Available online at www.sciencedirect.com

SciVerse ScienceDirect

Energy Procedia 18 (2012) 1458 – 1484

Energy Procedia

Generating Interface Prototype for EnergyPlus IDD file using Unified Modeling Language and Coloured Petri-nets

Bachkhaznadji Abdeldjebbar[a,]*, Belhamri Azeddine[a]

[a]*Département de Génie Climatique, Université Mentouri, Route ain El Bey Constantine 25000, Algeria*

Abstract

Based on user requirement techniques, this paper suggests a new approach to the generation of a building performance simulation tool interface prototype. This approach requires the Unified Modelling Language (UML) and scenario specifications for analysing user interactions with the computer program core. At the level of user interface specifications, designer can draw interfaces but it is difficult to specify their dynamic behaviour. Coloured Petri-nets are used in specification and check on the behaviour of these interfaces before their implementation. The EnergyPlus input data definition (IDD) file is provided as an example to illustrate the process of generating an executable interface prototype which can be used for scenario validation, and can be evolved towards the target interface tool. The result of the overall approach is a specification consisting of a global coloured Petri-net which can be analysed by means of a simulation together with the generated and refined IDD interface prototype

Keywords: Building; Energy simulation; Data input; Interface prototype

1. Introduction

The building of user interfaces (UI) is becoming a significant part of software engineering. Over the past years, we saw appearing many methods and tools allowing reducing the workload of developers, and producing interfaces of better quality. At the beginning of the eighties, the term UI was practically unknown. Developers were interested first of all in the validity of the applications and that they

* Corresponding author. Tel.: +213-055-030-4061; fax: +213-031-819-012.
E-mail address: b.abdeldjebbar@voila.fr.

1876-6102 doi:10.1016/j.egypro.2012.05.164

accomplish the treatments which were waited for. Since their appearance, user interfaces did not cease evolving by bringing more suppleness to the user, and by answering to its needs efficiently.

The acceptance of these interfaces is not due to the technological advances of computer sciences, but it also returns to the result of the cognitive sciences which are interested in the study of human interaction with computer systems. By the integration of the ergonomic rules (easiness of use, brevity, coherence, suppleness, etc.), these interfaces deserved the evaluation of users. Another main reason of this success comes due to fact that the user is principal of the correlation with the application during all working session.

The implementation of these types of UI brings however a new complexity in the task of software development. The complexity resides in the description of multi-task dialogue which is run by the user to lead several tasks at the same time. At the level of the specification of UI, developers can draw interfaces but it is often difficult to specify their dynamic behavior due to the complexity of the dialogue. Formal methods (high level Petri-nets, Harel's Statecharts, etc.) will be of big utility in specification and check on the behavior of these user interfaces before their implementation, which will allow reducing the time of tests, and of improving the quality of the UI.

For the description of user interactions with a computer program, and for the investigation of new solutions in re-engineering approaches, scenarios have received significant attention, and have been identified as an effective means for understanding user requirements, and for analyzing human computer interaction [1]. They are used in various activities of the cycles of development, since the acquisition of user needs up to the activities of test by way of specifications and prototypes, and by the evaluation of various alternatives of conception. It is therefore possible to say that scenarios are used for purposes of description and of exploration.

Scenarios also find their place in most object oriented methods (OMT, OOSE, etc.) to identify objects of the system and to represent its functional requirements. Object oriented analysis and design methods (modeling methods) become as well an essential part of a software development. Software design that would be difficult to describe textually, can readily be conveyed through diagrams. Modeling provides three keys benefits: visualization, complexity management, and clear communication. The unified modeling language (UML) which is part of the object management group (OMG) is a fruit of efforts of unification of several object-oriented methods. UML is a visual language for specifying, constructing, and documenting the artefacts of a system through models and diagrams, and it offers a good framework for scenarios engineering.

Any respectable process of software requirements engineering to an interactive system (a program offering an interface run by a user) will have to produce a specification of the behaviour of the system for purposes of validation, check, and for user evaluation of the UI prototype. In view of the adequacy in the acquisition of user requirements, scenarios are used as a means of description of the system behaviour. Being of partial descriptions, scenarios of a modelled system require an operation of integration, and the resultant specification needs to be formalized by formal methods. The system behaviour can be investigated before it is constructed using, for example a high level Petri-net model which can be analysed either by means of simulation (which is equivalent to program execution and program debugging) or by means of more formal analysis methods [2]. The process of creating the description and performing the analysis usually gives the developer an improved understanding of the modelled system.

EnergyPlus, among many building performance simulation programs, was purposefully developed as a simulation engine without user friendly graphical interface (UFGI) [3, 4, 5]. The program reads input and writes output as text files. An academic study on building energy simulation [6], stipulates that input data of the EnergyPlus program are still fairly laborious, and the program will not be adopted for general use without proper graphical input interfaces.

A number of published works on interface design for building performance simulation (BPS) tools [7] agree on the necessity of developing user interfaces capable of handling efficiently user needs by, for instance, reducing extensive input data that define a building and its systems [8], or by the creation of new user interfaces with complete functions for fast HVAC researching [9] etc. Also, literature and comparative surveys [10] indicate that most users who make use of BPS tools in design practice are much more concerned with the (1) usability and information management (UIM) of interface and (2) the integration of intelligent design knowledge-base (IIKB).

The object oriented programming (OOP) languages such as, visual Basic, visual C/C++, Python, etc. are the primary development tools proposed by most studies in their development of interfaces. Two interfaces (EPlusInterface, EasyEnergyPlus), for instance, suggested for the EnergyPlus program, are developed using exclusively two different OOP languages like Python and Visual Basic 6.0. All of these OOP languages do not allow for any user requirement engineering process; however they can make part of the procedure to code applications.

So far, no work on user requirement process, based on UML and scenario specifications, has been suggested in the field of UI design for BPS tools, particularly for EnergyPlus program. The process, if applied, will provide to the UI development with additional benefits which are as follows:

- Maintain a high level of communication between a team of developers by offering a language independent and a platform independent modelling method;
- Reduce time of interface developments;
- Improve and increase user involvement by simulation and interface prototyping;
- Describe interface behaviours as a formal method which will boost confidence in the correctness of interfaces by proof, refinement, and testing.

In the following subsections of the introduction, we will discuss some concepts and definitions, which are closely related to the approach, in respect to the following points: (1) Scenarios, (2) UML notation and scenarios, (3) Coloured Petri-nets. To better explain these concepts, we have adopted the structure of the input data dictionary (IDD) file of the EnergyPlus program as an example, since the latter is created with no formal user interface.

As defined in the input/output reference guide of EnergyPlus program (EnergyPlus v2.0 or latter), a data entry in the IDD file consists of a comma separated text by a semi column, where an entry defines a class of input objects and specifies all the data needed to model it. The IDD structure defines an input object (a record with data) as a key word. Each data item (a field) to the input object can be A's (alphanumeric) or N's (numeric) field. As noted in the IDD file, \ character is adopted as a convention for including comments which are specific for each data field in an input object. For illustration, a sample of an input data entry portion of a 'COIL: water: simple heating' input object with its associated data fields is given below:

```
COIL: Water: SimpleHeating,
  A1, \ field coil name
      \ type alpha
      \ required field
  A2, \ field available schedule
      \ type object-list
  N1; \ field UA of the coil
      \ units W/k
      \ autosizable
```

1.1. Scenarios

Several definitions of the term scenario exist in literature. The common denominator in these definitions is that a scenario describes a partial view of a system. Scenarios allow giving a detailed picture of the functional and organizational aspects of the system and link up user requirements with the functionality of the system. In the field of UI development, scenarios are seen as a use of a computer program in a given context. Scenarios find also their place in most object-oriented methods (OMT, Fusion, OOSE, etc.) to identify objects of the system and to represent functional requirements. In OOSE method, for example, scenarios (use cases) run all stages of the object-oriented design.

UML which is part of standards of OMG (Object Management Group) is the fruit of efforts of unification of several object-oriented methods. Unification concerned only methods OMT and Booch. The addition of the OOSE (object-oriented software engineering) method allowed to UML to have a rich notation covering all stages of development. Terms scenario and use case were used in several works to indicate the same concept. In UML a separation between both terms is made, and exact definitions are given. A use case is defined as a continuation of interactions with the system with the intention of accomplishing a whole transaction (task of the system). It is generally described by several scenarios. This will allow an acquisition structured by scenarios by use case. A scenario is therefore an authority of a use case, which describes a possible continuation of interactions in the realisation of a system task.

1.1.1 Aspect of scenarios

Scenarios evolved in time according to several aspects, and their interpretation seems to depend on the context of use and of the way they were acquired or generated. In a job of summing-up, Rolland et al. (1998) [11] offered a frame for the classification of scenarios according to four aspects: form, content, purpose and cycle of development, see Fig 1.

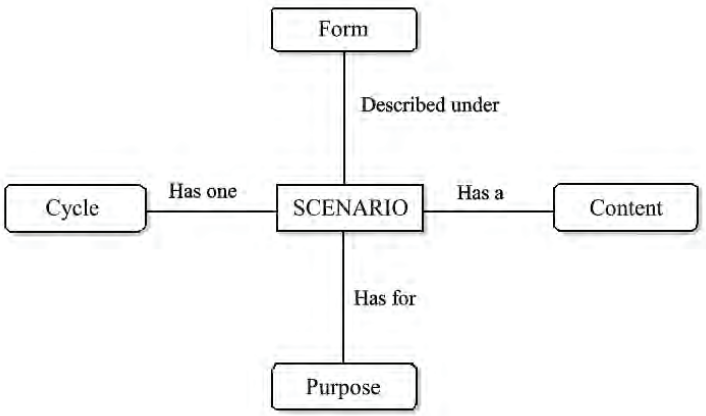

Fig. 1. classification of scenarios

1.1.1.1. Form of scenarios

The form of scenarios concern principally operations of acquisition and of specification. During the description of scenarios, modeling languages (semi-formals) or formal specification techniques can be

used for the validation of user requirements. For instance, in UML, scenarios are described by diagrams of collaboration or of sequence (section 1.2.3); scenarios are described as well using formalism derived from high level Petri-nets(section 1.3) [12, 13] etc.

1.1.1.2. Content of scenarios

Contents of scenarios depend of the way scenarios describe the system. We distinguish between abstract scenarios and concrete scenarios. Abstract scenarios refer to a way objects of a system are abstracted (i.e.: input-object, numeric-Field, alphanumeric-field, etc.), while concrete scenarios use particular instances of input objects (i.e.: COIL: Water: SimpleHeating; UA of the Coil (N1); coil name (A1), etc.), see the above example of section 1.

1.1.1.3. Purpose of scenarios

Scenarios are used for the capture of user requirements, for the description of user interaction with a system, and for the investigation of new solution in re-engineering approaches [14]. It is therefore possible to say that scenarios are used for purposes of description and of exploration. The scenarios of description are especially used to include and to describe behavioural aspects of a system, where scenarios represent points of view of external users. The scenarios of exploration are used when several solutions are to explore and to assess with the intention to choose the best.

1.1.1.4. Life cycle of scenarios

If we consider scenarios describing a system as being objects of a system, it is then possible to speak about persistent scenarios by analogy to the persistent objects. Persistent scenarios accompany the cycle of the development since the analysis of needs up to the production of material [15]. By opposition, they define temporary scenarios as having served only at the level of some stages of the cycle of development. It is the case, for example, where scenarios were only used for the acquisition of needs, or to support the production of diagram of states of the objects of the system [16] or for the validation of needs [17].

1.2. UML notation and scenarios

UML [18] is an object-oriented modelling language. It is the fruit of efforts of unification of several object-oriented methods with the aims of simplifying and of profiting from all advantages of these methods. It was conceived to be used as a modelling language, independently of the conventional OOP languages. UML has been evolved with many versions since it started officially in 1994. The latest release is UML 2.0 which this paper refers to. UML defines nine types of diagrams, each one of them represents a specific vision of the system:

- Functional or interactive view, described with the assistance of use case diagrams, sequence diagrams, and collaboration diagrams.
- Structural or static view, represented with the assistance of class diagrams, object diagrams, component diagrams, and deployment diagrams.
- Dynamic view, expressed by statechart diagrams, and activity diagrams.

In what follows, we first discuss UML diagrams that are relevant for our approach: use case diagram, sequence diagram, and class diagram.

1.2.1. Use case diagram

The use case diagram is a contribution of Ivar Jacobson in UML. A use case diagram describes the interactions between external actors and the system being modelled. It describes the sequence of actions carried out by a system with an aim of offering a service to the actors. A use case is a summary of scenarios for a simple task or goal. An actor (user) is who or what initiates the events involved in that task. Use cases are represented as ellipses, and actors are depicted as icons connected with solid lines to the use cases with which they interact. One use case can call upon the services of another use case. Such a relation is called an include relation, and its direction does not imply any order of execution.

Fig 2 shows the proposed use case diagram of the IDD interface. There is an actor (i.e. module developer) interacting with five use cases: *Identify_object*, *Generate_object*, *Edit_object*, *Delete_object*, *and Save_object to* IDD file. The use case *Generate_object*, call upon the service of the use case *Identify_object*, and include it as a subtask. An identification of an input object class may be a group of related input objects, or an input object, or a data field. Furthermore, the UML comprises the extend relation, which can be considered as a variation of the include relation, as well as generalisation relation which indicates that a use case is a special kind of another use case [18]. Use case diagram is helpful in visualizing the context of our application domain and the boundaries of the whole of the interface behaviour. An execution of a use case will typically be characterized by multiple scenarios (section 1.2.3).

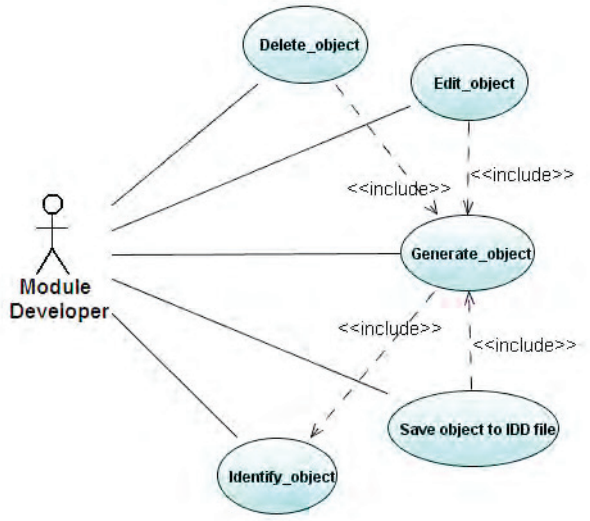

Fig. 2. use case diagram of the IDD interface prototype

1.2.2. Class diagram

A class diagram gives an overview of the system by showing its classes and the relationship among them. A class diagram is static, and is used to detail the pattern from which objects will be produced at run time. A class diagram identifies all the classes for a proposed system and specifies for each class its attributes, operations, and relationships to other classes. Relationships include inheritance, association, and aggregation. The class diagram is the central diagram of UML models.

In our previous work [19], we detailed the construction of a class diagram of the IDD file structure, where classes, attributes etc., are identified. Fig 3, depicts the proposed IDD class diagram. To building the objects (classes) model, we adopted the goal oriented approach [20]. The principle of this approach is to identify classes from use cases goals rather than use case descriptions with the use case-driven process [21].

In our development, classes, in the class diagram, are the entities that participate in achieving the goals allowing a user to interact with input classes described in the IDD structure file. They have their own features and can collaborate with use cases. The class diagram shows classes and relationships among them. In the example of figure 3, the kinds of relationships are associations. Four classes identify the IDD file structure diagram: a group class (*ObjectsGroup*), an input Object class (*InputObject*), a Numeric (*NumField*) and alphanumeric (*AlphaField*) classes. The two remaining ones, in the diagram, identify an actor/user (module *developer*) and an interface (*ICS*) classes. The association *Associate* shows the relationships between instances of the two classes: ObjectsGroup and InputObject. A multiplicity (1 to 1..*) is added to both ends of the association to point at the role of each class to the other. In the class diagram, an instance of the ObjectsGroup class may possess at least one instance of the InputObject class.

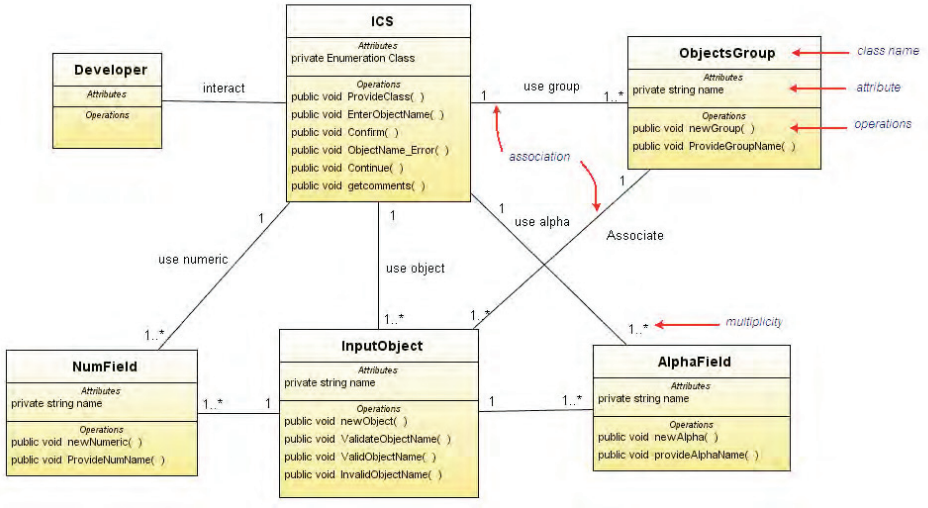

Fig. 3. UML class diagram of the IDD file structure

1.2.3. Sequence diagram

A scenario is an instance of a use case. It describes a group of interactions between actors and objects of the systems. In UML, scenario can be represented in the form of collaboration diagrams and/or sequence diagrams. Both types of diagrams rely on the same underlying semantics, and conversion of one to the other is possible [22].

For our work, we have chosen to use sequence diagram for their simplicity. A sequence diagram shows the interactions among the objects participating in a scenario in temporal order. It depicts the objects by their lifelines and shows the messages they exchange in time sequence. However, it does not capture the associations between objects. A sequence diagram has two dimensions: the vertical dimension represents time, and the horizontal dimension represents the objects. Messages are shown as horizontal solid arrows from the life line of the object sender to the life line of the object receiver.

Fig 4 depicts two sequence diagrams (two scenarios) of the use case *Generate_object*. Fig 4(a) represents a scenario where the developer is correctly entering a valid input object class name (*regularGeneration*), whereas Fig 4(b) shows the case where an invalid input object class name is entered (*errorGeneration*). The two scenarios are based on the input/output reference guide of EnergyPlus (EnergyPlus, v2.0 or latter). Thus, defining an input data entry in the IDD file, the following rules apply:

- A class name must be unique;
- The maximum length for a class name is 100 characters;
- Not an empty name.

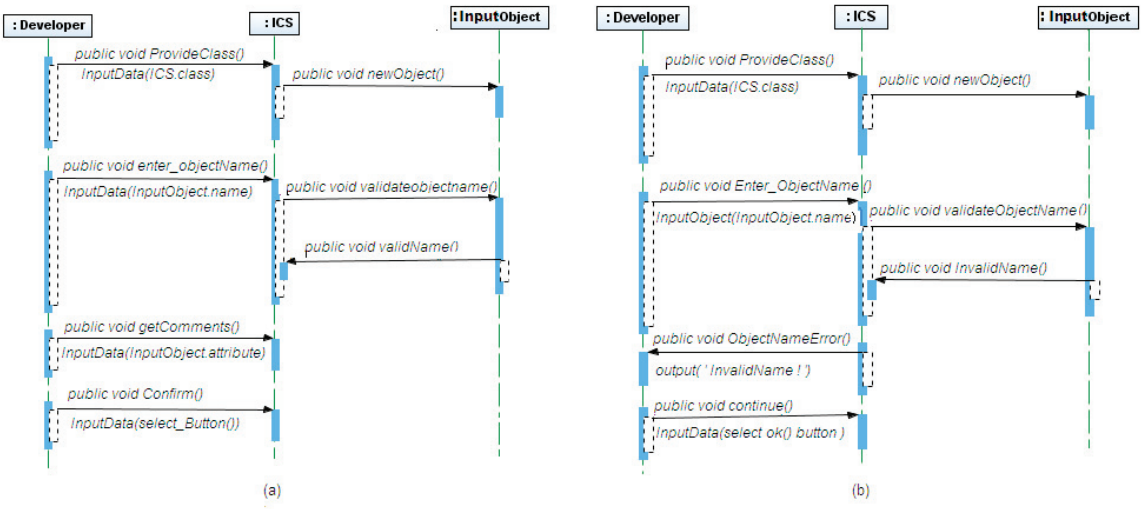

Fig. 4. sequence diagram: (a) regular generation (b) error generation

Also, a message in our work is guarded by a conditional constraint and is enriched with user interface information. Beyond the UML standard message constraints found in sequence diagram [18], two additional constraints *inputData* and *outputData* are defined. The *inputData* constraint indicates that the corresponding message holds input information from the actor. The *outputData* constraint specifies that the corresponding message carries information for display. Both *inputData* and *outputData* constraints have a parameter that indicates the kind of the actor action. It may be either a method's name (a class operation), or one or several class attributes etc. see Fig 4(a) and Fig 4(b) of the underlying class diagram. Once these *inputData* and *outputData* constraints are specified, this information is used to determine the corresponding widgets that will appear in the interface prototype. Widgets generation is based on the terminology and recommendation found in [23] and which include the following eight items:

- (WG_1) An enabled *TextField* widget is generated in case of an *inputData* constraint with a dependency to an attribute of type string, real, integer, e.g. in Fig 3 and Fig 4(a): *EnterObjectName(){inputData(InputObject.name)}*
- (WG_2) A group of Radio Button widgets in case of an inputData constraint with dependency to an attribute of type enumeration having a size less than or equal to 6, e.g. in Fig 3and Fig 4(a): *ProvideClass(){inputData(ICS.class)}*
- (WG_3) An enabled List widget is generated in case of inputData constraint with a size dependent attributes of type enumeration having greater than 6 or with dependent attributes of type collection.

- (WG_4) An enabled Table widget is generated of an inputData constraint with multiple dependent attributes.
- (WG_5) A disabled TextField widget is generated for an output constraint with a dependency to an attribute of type string, real, or integer.
- (WG_6) A label widget is generated for an outputData constraint with no dependent attribute, e.g. in Fig 3and Fig 4(b): *ObjectNameError(){outputData ("incorrect class name !")}*
- (WG7) A disabled List widget is generated in case of outputData constraint with dependent attributes of type enumeration having a size greater than 6 or with dependent attributes of type collection.
- (WG_8) A disabled Table widget is generated of an outputData constraint with multiple dependent attributes.

1.3. Coloured Petri-nets

Coloured Petri-nets (CP-nets) is a modelling language for systems, where synchronisation, communication, concurrency, and resource sharing play a major role. CP-nets are generally used for three different – but closely related purposes:

- A CP-net model is a description of a modelled system and it can be used as a specification or as a presentation.
- The behavioural aspect of a CP-net can be validated by means of a simulation and animation, and can be verified by means of more formal analysis methods, i.e. state spaces and place invariants [2]. The process of creating the description and performing the analysis, usually gives the modeller a dramatically improved understanding of the modelled system.

Coloured Petri-nets are also defined as discrete event modelling language combining Petri-nets (PNs) with a functional programming language. Petri-nets provide the foundation of graphical notation, and the basic primitives for modelling system (control structure, synchronisation, communication, and resource sharing). Data and data manipulation are described by the functional programming language standard ML (SML) [24, 25].

1.3. 1. Basics of Petri-nets

The theory of Petri-nets (PNs) has developed from the work of Carl Adam Petri and many others in the 60'ies and the 70'ies, and they were soon recognised as being one of the most adequate mathematical modelling languages for description and analysis of synchronisation, communication and resource sharing between concurrent processes. Basics of the theory of PNs could be well explained through the following illustrations of Figs 5, 6, and 7. The pictorial representation of a PNs as a graph contains two types of nodes; circles/ellipses (called places) and bars/rectangles (called transitions). These nodes, places and transitions, are connected by directed arcs from places to transitions and from transitions to places. If an arc is directed from node i to node j (either from a place to a transition or a transition to a place), then i is an input to j, and j is an output of i. In Fig 5 for example, place p_1 is an input to transition t_2, while places p_2 and p_3 are outputs of transition t_2.

The execution of PNs is controlled by the position and movement of markers (called tokens) in the net. Tokens indicated by black dots, resides in the places of the net. Tokens are moved by the firing of the transitions of the net. A transition must be enabled in order to fire (a transition is enabled when all of its input places have tokens in them). The transition fires by removing the enabling tokens from their input places and generating new tokens which are deposited in the output places of the transition. In the marked Petri-net of the Fig 5, the transition t_2 is enabled since it has a token in its input place p_1. Transition t_5, on the other hand, is not enabled since one of its input places (p_3) does not have a token. If

t_2 fires, the marked PNs of Fig 6 is obtained. The firing of transition t_2 removes the enabling token from places p_1 and puts tokens in places p_2 and p_3, the outputs of t_2. The distribution of tokens in a marked Petri-net defines the state of the net and is called its marking. The marking may change as a result of the firing of transitions. In different markings, different transitions may be enabled. For example, in the marked PNs of Fig 6, three transitions are enabled: t_1, t_3, and t_5, none of which were enabled in the marking of Fig 5.

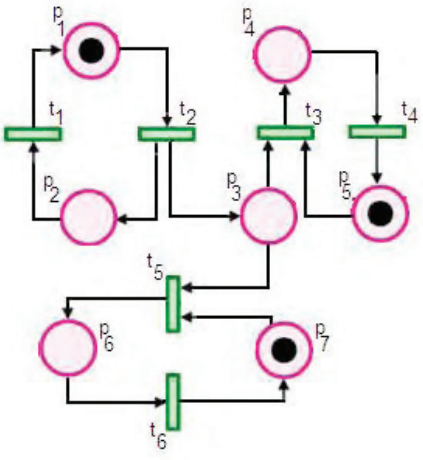

Fig.5. marked Petri-nets: t_2 is enabled

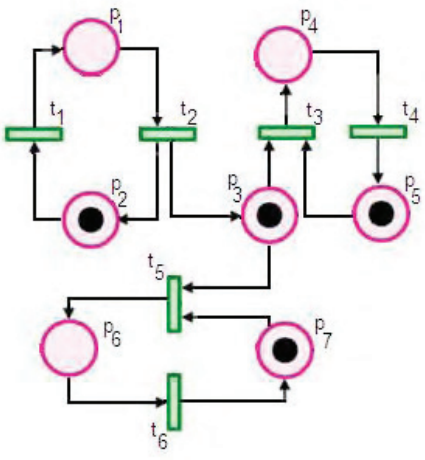

Fig.6. marked Petri-nets: t_2 is not enabled

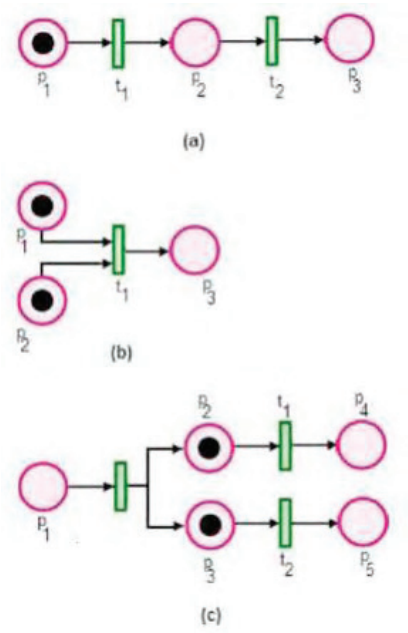

Fig.7. PNs with: (a) sequential exececution, (b) synchronuzation execution, (c) concurrency execution

In addition to the static properties represented by the graph, Petri-nets have dynamic properties that result from its execution. Fig 7, illustrates three useful properties in modelling systems and are briefly discussed in the following:

- Sequential execution: transition t_2 of the Petri-net can fire only after the firing of t_1. This imposes the precedence of constraints: t_2 after t_1, Fig 7(a);
- Synchronization: transition t_1 will be enabled only when there are at least one token at each of its input places, Fig 7(b);
- Concurrency: transitions t_1 and t_2 are executed concurrently in time, Fig 7(c). With this property, Petri-nets are able to model systems of distributed control with multiple processes.

1.3. 2. Standard ML Language (SML)

SML is a modern descend of the ML programming language used in the logic for computable function (LCF) theorem-proving project. Standard ML is mostly functional programming language and the key feature is the function. Inscriptions on Coloured Petri-net models are written in the Coloured Petri-net ML (CPN ML) programming language which is an extension of the standard ML programming language. Types, arc expressions and guards on transitions are specified on CPN ML which is strongly typed. Data types can be atomic (integer, string, real, Boolean, enumeration), and structured (products, records, unions, lists, subsets).

Arc expressions which are the textual inscriptions positioned next to the individual arcs, and are built from typed variables, constants, operators, and functions. When all variables in an expression are bound to values (of the correct type) the expression can be evaluated. The arc expression evaluates to a multi-set of token colours (data values). The example of Fig 8 that depicts the coloured Petri-net(section 2.2.1), an adaptation to CPNTools software [26] of the proposed use case diagram of the Fig 2, consider the arc expression *ae* on the arc connected to both the transition *GenerateObject* and the place *Sc_g*. It contains the variable *ae* declared as:

var ae: l;

where *l* is declared as a colour set of type list,

colset l = list SC; and *SC* is a colour set of type enumeration,

colset SC = with rc|ec;

Next to each place, there is an inscription which determines the set of token colours that places are allowed to have. The set of possible token colours is specified by means of a type and it is called the colour set of the place. By convention, the colour set is written below the place. In Fig 8, places: *Init*, *EndIdentify*, and *EndGenerate* have the colour set *UC* and it is defined in CPN ML programming language to be equal to the type enumeration:

Colset UC = with G|E|S|D;

This means that tokens residing on the three places will have an enumeration type as their token colour. The colour set *UC* is used to model the possible states of the coloured Petri-net. A place and a transition may also be connected by double-headed arcs. A double-headed arc is shortened for two directed arcs in opposite directions between a place and a transition which both have the same arc expression. This implies that the place is both an input place and an output place for the transition. For instance, the transition *IdentifyObject* and the place Init are connected by double-headed arc, Fig 8.

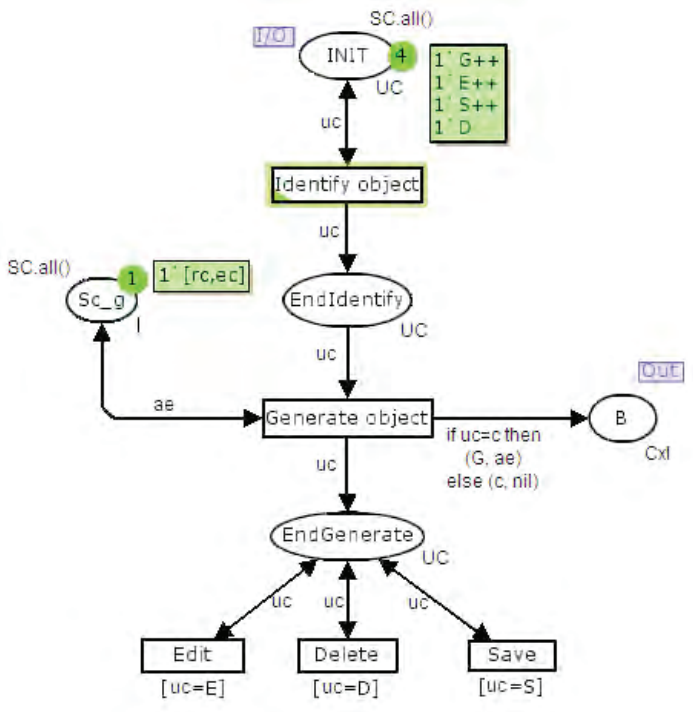

Fig. 8. Coloured PNs of the use case diagram of the IDD file

The current marking of each place is indicated next to the place. The number of tokens on the place in the current marking is shown in the small circle, while the detailed token colours are indicated in the box positioned next to the small circle. Initially, the current marking is equal to the initial marking, denoted M_0. In the example of Fig 8, the initial marking has four tokens on place *Init* and one token on place *Sc_g*.

In this paper, we suggest a new approach based on user requirement techniques, integrating concepts of UML and scenario specifications to generate an IDD interface prototype. It provides a process involving four activities to derive the interface prototype from scenarios and to generate a formal specification of the application. The key feature of this approach lies in:

- The use of scenarios in the sense of UML notation as sequence diagrams;
- The elaboration of a formal specification consisting of a global coloured Petri-net describing the entire behaviour of the interface prototype allowing simulation and testing;
- The derivation of an executable interface prototype which can be used for scenario validation and can be evolved toward the target tool interface.

Section 2 of this paper gives the four activities leading from scenario approach to executable IDD interface prototype. In section 3, some issues, related to the proposed approach, are discussed. Section 4, concludes the paper and provides an outlook of future work

2. Description of the approach

In this section, we describe the process for deriving the IDD interface prototype from scenarios using UML and coloured Petri- nets. We aim to provide two iterative processes: a formal specification process

as illustrated at the top of the Fig 9, and an IDD interface prototyping process shown at the bottom of the Fig 9, and presented in more details in Fig 10.

In this work, we focus on the interface process, that is, the transformation represented by the bold arrows of Fig 9. This process can be decomposed into four activities:

- Scenario acquisition (section 2.1): a class diagram, and use case are elaborated, and for each use case, a number of sequence diagrams annotated with user interface information.
- Generation of specifications (section 2.2): this activity consists of deriving coloured Petri-nets from the acquired use case diagram.
- Scenarios integration (section 2.3): coloured Petri-nets corresponding to the same use case are combined to obtain an integrated coloured Petri-net of the use case.
- IDD interface prototype generation (section 2.4): the integrated coloured Petri-net, in this activity, serves as input for interface prototype process.

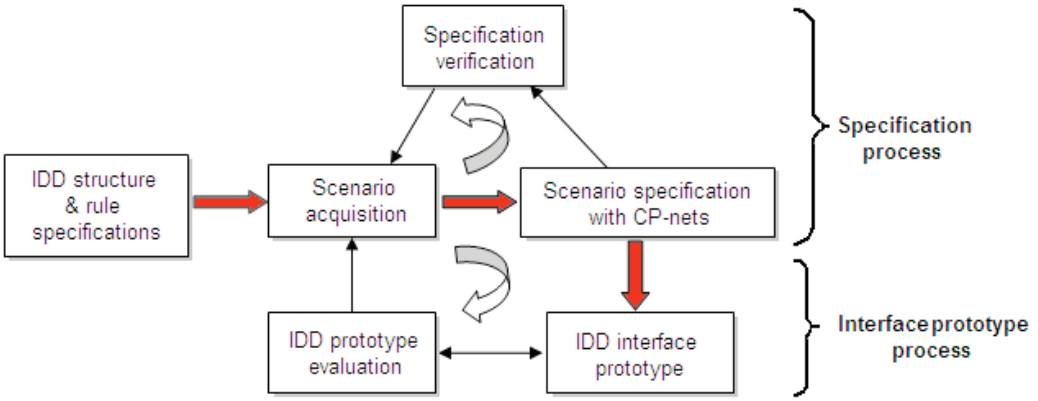

Fig. 9. IDD interface prototype process

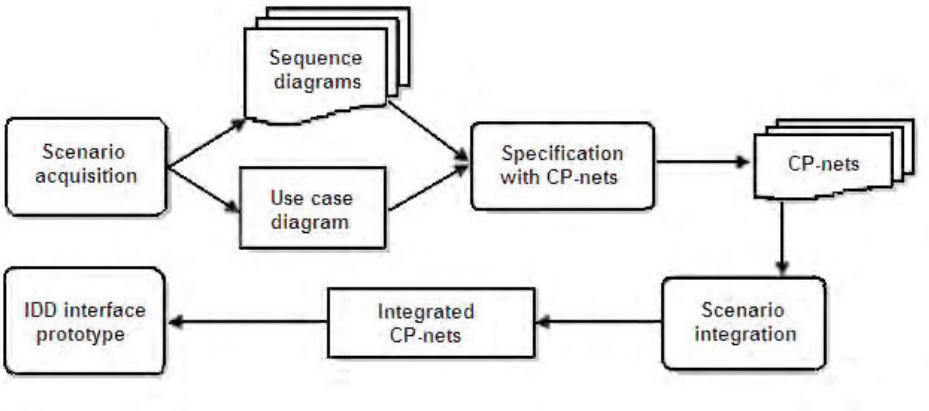

Fig. 10. details of the IDD interface prototype process

2.1. Scenario acquisition

In this activity, a use case diagram is elaborated capturing all the functionalities for a new object (class) data entry into the IDD file. For each use case (figure 2), scenarios are acquired in the form of sequence diagrams. Fig 4(a) and Fig 4(b) illustrate two sequence diagrams (two scenarios) of the use case *Generate_object*. Scenarios of the given use case are classified by type and ordered by frequency of use [27]. In our example, we consider two types of scenarios, normal scenario, which is executed in normal situation, and exceptional scenario of errors. The frequency of execution of a scenario is a number between 1 and 10, assigned to indicate how often a given scenario is likely to occur. For example, the use case *Generate_object* has normal scenario (scenario regularGeneration with frequency 10), and an exceptional scenario (errorGeneration with frequency 5). This classification is used for the composition of interface blocks (section 2.4.4). The generated interface prototype is sensitive to the frequency values of scenarios. For subsequent evaluation, an alternative interface prototype is generated if a scenario frequency is altered.

2.2. Generation of specification

This activity consists of deriving coloured Petri-nets from both the acquired use case diagram (Fig 2) and all sequence diagrams, see Fig 4. These derivations are explained below in the subsequent subsections: use case specification and scenario specification.

2.2.1. Use case specification

The Petri-net corresponding to the use case diagram is derived by mapping use cases into places, see Fig 11. A transition (Enter) leading to one place corresponds to the initiating action (event) of the use case. A place *Init* is added to model the initial state of the IDD interface prototype. After a use case execution, the IDD interface will return, via an (Exit) transition, back to its initial state for further execution of the use case. The place *Init* may contain several tokens to model concurrent executions. Fig 11depicts the Petri-net derived from the use case diagram.

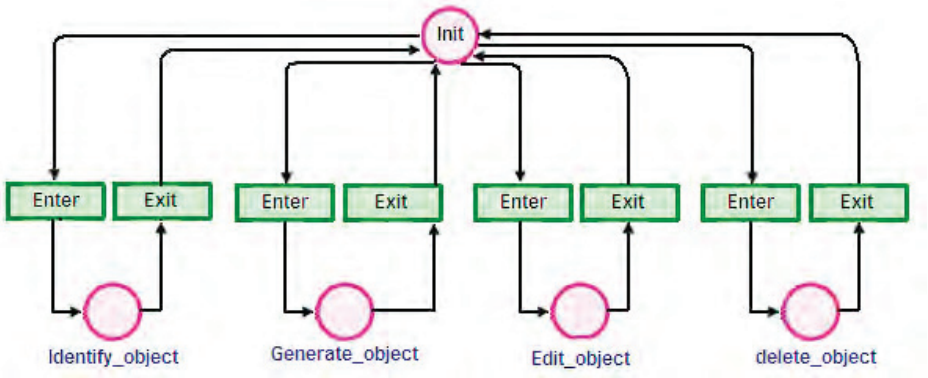

Fig. 11. PNs of the use case diagram specification

In a use case diagram, a use case can call upon the services of another with the relation *include*. The relation may have several meaning depending on the system being modelled. Consider two use cases: uc_1 and uc_2, Fig 12: the relation *include* between them may be interpreted in different ways [12]. Fig 12(a) gives the general form of this relation, uc_1 may be decomposed into three sub use cases: uc_{11} represents the part of uc_1 executed before the call of uc_2; uc_{12} is the part executed concurrently with uc_2 and uc_{13} is the part executed after termination of uc_2 (synchronization). It is possible that two of these three use cases are empty, resulting in one of the configuration types shown in figure 12(b), figure 12(c), figure 12d), figure 12(e), figure 12(f), and figure 12(g). The relation of type (g) between uc_1 and uc_2 means that uc_2 precedes uc_1; this implies that uc_1 is not directly accessible from the initial place (place *Init)*. So, transition from the place *Init* to uc_1 must be changed to transition from uc_2 to uc_1. In the use case diagram (Fig 2), the relation *include* between the following couple of use cases is of type (g):

- (uc_1:Generate_object);
- (uc_2): Identify_object);
- (uc_1: Edit; uc_2: Generate_object);
- (uc_1: Save ; uc_2: Generate_object);
- (uc_1: Delete; uc_2: Generate_object).

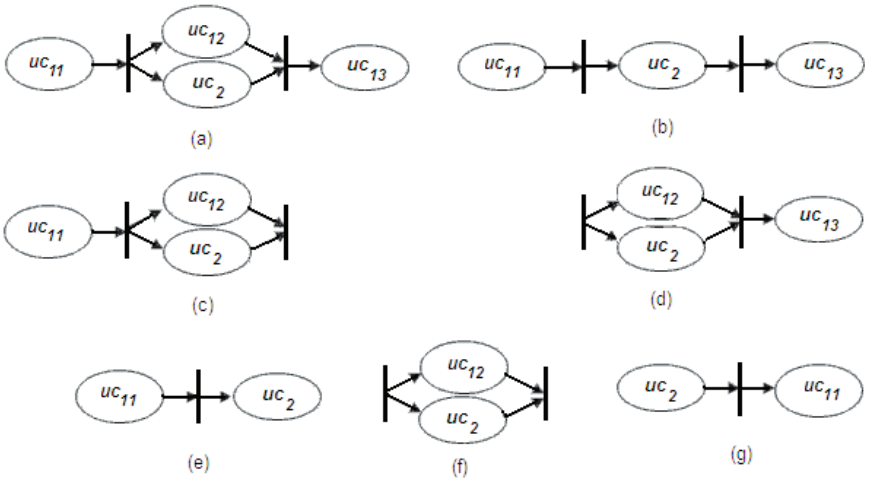

Fig. 12. the include relation and use cases

The CPN Tools software which we adopted in our work, allows for the refinement of transitions, but does not support the refinement of places. Therefore, in order to substitute use cases, which are represented as places, for the coloured Petri-net representing integrated scenarios (section 2.3), the coloured Petri-net obtained after processing the *include* relation requires adaptation: each subnet (*Enter* → $place_i$ → *Exit*) is substituted by a simple transition representing the use case, cf. dashed square in Fig 13, and intermediate places, such as *enIdentify* and *endGenerate* are inserted, see Fig 8.

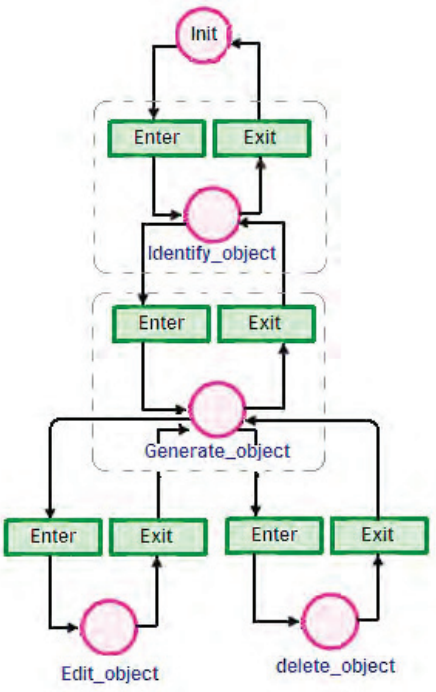

Fig. 12. PNs of the IDD file prototype representing integrated scenarios

2.2.2. Scenario specifications

For each scenario of the use case *Generate_object*, we construct an associated table of object state. This table is directly obtained from the sequence diagram of the scenario by following the exchange of messages from top to bottom, and identifying the changes in object state caused by the messages. In our example, tables 1 and 2 show the object states associated with the scenarios r*egularGeneration*, and *errorGeneration*, Fig 4. In such tables, a scenario state is represented by a state vector of the objects participating in the scenario (column *scenario state* in tables 1 and 2 respectively). From each object state table a coloured Petri-net is generated by mapping scenario states into places and messages into transitions, Figs 13 and 14. Each scenario is assigned a distinct colour, e.g., *rc* for the *regularGeneration* scenario, and *ec* for the *errorGeneration* scenario. All coloured Petri-nets (scenarios) of the same use case will have the same initial place (state) which we call *B,* see Fig 13 and 14. This place will serve to link the integrated coloured Petri-net (section 2.3) with the coloured Petri-net modelling the IDD use case diagram depicted by Fig 8.

Table 1. Object state table associated with the scenario regularGeneration

Object messages	Developer	ICS	Input object	Scenario state
Provide class	Present	Select object	Void	S1= (present, provide class, void)
New object	Present	Select object	Object generated	S2= (present, provide class, object generated)
Enter object name	Present	Prompt name	Object generated	S3= (present, prompt name, object generated)
Validate object name	Present	Name entered	Validate name	S4= (present, name entered, validate name)
Valid name	Present	Name entered	Valid name	S5= (present, name entered, valid name)
Get comments	Present	Get comments	Valid name	S6= (present, get comments, valid name)
Confirm	Present	Confirm	Valid name	S7= (present, confirm, valid name)

Table 2. Object state table associated with the scenario errorGeneration

Object messages	Developer	ICS	Input object	Scenario state
Provide class	Present	Select object	Void	S1= (present, provide class, void)
New object	Present	Select object	Object generated	S2= (present, provide class, object generated)
Enter object name	Present	Prompt name	Object generated	S3= (present, prompt name, object generated)
Validate object name	Present	Name entered	Validate name	S4= (present, name entered, validate name)
Invalid name	Present	Name entered	Invalid name	S8= (present, name entered, invalid name)
Error	Present	Error	Invalid name	S9= (present, error, invalid name)
Continue	Present	Confirm	Invalid name	S10= (present, confirm, invalid name)

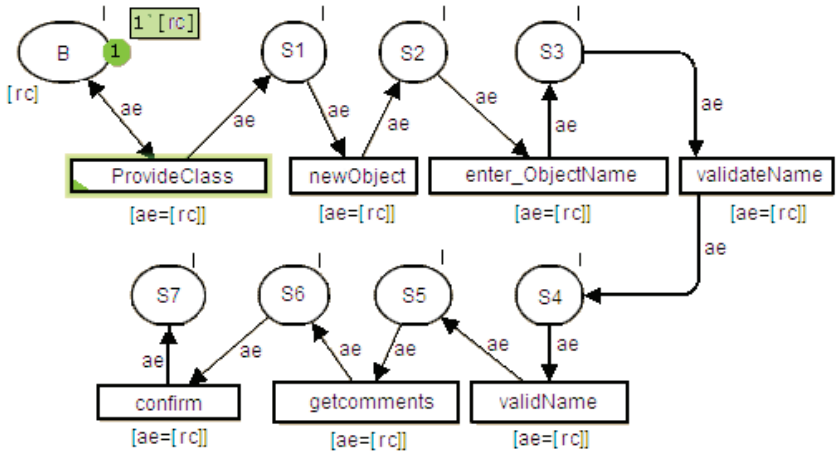

Fig. 13. CPNs of the scenario regularGeneration

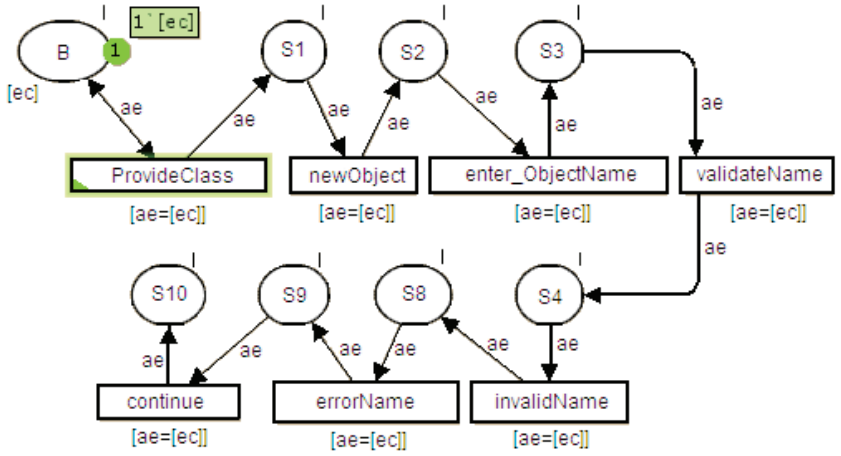

Fig. 14. CPNs of the scenario errorGeneration

2.3. Scenarios integration

In this activity, we aim to combine all coloured Petri-nets corresponding to scenarios of use case uc_i, in order to produce an integrated coloured Petri-net modelling the behaviour of the use case. After integrating the two scenarios, the initial place *B,* Fig 15 will be shared, yet we do not know which scenario will be executed, and neither the token colour *rc* nor the token colour *ec* can be assigned to the initial place *B*. This problem was described by Elkoutbi and Keller [12, 28], and it was referred to it as interleaving problem.

To solve the interleaving problem, we introduce a composite colour set, i.e., a token colour that can take on several colours. Using CPN Tools software [2] a colour set is modelled by a list of colours. Upon visiting the places of the integrated coloured Petri-net, it will be marked by the intersection of its token colours of the place being visited. When the token passes to the place S4, it keeps the colour set [*rc*, *ec*] to S5 its colour changes to [*rc*] and will remain unchanged for the rest of its route, or it passes from the place S4 to the place S8 its colour changes to [*ec*] and will remain unchanged for the rest of its route.

Transitions that belong to only one of the scenario *regularGeneration* or *errorGeneration* will be guarded by the token colour of the respective scenarios, see transitions of Fig 15: [*getcomments, confirm*], and transitions: [*errorName, continue*]. But this is not the case for transitions (*validName* and *invalidName*) which are required to transform colours from the colour set (list of token colours) to a single colour. Therefore they must be guarded by the composite colour set [*rc, ec*]. For transitions that are shared by the two scenarios *regularGeneration* and *errorGeneration*, they will be guarded by the composite colour set.

Integrated coloured Petri-net corresponding to a given use case can be connected to the coloured Petri-net derived from the use case diagram of Fig 8 through a substitution transition *UseCase,* see Fig 16*,* which is appended to the place *B* of the integrated coloured Petri-net. This coloured Petri-net can be organized as a set of hierarchically related modules. This substitution transition will transform tokens of the coloured Petri-net of the use case diagram to the composite colour set of the integrated coloured Petri-net.

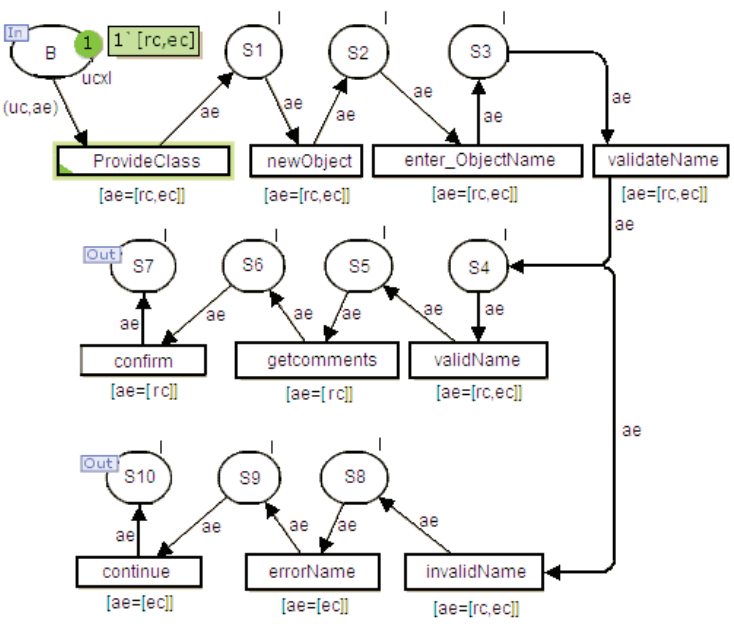

Fig. 15. CPNs of the integrated scenarios: regular generation and errorGeneration

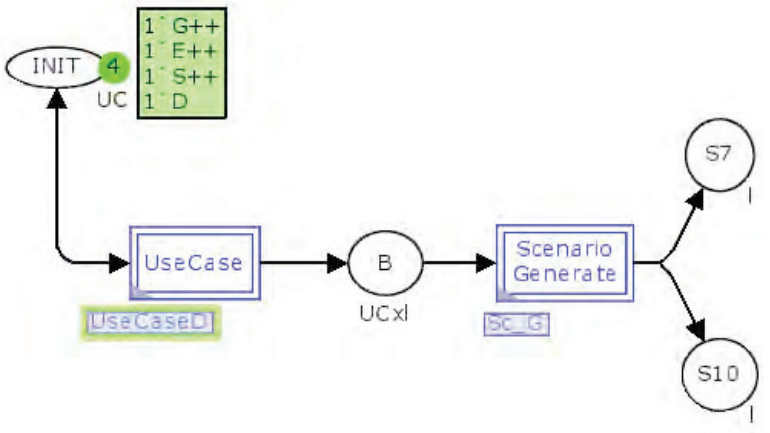

Fig. 16. CPNs of the substitution transition use case

2.4. Interface prototype generation

In this activity, we derive from the coloured Petri-net specification an interface prototype. The generated interface comprises a menu to switch between use cases which are directly accessible from the initial state (place *Init*) of the IDD interface. The various screens of the interface prototype represent the static aspect, the dynamic aspect of the interface Prototype, as captured in the coloured Petri-net specification, maps into the dialog controls of the interface prototype. In our current implementation, prototypes are Java (an OOP language) application comprising each a number of frames and navigation functionalities, Fig 17.

The operation of interface prototype generation is composed of the following steps:

- generating graph of transitions (section 2.4.1);
- masking non interactive transitions (section 2.4.2);
- identifying interface blocks (section 2.4.3);
- composing interface blocks (section 2.4.4);
- generating frames from composed interface blocks (section 2.4.5).

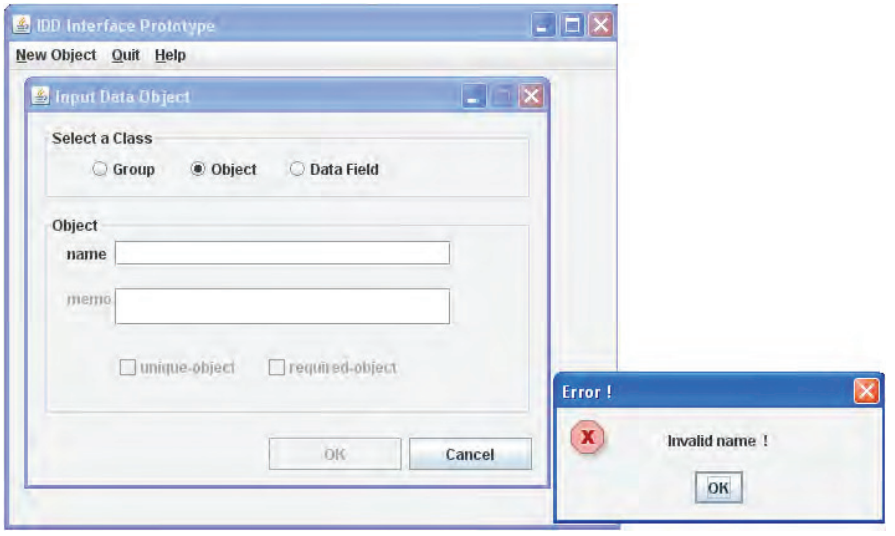

Fig. 17. java implementation of the IDD interface prototype

2.4.1. Generating graph of transition

This operation consists of deriving a directed graph of transitions (GT) from the Coloured Petri-net of a use case *Generate_object*. Transitions of the coloured Petri-net will represent the nodes of the GT, and edges will indicate the precedence of execution between transitions. If transition t_1 precedes transition t_2, we will have an edge between the nodes representing t_1 and t_2.

A GT has a list of nodes nodeList, a list of edges edgeList, and a list of initial nodes initialNodeList (entry nodes for the graph). The list of nodes nodeList of a GT is easily obtained since it corresponds to the transition list of the coloured Petri-net at hand. The list of edges edgeList of a GT is obtained by linking transitions preceding a place ($\rightarrow$ p) with the ones following this place (p$\rightarrow$).

The asterix character (*) is used to make initial nodes in the graph. Nodes of the graph GT are identified as follows:

- t_1 = provideClass;
- t_2 = newObject;
- t_3= EnterObjectName;
- t_4 = validateName;
- t_5 = validName;
- t_6 = getcomments;
- t_7 = confirm;
- t_8 = invalidName;
- t_9 = errorName;
- t_{10} = continue

The graph of transition (GT) of the integrated coloured Petri-net of the use case *Generate_object* is shown in Fig 18(a).

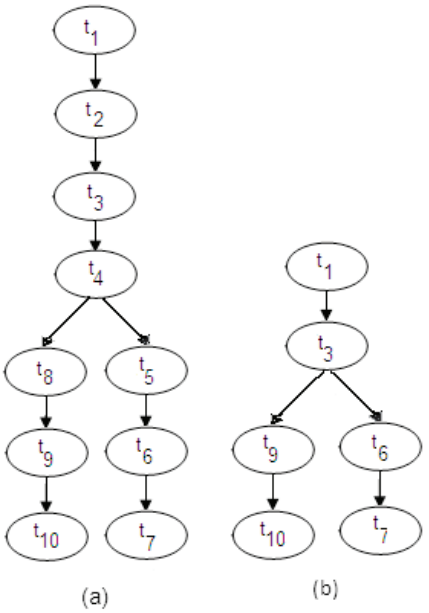

Fig. 18. the use case Generate_object : (a) graph of transition GT; Graph of transition GT'

2.4.2. Masking non interactive transitions

This operation consists of removing all the transitions that do not directly affect the interface (i.e. transitions without interface constraints). These transitions are called non-interactive transitions. All such transitions are removed from the list of nodes nodeList and from the list of initial nodes initialNodeList, and all edges defined by those transitions are removed from edgeList.

When a transition t_i is removed from nodeList, we remove all edges where a transition t_i takes part, and we add new edge in order to bridge the removed transition nodes. If the initialNodeList of initial

transitions contains any non-interactive transitions, they are replaced by their successor nodes. The result of this operation on the graph of transitions of Fig 18(a) is the updated graph of transitions GT' shown in Fig 18(b).

2.4.3. Identifying interface blocks

This operation consists of constructing a directed graph where nodes represent interface blocks (IBs). An IB is a subgraph of GT' consisting of sequence of transition nodes that is characterized by a single input and a single output edge. The beginning and the end of an IB are identified from the graph GT' based on the following block generation rules (BGR_1 to BGR_6) [27]:

- (BGR_1) An initial node of GT' is a beginning of an IB.
- (BGR_2) A node that has more than one input edge is a beginning of an IB.
- (BGR_3) A successor of a node that has more than one output edge is the beginning of an IB.
- (BGR_4) A predecessor of a node that has more than one input edge ends an IB.
- (BGR_5) A node that has more than one output edge ends an IB.
- (BGR_6) A node that has an output edge to an initial node ends an IB.

Applying these rules to the GT' of Fig 18(b), we obtain the graph of interface blocks (GIB) shown on Fig 19(a). In our example, (BGR_1) determines the beginning of IB_1 (t_1) and (BGR_5) the end of the IB_1 (t_3). The interfaces blocks IB_2 and IB_3 are generated by applying (BGR_3) and (BGR_5).

2.4.4. Composing the interface blocks

Generally interface blocks (IBs) obtained from the previous operations, contain only few widgets and represent small parts of the overall use case functionalities. In order to have more interesting blocks that can be transformed into suitable graphic windows, a combination of interface blocks (IBs) is adopted based on the following rules (BGR_7 to BGR_9) [26]:

- (BGR_7) Adjacent IBs belonging to the same scenario are merged (scenario memberships).
- (BGR_8) The operation of composition begins with scenario having the highest frequency (section 2.1).
- (BGR_9) Two IBs can only be grouped if the total of their widgets is less than 20 (ergonomic criterion, according to IBM [23]

Applying these rules to the GIB of figures 19(a) and 19(b) respectively, we obtain the graph GIB' shown in Fig 19(c).

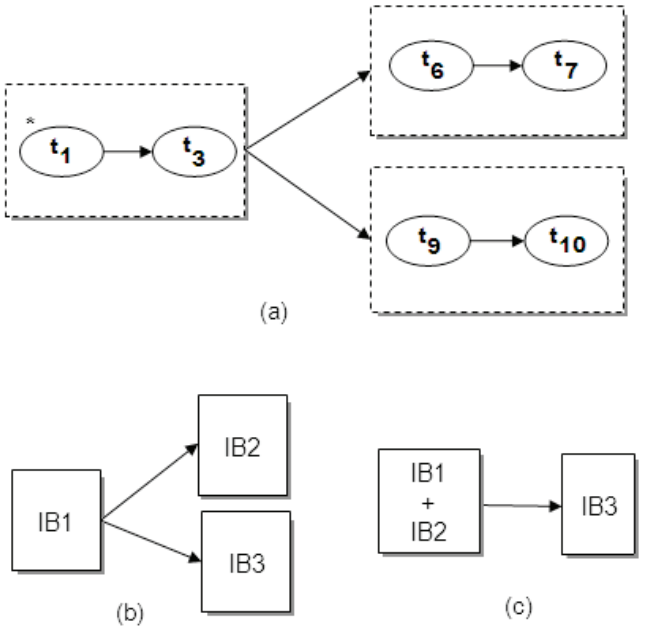

Fig. 19. (a) interface block of the GT; (b) interface block of the GT'; (c) interface block GIB'

2.4.5. Generating frames from composed interface blocks

In this operation, we generate for each IB of a graph of transitions (GT') a graphical frame. The generated frames contain the widgets of all the transitions belonging to the concerned interface block (IB). For the IDD interface prototype, the generated menu is composed uniquely with the use case *Identify_object*, Fig 20. It is the only use case which has a direct access to the initial state (place *Init*) of our coloured Petri-net specification.

The two frames derived from the composed building blocks of Fig 19(c) are shown in Fig 21 and FIg 22. The dynamic aspect of the interface prototype is controlled by the behaviour specification of both graphs of transitions GT and GT'. The execution of the IDD interface prototype relies on the traversal of the graph of transition GT'. The prototype responds to all user interaction events captured by the GT' and ignore all other events.

For example, after selecting the use case *Identify_object* from the main menu of Fig 20 (a top window), another window frame is displayed allowing the user to select between three input class types: a group of related objects, or an input object, or a data field (alphanumeric or numeric). Once an object class is selected, an appropriate display of data entry is enabled in the same frame, Fig 21. When execution reaches a node in GT' from which several continuation paths are possible, the IDD interface prototype displays appropriate dialog box for a proper scenario. The example of Fig 22 corresponds to an error generation scenario of an input object data entry name.

Fig. 20. IDD interface prototype: main frame

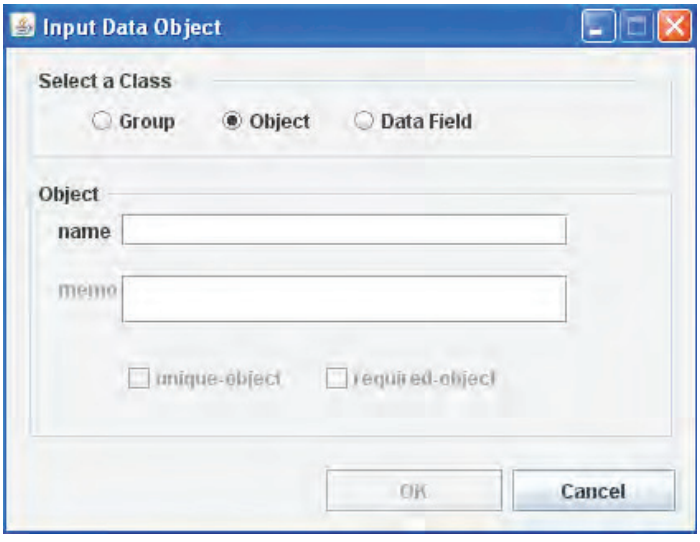

Fig. 21. IDD interface prototype: object selection

Fig. 22. frame for error generation process

3. Discussion of the interface generation approach

Below, we discuss the approach in respect to the following points: 1. scenario-based approach, 2. system and object view, 3. visual formalism for modelling the IDD interface prototype.

3.1. Scenario based approach

The approach to IDD interface generation exhibits the advantages of scenarios based techniques. We put the emphasis on (dynamic) modelling aspect, and generate the dynamic core of the interface rather than focus on screen design. As scenarios are partial descriptions, there is a need to elicit all possible scenarios of the IDD interface to produce a complete specification.

In our work, coloured Petri-nets are used to hold back scenarios interleaving [27]. We defined composition variables (coloured tokens) as described in section 2.3 to prohibits scenarios interleaving, that is, the resultant specifications will capture exactly the input scenario and not more. It is well known [28] that scalability is an inherent problem when dealing with scenarios for large applications. The methodology eases this difficulty by integrating scenarios for each use case, rather than treating them as unstructured mass of scenarios.

3.2. System and Object view

In this paper, we concentrate on using sequence diagrams as scenarios for the restructuring of the formal IDD interface prototype (system view) rather than addressing the specification of individual object , i.e., ICS class, InputObject class, etc…, see Fig 3, in an interactive system (object view).

For the purpose of IDD interface prototype generation, system view is appropriate when in all use cases and associated scenarios; only one actor interacts with the IDD interface. In the case of collaborative tasks (more than one user interacts with the use cases), however, an object view will be more suitable [29].

3.3. Visual formalism for modelling the interactive IDD interface prototype

Coloured Petri-nets are among the most powerful visual formalism used for specifying complex and interactive system. Coloured Petri-nets are known for their support of pure concurrency, all transitions having sufficient number of tokens in their input places may concurrently be fired. Coloured Petri-nets in their basic form do not support hierarchy, but the extension of coloured Petri-nets used in tools such as CPNTools software allows for hierarchies in the specification [24]. Non-deterministic choices can be more easily modelled using coloured Petri-nets. When two or more transitions share the same input places, the system (CPNTools) chooses randomly to fire one of these transitions (section 2.3). Tokens that are specific to coloured Petri-nets can be used both in controlling and simulating the IDD interface behaviour and in modelling data and resources of that system. If the place *Init* of the Fig 8 contains only one token, the IDD interface system can just execute one use case at a time. When the place *Init* contains n tokens, n concurrent executions of different use cases are possible. It may be possible to execute n scenarios of the same use case (multiple instances).

4. Conclusion and future work

The work presented in this paper proposes an approach to the generation of an IDD interface prototype from scenarios. Scenarios are acquired as sequence diagrams according to UML notation, and enriched

with user interface information. These sequence diagrams are transformed into coloured Petri-net specifications, from which an interface prototype of the IDD file is generated. Both the static and the dynamic aspects of the interface prototype are derived from the coloured Petri-net specifications.

The most interesting features of this approach lie in the use of the scenario approach addressing not only sequential scenarios but also scenarios in the sense of the UML (which support concurrency in scenarios), and in the derivation of executable interface prototype which can be used for scenario validation, and can be evolved towards the target IDD interface application.

As future work, we plan to pursue further development in particular the automation of the scenarios integration activity by introducing an algorithm which takes an incremental approach to integration. Given two scenarios with corresponding coloured Petri-nets (CP-net_1 and CP-net_2); the algorithm will merge all places in CP-net_1 and CP-net_2 having the same names. The merged places will have as token colours the *union* of token colours of the two scenarios. Then, the algorithm looks for transitions having the same input and output places in the two coloured Petri-nets and merges them to obtain the integrated coloured Petri-net.

References

[1] Potts, C., Takahashi, K., & Anton, A. Inquiry-Based Scenario Analysis of System Requirements. Technical Report GIT-CC-94/14, Georgia Institute of Technology. 1994

[2] Jensen, K., & Kristensen, L.M. Coloured Petri-nets: Modelling and Validation of Concurrent Systems.Springer-Verlag,Textbookin preparation. Available from: http://wiki.daimi.au.dk/cpntools/cpntoolssttt.wwik.

[3] Crawley, D.B., Lawrie, L.K., Winkelmann, F.C., Buhl, W.F., Erdem, A.E., Huang, Y.J., Pedersen, C.O., Liesen, R.J., Fisher, D.E.,Strand, R.K., & Taylor, R.D. EnergyPlus: a New Generation Building Energy Simulation Program. Paper presented at the 6th International IBPSA Conference, Kyoto, Japan. 1999

[4] Crawley, D.B., Winkelmann, F.C., Laurie, L.K., and Pedersen, C.O. EnergyPlus: New capabilities in a Whole Building Energy Simulation Program. Paper presented at the 7th International IBPSA Conference, Rio de Janeiro, Brasil, 2001.

[5] Fischer, D.E., Taylor, R.D., Buhl, F., Liesen, R.J., and Strand, R.K. A Modular Loop-Based Approach to HVAC Energy Simulation and its Implementation in EnergyPlus. Paper presented at the 6th International IBPSA Conference, Kyoto, Japan. 1999

[6] Olsen, E.L. Performance comparison of UK Low Energy Cooling Systems by Energy Simulation (Master of Science, Massachusetts Institute of Technology). 2002

[7]Edwin, O.S, Van Djik, L.R., Peter, G.L., and Prof, I.R. An architect Friendly Interface for a Dynamic Building Simulation Program. Paper presented at the Sustainable Building Conference. 2002.

[8] Barry, O.S., and Marcus, K. Specification of an IFC Based Intelligent Graphical user Interface to support Building Energy Simulation. Paper presented at the 9th IBPSA Conference, Montreal, Canada. 2005

[9] Junjie, L., Wenshen, L., and Xianjie, Z. The Study of the Simple Hvac Interface of EnergyPlus in Chinese. Paper presented at the Building simulation Conference, Beijing, China. 2007.

[10] Shady, A., Liliana, B., De Herde, A., and Jan, J. Architect Friendly: A Comparison of Ten Different Building Performance Simulation Tools. Paper presented at the 11th International IBPSA Conference, Glasgow, Scotland. 2009

[11] Rolland C., Ben Achour C., Cauvet C., Ralyté J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Dubois P. K., and Heymans, P. A Proposal for a Scenario Classification framework. In Requirements Engineering Journal, Vol.3, No.1, 1998; p.23- 47.

[12] Elkoutbi, M., & Keller, R. K. Modelling Interactive Systems with Hierarchical Coloured Petri-nets. Paper presented at Adv. Simulation Technologies Conference, Boston MA, USA, Soc. for Comp. Simulation Intl. HPC98, Special session on Petri-Nets, 1998; 432-437.

[13] Lee, W.J., and Kwon Y.R. Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering. IEEE Transactions on Software Engineering, Vol.25, No.12, 1998 ; p. 1115-1130.

[15] Potts C. Using Schematic Scenarios to Understand User Needs. In Proceedings of the Symposium on Designing Interactive Systems, Michigan USA, 1995; p.247-256.

[16] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. Object-Oriented Modelling and Design. Prentice–Hall; Inc. 1991.

[17] Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., & Chen, C. Formal Approach to Scenario Analysis. IEEE Software, 2(11), 1994; p.33-41.

[18] Rumbaugh, J., Jacobson, I., & Booch, G. The Unified Modelling Language Reference Manual. Addison Wesley. 1999.

[19] Bachkhaznadji A., Belhamri. A. A Model Driven Application for HVAC Energy Simulation Data: EnerMDA. Paper presented at the3rdCongress of HVAC Engineering, Climamed, Lyon, France, 2006; p. 563- 572.

[20] Liang, YFrom Use Case to Classes: A Way of Building Object Model with UML. Journal of Information and Software Technology, Volume 45,2003; p. 83-93.

[21] Graham, I. Use Cases Combined with Booch/OMT/UML, Process and Product. Journal of Object Programming, 1995; p.76-78.

[22] Glinz, M. Statecharts for Requirements Specifications-As simple as Possible, as Rich as Needed. Position Paper in the ICSE 2002 Workshop: Scenarios and state machine models, algorithms, and tools, Orlando, Florida, USA. 2002.

[23] IBM (1991). Systems Application Architecture: Common User Access-Guide to User Interface Design-Advanced Interface Design Reference, IBM.

[24] Jensen, K. Coloured Petri-nets, Basic Concepts, Analysis Methods and Practical use. Volume3: Practical use, Springer-Verlag. 1997.

[25] Kristensen, L.M., Jorgensen, J.B., & Jensen, K. Application of Coloured Petri-nets in System Development. In Lectures on Concurrency and Petri-nets – Advances in Petri-nets. Paper presented at the 4th Advanced Course on Petri-nets. Volume 3098, 2004; p. 626-685, Springer-Verlag

[26] CPNTools version 2.2.0. Department of Computer Science, University of Aarhus 2006,. Available from http:// wiki.diami.au.dk/cpntools/.

[27] Elkoutbi, M., Khriss, I., & Keller, R. K. (). Automated Prototyping of User Interface Based on UML Scenarios. Journal of Automated Engineering, 2006; p.5-40.

[28] Elkoutbi, M., & Keller, R. K. User Interface Prototyping based on UML scenarios and High-level Petri-Nets. Paper presented at the 21st International Conference on ATPN, Springer-Verlag LNCS 1825, Aarhur, Denmark, 2000; p.166-186.

[29] Khriss, I., Elkoutbi, M., Keller, K.M. Automating the Synthesis of UML statechart Diagrams from Multiple Collaboration diagrams. Paper presented at the International Workshop on the Unified Modelling Language UML'98. Beyond the Notation, Mulhouse, France, 1998; p. 115-126.