
 \end{aligned} \tag{35}$$

The proposed scheme is strengthened by object-oriented modelling elements, which represent the relationships existing among *object versions* of different *model versions*, allowing the navigation along the history of the *object versions* that comprise a given *model version*. The relationships among *object versions* are represented by means of explicit links at the *versions' level*, named *history* (Fig. 10). Each transformation operation that is applied to a *model version* incorporates the necessary information in terms of the previous link to trace the model evolution. The incorporation of *history* relationships allows the definition of attributes oriented towards the characterization of the *history* of the executed operations; for instance, attributes aim to capture temporal information (when the operation was performed) and documentary information (tool employed, actor carrying out the operation, reasons for doing it).

4.1. Extension of the version administration model

This basic model can be specialised according to the particular domain being tackled. The specialisation can be done in terms of the different operations that are applied to the distinct *design objects*, in terms of the different *design objects* that participate in the design process, and in terms of the allowed *relationships* among design objects.

4.1.1. Extension of primitive operations

Due to the increasing complexity and to the dynamic changes occurring in the several engineering domains, it is not possible to represent “a priori” all the elements for all projects and situations in an information model (Bayer & Marquardt, 2004). The primitive operations *add*, *delete*, and *modify* are not sufficient to capture and trace how a design process has been carried out. Then, the previously introduced model must be specialised according to the particular domain being tackled. The particular operations must be expressed in terms of *added* and *deleted* predicates previously defined in expressions (31) and (32). For example, the *refine*(v, α) operation allows decomposing an object version v into one or more versions of design objects, where α is a set of object versions v_i .

$$\begin{aligned}
 &(\forall \phi: \text{SequenceOfOperations}, \alpha: \text{PObjectVersion}, v_i, \\
 &\quad v: \text{ObjectVersion}, m: \text{ModelVersion}) \\
 &\quad \text{refine}(v, \alpha) \in \phi \wedge v_i \in \alpha \\
 &\quad \Rightarrow \text{deleted}(v, \text{apply}(\phi, m)) \wedge \text{added}(v_i, \text{apply}(\phi, m))
 \end{aligned} \tag{36}$$

The definition of new operations, in a similar way to expression (36), allows enlarging the set of operations, incorporating domain specific ones. This can be done without modifying the successor state axiom (expression (33)).

A possible extension is related to the different operations that are carried out when performing *Synthesis*, *Analysis*, or *Decision* activities in the domain of process systems engineering. For example, in the proposal by Eggersmann et al. (2003) operations like *propose*, *add*, *delete*, *modify*, *merge*, *select*, and *request* were identified during the execution of the *Synthesis* activities. The *propose* operation conveys the action of putting forward something new as product data (*design object*). *Merge* creates something new by combining already existing elements in a consistent way, *select* chooses something from a given set of possibilities, and *request* solicits additional information, allowing a synthesis activity to be interactive. In turn, during an *Analysis* activity, operations like *calculate*, *estimate*, *determine*, *experiment*, and *request* can be recognised. In order to present their meaning, let us consider that after executing a *synthesis* activity, product data will exist, but some of their attribute values may be unspecified. These values are needed as a basis for subsequent *decisions*. One of the most common cases is to perform calculations (operation *calculate*) in order to provide the missing information. If calculations are not possible or cannot be afforded, values can be *estimated*. In other situations, attribute values may be obtained in the literature or by browsing databases (*determine*). The engineer may perform an *experiment* activity to test a proposed design and generate additional data. Finally, decision activities may include operations such as *choose*, *evaluate*, *justify*, and *request*. A *choose* operation refers to the selection of one or more design products among a number of possible alternatives. Before choosing among them, some information generated during an analysis activity is compared with the requirements to be fulfilled. In this way, an *evaluation* operation provides arguments to justify a decision. Similarly, *justify* offers a rationale for the selection of a certain alternative. The *request* operation solicits additional information, allowing a *decision* to be interactive. As already seen, requesting for new information can occur at every stage of the design process and is therefore not specific to a particular activity.

These operations are expressed in terms of *added* and *deleted* predicates defined previously in expressions (31) and (32), respectively. For example, the *evaluate*(v_p, α) operation generates a set of arguments that qualify a given position v_p , where α embraces the arguments v_a . The definition of the new operation in a similar way to expression (37) allows the primitive operations to be extended without modifying the successor state axiom presented in expression (33).

$$\begin{aligned}
& (\forall \phi: \text{SequenceOfOperations}, \alpha: \text{PObjectVersion}, \\
& \quad v_a, v_p: \text{ObjectVersion}, m: \text{ModelVersion}) \\
& \quad (\text{evaluate}(v_p, \alpha) \in \phi) \wedge v_a \in \alpha \Rightarrow \text{added}(v_a, \text{apply}(\phi, m))
\end{aligned}
\tag{37}$$

4.1.2. Extension of design objects

Another possible extension is related to the distinct *design objects* that participate in the design process, as well as to the allowed *relationships* among them. In order to make this extension possible in a flexible way, the *Domain* package is defined (see Fig. 10). This package is understood as a language that allows us to define the various design object types with their properties and relationships between them. Thus, an instance of *DesignObjectType* class is created for each design object type identified in the design process domain. The properties that can evolve while the design is being carried out are defined by instances of *Property* class. Finally, *DomainRelationship* class represents the allowed relationships between design objects.

Then, to capture and maintain the evolution of the design process, the *design object types* defined in the *context activity* (in *Requirement*, *Artefact*, and *Decision* representation spaces, see Fig. 1) have to be defined as instances of *Domain* package. Fig. 11 shows a partial view of *design object* and *domain relationship* instances.

4.1.3. Retrieving knowledge at the operation context level

This section presents a case study focused on the Operation Context level. The main axiom specified at this level is the successor state axiom defined in expression (33), where a formal specification of the cases in which an *object version* belongs to a *model version* is presented. Its specification using the O-Telos language is shown in Fig. 12. It is introduced as a rule of *model version*.

The next figure, Fig. 13, shows a partial view of the graphical representation of one of the steps of the design process generated by ConceptBase. It corresponds to the synthesis of the AFD. As it is illustrated in the figure, the *CreateAFD* activity has the *CreateInputOutputStructure* and *CreateRecycleStructure* basic activities as subactivities. As it is shown in Fig. 13, the basic activities are materialised in a sequence of operations that are performed on the design objects, allowing their evolution to a new version. The *ModelVersion_2* model version was the activity context of the *CreateInputOutputStructure* basic activity and that model version has been inferred from the object base by means of expression (33). This expression has been written in O-Telos as presented in Fig. 12. The instances that belong to the model version are versions of the design objects that are part of the requirement, artefact and decision representation spaces. When the *fiCreateInputOutputStructure* sequence of operations was applied to the *ModelVersion_2*, this last evolved to a new model version,

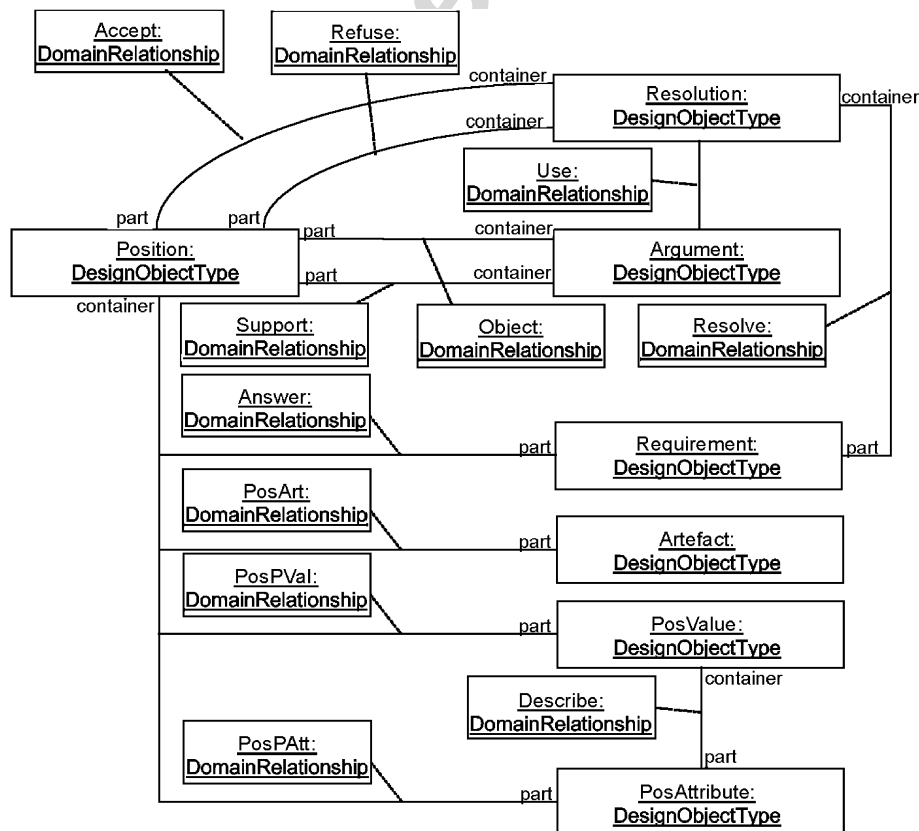


Fig. 11. Domain representation: a partial view of its extension.

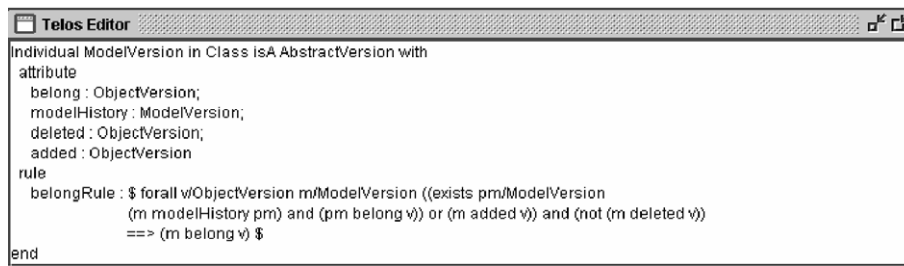


Fig. 12. Successor state axiom using O-Telos.

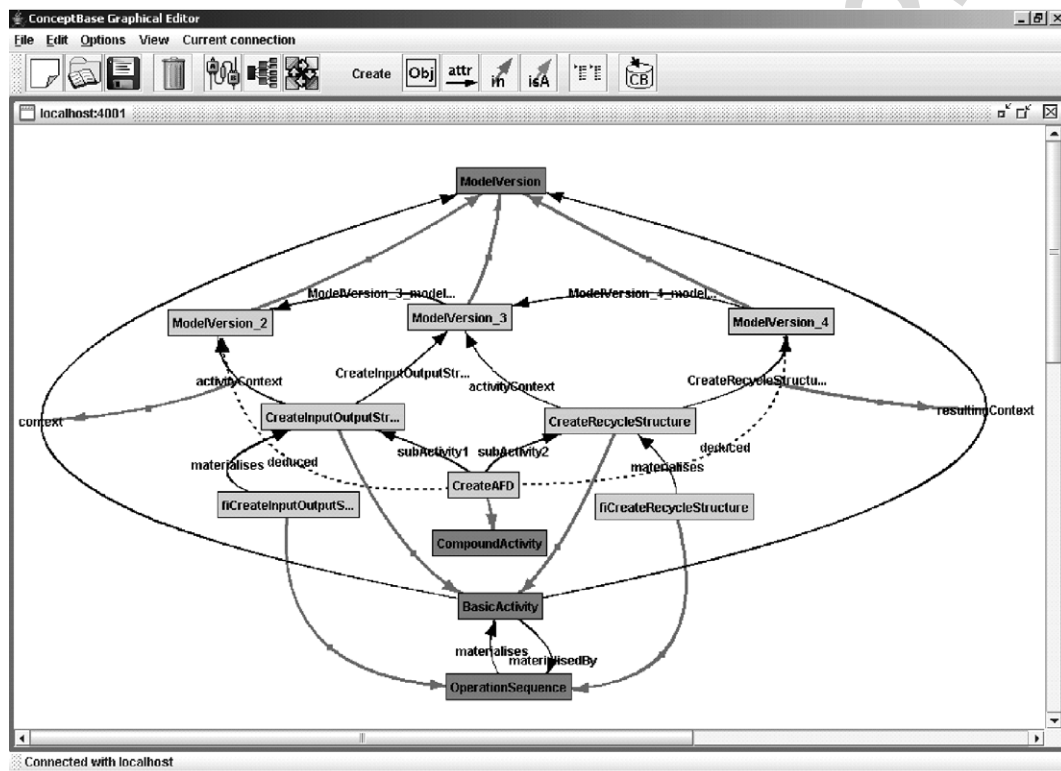


Fig. 13. Partial view of the CreateAFD activity that was captured in ConceptBase.

called *ModelVersion_3*, conforming a new context to new activities. Therefore, *ModelVersion_3* is the context of the *CreateRecycleStructure* basic activity. Furthermore, the contexts of the *CreateAFD* compound activity are inferred. They are labelled as *deduced* and represent that *ModelVersion_2* is the context of this compound activity, and *ModelVersion_4* is its resulting context.

Fig. 14 lists the object versions that belong to the model versions illustrated in Fig. 13. These are *ModelVersion_3* and *ModelVersion_4*. The *predecessor* attribute represents the previous model version. Furthermore, the figure illustrates a partial view of two object versions, *ObjectVersion_3* and *ObjectVersion_4*. It is possible to see that object versions are linked to *versionable objects*, as *VersionableObject_3* and *VersionableObject_4*, respectively. These versionable objects are related to the design object type they represent, as a flow (*MCMLAFlow*).

5. Supporting the conflict management process

On the basis of the knowledge captured by CoMoDe, this section provides specific procedures to handle different situations that may arise as the result of working in a collaborative environment.

In this proposal, it is considered that different design teams may perform independent concurrent activities on “a priori” independent or slightly coupled parts of the artefact being designed. As Kvan (2000) states, it is the most common type of concurrent work, with different actors contributing what they can in different domains of expertise at moments when they have the most suitable knowledge for a particular situation. Independent activities mean two or more actors operating in their particular model versions that were derived from a common model version without having an “a posteriori” conflict.

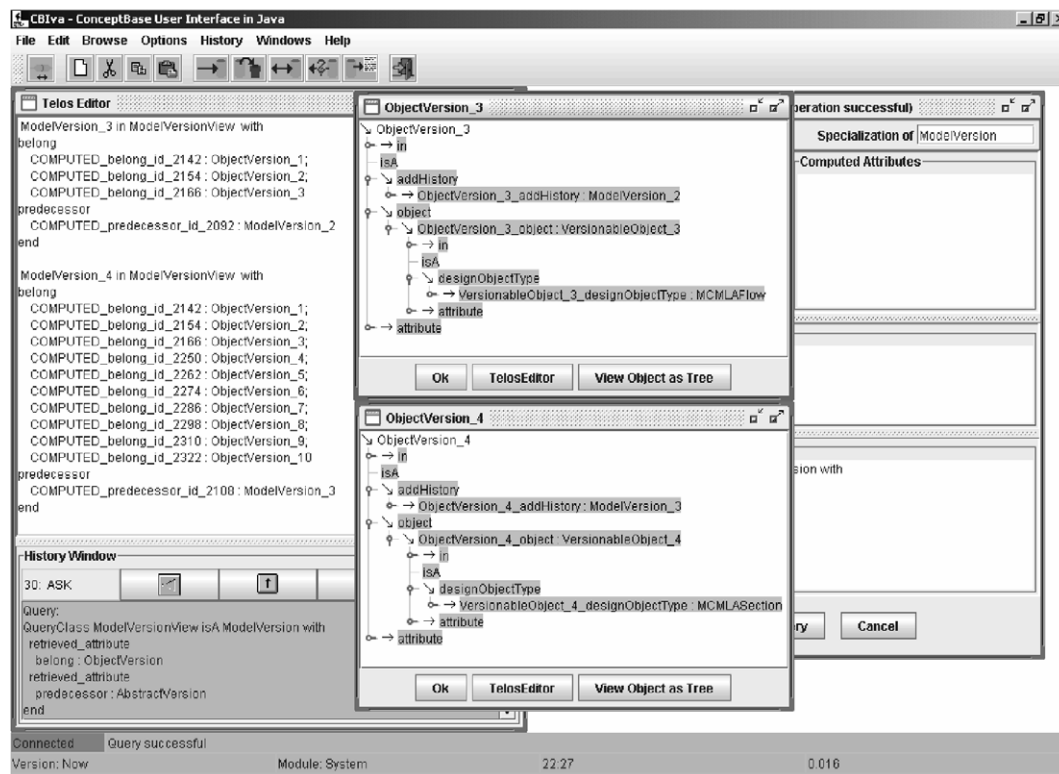


Fig. 14. Partial view of object versions belonging to the contexts of the CreateRecycleStructure activity.

For illustration purposes, let us consider the case study employed. The *ModelVersion_5* model version (Fig. 15) represents the state of design process after the literature review has been carried out (*LiteratureResearch* activity), the flowsheet synthesis activity revealing that there are three main parts of the process to be designed (reaction, separation and extrusion) has been executed (*CreateAFD* activity) as well as the decision between batch and continuous operation has been made (*AlternativesBatchCont* activity). A block flow diagram representing the continuous polymerization of polyamide-6 is shown in the partial view of *ModelVersion_5* in Fig. 15. Then, *DesignReaction*

activity carries out the *reaction design* and the *separation design* is performed by *DesignSeparation* activity. Therefore, distinct design actors concurrently work on slightly coupled parts of the artefact being designed, and it is necessary to unify these partial design representations in a unique model version representing the complete state of the design process. In this case, conflict handling must be addressed along the “parallel” course of actions, generating the new model version *ModelVersion_8* (Fig. 15) that unifies both representations.

As seen in the previous paragraph, a cooperative design environment may lead to several design state

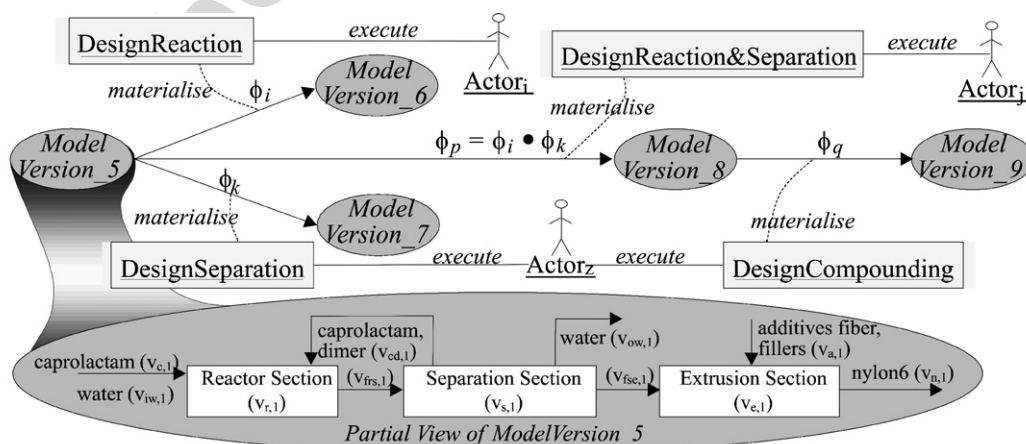


Fig. 15. Conceptual representation of the capture of the design process.

representations at a given time point (*ModelVersion_6* and *ModelVersion_7* in Fig. 15) and a new model version that unifies them has to be generated (*ModelVersion_8* in Fig. 15). Thus, this model version has to unify the work performed by various actors (activities *DesignReaction* and *DesignSeparation* in the scenario illustrated in Fig. 15). For this reason, the new model version can be inferred as the application of the sequence of operations resulting from the concatenation of sequences ϕ_i and ϕ_k to the model version *ModelVersion_5*. Thus, ϕ_p is equal to $\phi_i \bullet \phi_k$ and *ModelVersion_8* results from $apply(\phi_i \bullet \phi_k, ModelVersion_5)$. Then, potential conflicts can arise there and the next subsections provide specific procedures to handle different situations. Once a conflict has been detected, the information captured and structured by the design process model (Fig. 1) enables capitalising the knowledge about how this conflictive situation has been obtained.

5.1. Detecting conflicting concurrent versions

Usually, actors operate on design objects, which are the basis for solutions obtained by other actors. If these objects change, some solutions may turn out to be inconsistent.

In order to achieve this consistent model merging between *ModelVersion_6* and *ModelVersion_7*, it is necessary to find if the sequence of operations resulting of the concatenation of two sequences ($\phi_p = \phi_i \bullet \phi_k$) could be applied or not to the common model version (*ModelVersion_5*). Thus, it is necessary to specify the preconditions to apply a sequence of operations ϕ to a given model version m , fact that is expressed by the $poss_{so}(\phi, m)$ predicate in expression (38).

$$(\forall \phi : SequenceOfOperations, m : ModelVersion) \\ poss_{so}(\phi, m) \iff (\forall op_i : Operation, op_i \in \phi) \\ poss_o(op_i, \phi, m) \quad (38)$$

Therefore, if a precondition is violated a conflict is detected (expression (39)). Now, a sequence of operations ϕ may be applied to a model version m if each operation op_i belonging to ϕ can be applied to m , as well as op_i can be applied in all the situations generated by applying the $i - 1$ previous operations in the sequence, where op_i is the i th operation belonging to ϕ . This fact that is defined by the $poss_o(op_i, \phi, m)$ predicate, is introduced in expression (40).

$$(\forall \phi_1, \phi_2 : SequenceOfOperations, m : ModelVersion) \\ conflict_{so}(\phi_1, \phi_2, m) \Leftarrow poss_{so}(\phi_1, m) \wedge poss_{so}(\phi_2, m) \\ \wedge \neg poss_{so}(\phi_1 \bullet \phi_2, m) \quad (39)$$

$$(\forall \phi : SequenceOfOperations, m : ModelVersion) \\ (\forall op_i : Operation, op_i \in \phi, \\ \exists \phi_1, \phi_2 : SequenceOfOperations, \phi = \phi_1 \bullet op_i \bullet \phi_2) \\ poss_o(op_i, \phi, m) \iff (\forall m_i : ModelVersion, \\ m < m_i \leq apply(\phi_1 \bullet op_i, m)) poss(op_i, m) \quad (40)$$

The $poss(op, m)$ predicate expresses that an operation op is applicable to a given model version m . This fact is represented by the following axioms:

- operation $add(v)$ can be applied to model version m if the object version v does not belong to m (expression (41));
- operation $delete(v)$ can be applied to the model version m if the object version v belongs to m (expression (42));
- operation $modify(v_i, v_j)$ can be applied to m if v_i belongs to m and v_j does not belong to it (expression (43)).

$$(\forall v : ObjectVersion, m : ModelVersion) \\ poss(add(v), m) \iff \neg belong(v, m) \quad (41)$$

$$(\forall v : ObjectVersion, m : ModelVersion) \\ poss(delete(v), m) \iff belong(v, m) \quad (42)$$

$$(\forall v_i, v_j : ObjectVersion, m : ModelVersion) \\ poss(modify(v_i, v_j), m) \iff belong(v_i, m) \wedge \neg belong(v_j, m) \quad (43)$$

It should be remarked that this type of formalization can also be made for extended operations. Since extended operations are expressed in terms of basic ones, the possibility of applying an extended operation can be formally expressed by resorting to the primitive operation axioms (expressions (41)–(43)).

Note that a sequence ϕ can be applied to m if each operation of ϕ is applicable to m and does not violate the preconditions of the other operations belonging to ϕ . Fig. 16 illustrates the sequence of operations ϕ_p applied to model version m_0 , resulting on model version m_p . Operations op_1, op_2, \dots, op_n belong to the sequence ϕ_p . It must be remarked that the framework does not store all model versions, the only model versions kept there are those generated after applying a sequence of operations; in Fig. 16, model versions m_o and m_p . Thus, as the sequence is $\phi_p = \{op_1, op_2, \dots, op_n\}$, there exists $n - 1$ model versions ($m_{op1}, m_{op2}, \dots, m_{opn-1}$) that are not maintained by the framework. They are inferred by means of expression (33). Therefore, the sequence ϕ_p can be applied to model version m_o (expression (38)) if each operation op_i that comprises it can be applied to m_o , and the $i - 1$ successor model versions generated when the operations are applied (expression (40)). Then, $poss_{so}(\phi_p, m_o)$ is true if $poss_o(op_i, \phi_p, m_o)$ is true for each operation op_i that belongs to ϕ_p . In this case, $poss_o(op_1, m_o)$, $poss_o(op_2, m_o)$, $poss_o(op_2, m_{op1})$, \dots , $poss_o(op_n, m_o)$, $poss_o(op_n, m_{op1})$, \dots , $poss_o(op_n, m_{opn-1})$.

As a consequence, if the sequence of operations resulting from the concatenation of sequences the ϕ_i and ϕ_k has to be applied to *ModelVersion_5*, to unify both model versions, $poss_{so}(\phi_i \bullet \phi_k, ModelVersion_5)$ will be true if no conflicts among the applied operations appear. A conflict will arise if an operation modifies the preconditions needed to perform another operation of the sequence. Expression (44) introduces the $conflict_o(\phi, op_i, m)$ predicate to determine which operation op_i produces the conflict.

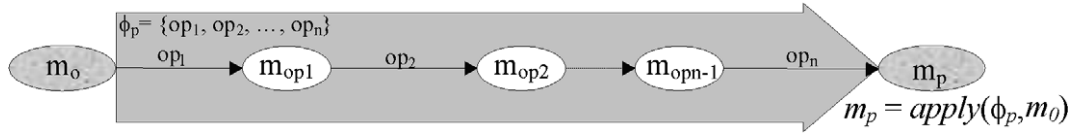


Fig. 16. Version generation approach.

$$\begin{aligned}
 &(\forall \phi: \text{SequenceOfOperations}, m: \text{ModelVersion}) \\
 &(\exists op_i: \text{Operation}, op_i \in \phi) \text{conflict}_o(\phi, op_i, m) \\
 &\iff \neg \text{poss}_o(op_i, \phi, m)
 \end{aligned} \quad (44)$$

For illustration purposes, if the design of the reaction and the separation sections are carried out by activities *DesignReaction* and *DesignSeparation* (Fig. 15), respectively, then, these activities are materialised by sequences ϕ_i and ϕ_k , and $\text{poss}_{so}(\phi_i, \text{ModelVersion}_5)$ and $\text{poss}_{so}(\phi_k, \text{ModelVersion}_5)$ are true. If the actors operate on disjoint subsets of object versions belonging to *ModelVersion_5*, then $\text{poss}_{so}(\phi_i \bullet \phi_k, \text{ModelVersion}_5)$ is true and that means that there is no conflict ($\text{conflict}_{so}(\phi_i, \phi_k, \text{ModelVersion}_5)$ evaluates to false) and model version *ModelVersion_8* is obtained. However, when both actors operate on and modify the same design object, conflicts may appear. For instance, when they modify the same stream that connects the output of the reactor section with the input of the separation section ($v_{frs,1}$ in Fig. 15), the predicate $\text{poss}_{so}(\phi_i \bullet \phi_k, \text{ModelVersion}_5)$ evaluates to false, thus indicating that there is a conflict ($\text{conflict}_{so}(\phi_i, \phi_k, \text{ModelVersion}_5)$ is true). Let $v_{frs,1}$ be the original object version that belongs to *ModelVersion_5* ($\text{belong}(v_{frs,1}, \text{ModelVersion}_5)$ is true), representing the conflicting stream. A new version of the stream is generated (belong($v_{frs,2}$, *ModelVersion_6*) is true) by the execution of a modify operation ($\text{modify}(v_{frs,1}, v_{frs,2}) \in \phi_i$) while the reactor design activity is carried out. In parallel, the separation design activity creates another version of the stream (belong($v_{frs,3}$, *ModelVersion_7*) is true) due to another modify operation ($\text{modify}(v_{frs,1}, v_{frs,3}) \in \phi_k$). This scenario presents a conflict because $\text{modify}(v_{frs,1}, v_{frs,2})$ deletes the precondition needed to perform $\text{modify}(v_{frs,1}, v_{frs,3})$. Therefore, $\text{poss}_o(\text{modify}(v_{frs,1}, v_{frs,3}), \phi_i \bullet \phi_k, \text{ModelVersion}_5)$ evaluates to false.

5.2. A heuristic to extend the conflict detection process

It is important to consider that the $\text{conflict}_{so}(\phi_1, \phi_2, m)$ predicate introduced in expression (39) does not cover all possible causes of conflicts when the work carried out by various actors is unified in a new model version. When an actor uses a version of a design object as a source of information to synthesize an artefact and, in parallel, another actor modifies or deletes this version, a potential conflict may exist and it has to be detected.

For example, Fig. 17 presents a scenario on the case study being considered. Let $v_{frs,1}$ be an object version of a stream that belongs to *ModelVersion_5*. The stream interconnects the reactor ($v_{r,1}$) with the separation section ($v_{s,1}$). A new version of the stream is generated while the reactor design activity is carried out, because a modify operation ($\text{modify}(v_{frs,1}, v_{frs,2}) \in \phi_i$) is performed. In parallel, the separation design activity synthesizes the separation section, evolving it into a more detailed representation as a result of a *refine* operation ($\text{refine}(v_{s,1}, \{v_{s1,1}, v_{s2,1}, v_{s3,1}\}) \in \phi_k$). To perform this operation, actor z does not modify the stream version ($v_{frs,1}$) but uses it. Fig. 17 shows that the result obtained by applying ϕ_p to *ModelVersion_5* is the *ModelVersion_8* model version, where the input of $v_{s1,1}$ is $v_{frs,2}$. In this situation, $\text{poss}_{so}(\phi_i \bullet \phi_k, \text{ModelVersion}_5)$ will evaluate to true. Actually, there is a potential conflict because the input of $v_{s1,1}$ is $v_{frs,2}$ in *ModelVersion_8* and it is not the input stream version ($v_{frs,1}$) used by actor z when he/she was executing *DesignSeparation* activity.

To identify such potential problem, the following heuristic is defined: given an initial model version, m_o , two concurrent successor model versions, m_i ($m_i = \text{apply}(\phi_i, m_o)$) and m_k ($m_k = \text{apply}(\phi_k, m_o)$), and two object versions v_1 and v_2

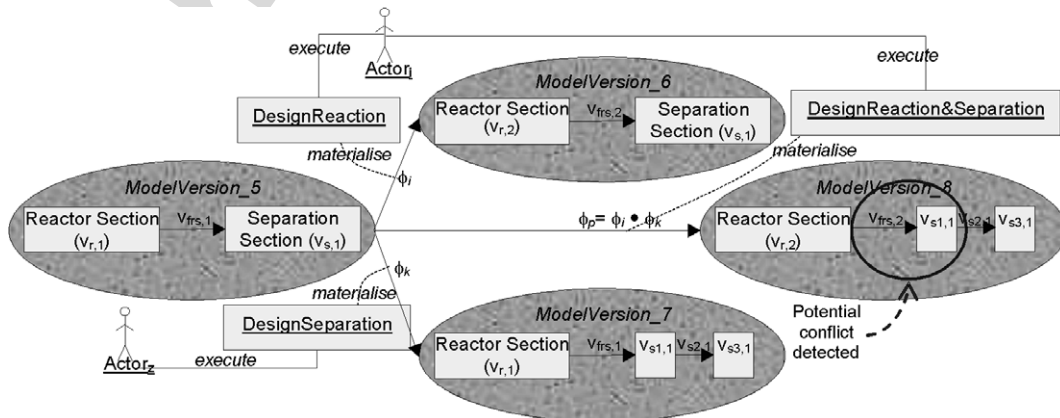


Fig. 17. Capturing the Polyamide-6 design process. Concurrent work on slightly coupled parts.

belonging to model version m_p ($m_p = \text{apply}(\phi_i \bullet \phi_k, m_o)$), then a potential conflict would occur in model version m_p , as a result of applying sequence $\phi_i \bullet \phi_k$ to the previous model version m_o , if it is verified that:

- versions v_1 and v_2 are linked by an association a_k (this association is inferred by means of expression (35)),
- and versions v_1 and v_2 do not belong to the previous model version m_o ,
- and one of the versions (v_1) belongs to the model version which is a consequence of applying ϕ_i to m_o ,
- and the other version (v_2) belongs to the model version which is the product of applying ϕ_k to m_o .

This heuristic is formally stated by expression (45).

$$\begin{aligned}
 (\forall \phi_j, \phi_k: \text{SequenceOfOperations}, m: \text{ModelVersion}, \\
 v_1, v_2: \text{ObjectVersion}) \text{potentialConflict}(v_1, v_2, \phi_j, \phi_k, m) \\
 \Leftarrow (\exists a_k: \text{Association}) \text{inferredAssociation} \\
 (a_k, v_1, v_2, \text{apply}(\phi_j \bullet \phi_k, m)) \wedge \neg \text{belong}(v_1, m) \\
 \wedge \text{belong}(v_1, \text{apply}(\phi_j, m)) \wedge \neg \text{belong}(v_2, \text{apply}(\phi_j, m)) \\
 \wedge \neg \text{belong}(v_2, m) \wedge \text{belong}(v_2, \text{apply}(\phi_k, m)) \\
 \wedge \neg \text{belong}(v_1, \text{apply}(\phi_k, m))
 \end{aligned} \quad (45)$$

The $\text{potentialConflict}(v_{fs,2}, v_{s1,1}, \phi_i, \phi_k, \text{ModelVersion}_5)$ predicate, in the scenario presented by Fig. 17, evaluates to true when a potential conflict exists between $v_{fs,2}$ and $v_{s1,1}$ in the model version resulting from applying the sequence $\phi_i \bullet \phi_k$ to ModelVersion_5 .

6. Conclusions

The main contribution of this paper is the proposition of an integrated model of the knowledge employed in the engineering design process (*CoMoDe*), a first step towards the development of a computational environment to support that process. As Marquardt and Nagl (2004) explained, one of the most important issues to successfully establish design process excellence is a lack of common understanding and terminology related to the design process and its results. In order to tackle this problem, Yang and Marquardt (2004) suggested the use of the ontology technology to provide physicochemical concepts as the basis of an ontology-based conceptual modelling tool. *CoMoDe* goes further on, representing the essential concepts of design processes and their products in an integrated way. Therefore, the integrated model proposed in this paper provides two fundamental steps towards the development of computational tools to support the engineering design process and to guide designers in the different activities of a design project. First, the model proposed in the operation context granularity level offers an explicit mechanism to capture and trace the different model versions that participated in the design process. Second, the knowledge captured in the operation context level is structured and organised with the aim to manage the knowledge acquired; therefore, the

knowledge captured is structured in several representation spaces and allows us to know the activities, operations and actors that generated each design product, the requirements that were imposed as well as the rationale behind each adopted decision. It also enables the analysis of the reasoning line employed during the design process, setting the grounds for learning and future reuse.

The main capabilities of *CoMoDe* are listed below:

- The proposal allows us to represent the various design states by capturing design object versions that arise during a design process. It is worth noticing that not only the states of the artefact being designed can be represented but also the different states of all design objects, i.e., artefacts, requirements, arguments, resolutions, positions, attributes, and values. The design objects to be captured can be defined according to the needs of a particular domain. It is possible by the domain model integrated in the version administration model (see Fig. 10).
- The requirement concept allows the representation of design goal aspects. Its definition is extensible according to a given design domain.
- The proposed extension of the IBIS model allows representing the design decisions and their rationale by an explicit model of the alternatives' selection process and the justification of a given decision by means of arguments. Moreover, the IBIS concepts are modelled as design objects, and then their evolution can be captured. Therefore, it is possible to represent how a decision taken (represented by the resolution concept and the items associated with it) was changed when a greater knowledge was acquired (mainly represented by the argument concept). Taking this point into account, an important contribution is the representation of the history of the operations that originated a given design. This knowledge establishes the basis for knowing how a given version of a design object was obtained, and to understand how a design process was carried out. This knowledge is also fundamental to define the partial workflow of design processes.
- Design processes are captured by introducing a minimal impact on the design activities performed by designers. This feature establishes a distinction with the proposal of Eggersmann, Schneider, and Marquardt (2002), which considers capturing the design process by interviewing the designer after completing (part of) the design process.
- Situational calculus represents the activities carried out during a design process, and therefore, it enables to capitalise the information on how the various design objects were obtained. Thus, the history of operations performed on versions of design objects can be kept. Furthermore, the preconditions needed to perform a sequence of operations enable the detection of potential conflicts between model versions developed in parallel. This conceptual framework provides the foundations for the proposal of formal means for detecting potential conflicts.

The model was formally specified using the O-Telos language, which allows the integration of object-oriented technology with the first order logic in a concise representation. Finally, the proposed model was tested by applying it to a case study on the design of a polyamide-6 plant.

Acknowledgement

The authors wish to acknowledge the financial support received from CONICET, Universidad Nacional del Litoral, Universidad Tecnológica Nacional and Agencia Nacional de Promoción Científica y Tecnológica (PICT 12628).

References

- Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice (2nd ed.). Addison-Wesley.
- Bayer, B., & Marquardt, W. (2004). Towards integrated information models for data and documents. *Computers and Chemical Engineering*, 28, 1249–1266.
- Booch, G., Rumbaugh, J., & Jacobson, I. (2005). The unified modeling language user guide (2nd ed.). Addison-Wesley Professional.
- Boyle, J.-M. (1989). Interactive engineering system design: a study for artificial intelligence applications. *Artificial Intelligence in Engineering*, 4, 58–69.
- Brown, D., & Chandrasekaran, B. (1989). Design problem solving. Knowledge structures and control strategies. Pitman.
- Cameron, I., Fraga, E., & Bogle, I. (2005). Process modelling goals: concepts, structure and development. In L. Puigjaner & A. Espuña (Eds.). *European symposium on computer-aided process engineering* (vol. 15, pp. 265–270). Elsevier.
- Carnduff, T., & Goonetillake, J. (2004). Configuration management in evolutionary engineering design using versioning and integrity constraints. *Advances in Engineering Software*, 35, 161–177.
- Eggersmann, M. (2004). Analysis and support of work processes within chemical engineering design processes. *Fortschritt-Berichte, VDI Reihe 3, Nr. 840*, Düsseldorf, VDI-Verlag.
- Eggersmann, M., Krobb, C., Gonnet, S., Mannarino, G., Leone, H., & Henning, G. (1999). Modeling of the design process as an enterprise activity. In *Proceedings of Enpromer'99*, Brasil.
- Eggersmann, M., Henning, G., Krobb, C., Leone, H., & Marquardt, W. (2001). Modeling of actors within a chemical engineering work process model. In *Proceedings international CIRP design seminar* (pp. 203–208).
- Eggersmann, M., Schneider, R., & Marquardt, W. (2002). Modeling work processes in chemical engineering – from recording to supporting. In J. Grievink & J. van Schijndel (Eds.). *European symposium on computer aided process engineering* (vol. 12, pp. 871–876). Elsevier.
- Eggersmann, M., Gonnet, S., Henning, G., Krobb, C., Leone, H., & Marquardt, W. (2003). Modeling and understanding different types of process design activities. *Latin American Applied Research*, 33, 167–175.
- Floudas, C. (1995). Nonlinear and mixed-integer optimization: fundamentals and applications. Oxford University Press.
- Fox, M., & Gruninger, M. (1998). Enterprise modelling. *AI Magazine*, 19(3), 109–121.
- Goel, V. (1994). A comparison of design and nondesign problem spaces. *Artificial Intelligence in Engineering*, 9, 53–72.
- Gruber, T. (1995). Toward principles of the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5, 6), 907–928.
- Gzara Yesilbas, L., & Lombard, M. (2004). Towards a knowledge for collaborative design process: focus on conflict management. *Computers in Industry*, 55, 335–350.
- Harzallah, M., & Vernadat, F. (2002). IT-based competency modeling and management: from theory to practice in enterprise engineering and operations. *Computers in Industry*, 48, 157–179.
- Heller, M., Jäger, D., Schlüter, M., Schneider, R., & Westfechtel, B. (2004). A management system for dynamic and interorganizational design processes in chemical engineering. *Computers and Chemical Engineering*, 29, 93–111.
- Jarke, M., Jeusfeld, M., & Quix, C., editors. (2004). ConceptBase V6.2 user manual.
- Jarke, M., List, T., & Weidenhaupt, K. (1999). A process-integrated conceptual design environment for chemical engineering. *Lecture Notes in Computer Science*, 1728, 520–537.
- Kitamura, Y., & Mizoguchi, R. (2003). Ontology-based description of functional design knowledge and its use in a functional way server. *Expert Systems with Applications*, 24, 153–166.
- Kunz, W., & Rittel, H. W. J. (1970). Issues as elements of information systems. Institute of Urban and Regional Development. Working Paper 131. University of California, Berkeley.
- Kvan, T. (2000). Collaborative design: what is it? *Automation in Construction*, 9, 409–415.
- Liao, S. (2005). Expert system methodologies and applications – a decade review from 1995 to 2004. *Expert Systems with Applications*, 28, 93–103.
- Mandow, L., & Pérez-de-la-Cruz, J. (2004). Sindi: an intelligent assistant for highway design. *Expert Systems with Applications*, 27, 635–644.
- Mannarino, G. S. (2001). Coordinates, Un lenguaje para el modelado de empresas. PhD thesis, Universidad de Buenos Aires, Argentina.
- Marquardt, W., & Nagl, M. (2004). Workflow and information centered support of desing process – the IMPROVE perspective. *Computers and Chemical Engineering*, 29, 65–82.
- Mittal, S., & Araya, A. (1992). A knowledge-based framework for design. *Artificial intelligence in engineering design (vol. 1): design representation and models of routine design* (pp. 273–296).
- Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1990). Telos: representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4), 352–362.
- Nagl, M., & Marquardt, W. (1997). SFB-476 IMPROVE: Informatische Unterstützungübergreifender Entwicklungsprozesse in der Verfahrenstechnik. In M. Jarke, K. Pasdach, & K. Pohl (Eds.), *Informatik '97: Informatik als Innovationsmotor Informatik aktuell* (pp. 143–154). Springer-Verlag.
- Nagl, M., Westfechtel, B., & Schneider, R. (2003). Tool support for the management of design process in chemical engineering. *Computers and Chemical Engineering*, 27, 175–197.
- Reiter, R. (2001). Knowledge in action: logical foundation for describing and implementing dynamical systems. The MIT Press.
- Roda, I., Poch, M., & Bañares-Alcántara, R. (2000). Application of a support system to the design of wastewater treatment plants. *Artificial Intelligence in Engineering*, 14, 45–61.
- Roldán, M. L., Gonnet, S., & Leone, H. (2005). A version support model for architecture based design process. In ASSE 2005: Simposio Argentino de Ingeniería de Software (pp. 19–33).
- Scherl, R., & Levesque, H. (2003). Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1–2), 1–39.
- Stephanopoulos, G., Henning, G., & Leone, H. (1990). MODEL.LA. A modeling language for process engineering. Part I: The formal framework. *Computers and Chemical Engineering*, 14(8), 813–846.
- Vernadat, F. (1996). Enterprise modelling and integration: principles and applications. Chapman & Hall.
- Westerberg, A., Subrahmanian, E., Reich, Y., Konda, S., & the n-dim group (1997). Designing the process design process. *Computers and Chemical Engineering*, 21(Suppl.), S1–S9.
- Westfechtel, B. (1999). *Models and tools for managing development process. Lecture notes in computer science* (1646). Springer.
- Yang, A., & Marquardt, W. (2004). Ontology-based approach to conceptual process modeling. In A. Barbosa-Póvoa & H. Matos (Eds.). *European symposium on computer-aided process engineering* (vol. 14, pp. 1159–1164). Elsevier.