An Eclipse Plug-in to Model System Requirements and Adaptation Capabilities

Luciano Baresi and Liliana Pasquale

Politecnico di Milano - Dipartimento di Elettronica e Informazione
p.zza L. da Vinci, 32, 20133 - Milano (Italy)
{baresi|pasquale}@elet.polimi.it

Abstract. Requirements elicitation is an important phase of the software life-cycle, since it helps to reduce the software development time and costs. Unfortunately, existing tools for software design are mainly focused on building a "solution" (i.e., the software system), and neglect the elicitation and analysis of the stakeholders' requirements. Besides, systems are constantly required to adapt to cope with the variability of the environment in which they operate. For this reason, self-adaptation must be also taken into account during requirements elicitation.This paper proposes a graphical designer to express the conventional (functional and non functional) requirements together with the adaptation capabilities of the system. It has been developed as an Eclipse plug-in and leverages other Eclipse projects, such as EMF [1], GMP [2] and Xtext [3], which helped us to make the designer usable and extensible.

1 Introduction

Requirements elicitation is a delicate phase in the software development process. If requirements are not captured completely and unambiguously, IT projects would experiment delivery delays and overrun costs. For this reason, it is fundamental to provide requirements engineering (RE) tools to enable the software designer to express and manage the requirements after their definition. Unfortunately, existing tools are mainly focused on aiding the designers to build a "solution" (i.e., the software system), and neglect a rigorous analysis of the requirements of the stakeholders.

Besides, the RE process would not be effective enough if the designer just takes into account the "conventional" functional and non functional requirements. As a matter of facts, many software systems need to self-adapt to cope with the variability of the environment in which they operate. Variability may be due to requirements violations, context changes, or modifications of the business objectives of the stakeholders. For this reason, self-adaptation becomes a requirement "per-se" and must be related to the other conventional requirements of the system.

This paper proposes a graphical tool, the FLAGS Designer, to aid software designers to elicit and represent the requirements of self-adaptive systems. Our tool is based on the FLAGS (Fuzzy Live Adaptation Goals for Self-Adaptive

Systems) [4] modeling notation. FLAGS generalizes the existing KAOS [5] model, adds *adaptation goals* to embed adaptation countermeasures, and fosters self-adaptation by considering requirements as *live*, runtime entities. The adoption of a goal model allows us to represent the requirements in a hierarchical way, by refining high-level goals into system requirements and operations. Adaptation goals embed the adaptation actions (i.e., changes in the model) to be performed if one or more goals are not fulfilled satisfactorily. The execution of an adaptation goal at runtime depends on the satisfaction level of related goals and the actual conditions of the system and of the environment. Finally, to pursue the alignment between requirements and the running systems, and delegate decisions about adaptation to the goal model, FLAGS proposes to turn goals into live entities, as suggested by Kramer and Magee [6].

Requirements modeling has been always tedious, since the size of the model can grow easily, making it unmanageable. For this reason, FLAGS divides the requirements model in different views (domain, goals, operations, and adaptation). Each view may contain a portion of the model to make it more understandable and reduce its complexity. FLAGS also proposes a suitable language that specifies goals as fuzzy constraints that quantify the degree x ($x \in [0, 1]$) to which a goal is satisfied/violated. As for goals' satisfaction, a crisp notion (yes or no) would not provide the flexibility necessary in systems where some goals cannot be clearly measured (soft goals), and small/transient violations must be tolerated.

The FLAGS Designer supports the creation of new instances of the FLAGS model and provides some preliminary mechanisms to convert the elements of the model into runtime entities. It provides a nice and intuitive graphical interface that makes possible for non-practitioners to create and manage FLAGS models with a minimal effort. Finally the FLAGS Designer offers suitable mechanisms to represent requirements at runtime and assess their satisfaction. A first prototype of the FLAGS Designer has been developed as an Eclipse plug-in[1] and distributed under GNU GPL 3 license. It leverages other Eclipse projects, such as EMF (Eclipse Modeling Framework) [1], GMP (Graphical Modeling Project) [2] and Xtext [3], which helped us to make the designer usable and extensible. The first release of the FLAGS designer has been successfully employed to model the requirements of real cases study in the health care domain. In particular it has been used to model a set of safety-critical surgical actions (i.e., puncturing, suturing, cutting) for the ISUR European Joint project [7].

The rest of the paper is structured as follows. Section 2 motivates our design choices. Section 3 illustrates how the FLAGS Designer supports the creation of new instances of the FLAGS model through a running example. Section 4 discusses some related work. Section 5 concludes the paper.

2 The Overall Solution

The FLAGS Designer has ben created to satisfy a certain set of requirements. It must support the creation and management of new or existing instances of the

[1] The graphical designer is publicly available at http://code.google.com/p/flags/.

FLAGS model, including their views. These model instances must comply with a set of consistency rules (e.g., a goal cannot refine itself), while the definition of goals must conform to the FLAGS language. Since golas are conceived as runtime entities it is also necessary to trace them into live objects and assess their satisfaction at runtime. Besides, we require that both domain experts and non-practitioners be able to define new instances of the FLAGS model with the minimal effort. Finally, requirements models and notations are in continuous evolution, and, for this reason, the FLAGS Designer must be extensible.

The FLAGS Designer has been released as an Eclipse plug-in. A plug-in is a module that may be used by other plug-ins, or be combined with other plug-ins on the same Eclipse platform, or be extended by other plug-ins. This choice supports the reusability and extensibility of the FLAGS Designer and allows us to exploit other plug-ins (EMF, GMP, and Xtext) to reduce the release time of our solution. We also decided to develop the FLAGS Designer according to the MDA (Model Driven Architecture) principles to easily turn goals into runtime entities. In particular, we defined the FLAGS model through EMF, which enables the automatic translation of the model into correct and easily customizable Java code. EMF also allows us to annotate the model with suitable OCL (Object Constraint Language) [8] constraints to forbid the creation of instances of the FLAGS model that violate its consistency rules.

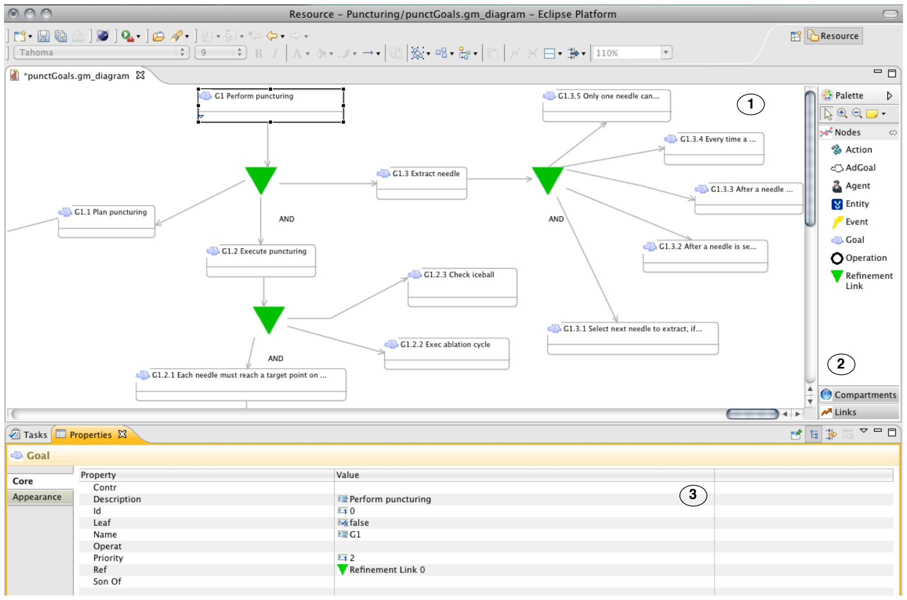
GMP greatly helped us to develop a user-friendly graphical interface. GMP provides a set of generative components and runtime infrastructures for developing graphical editors based on EMF. One can generate a graphical editor, by simply creating a tooling, graphical and mapping model definition. The interface of the FLAGS Designer is intuitive: the elements of the model can be created by drag and dropping them from a palette, while links can be added by connecting two elements in the model. Finally we used Xtext to define the syntax of the FLAGS language. Xtext allows us to automatically derive a parser that recognizes a FLAGS formula and generates its corresponding AST (Abstract Syntax Tree). We modified the visit algorithm of the AST to derive the monitors that check the satisfaction of the corresponding goal at runtime.

3 The FLAGS model

This section describes how the FLAGS Designer supports the creation of new instances of the FLAGS model, including their different views (i.e., domain, goals, operations and adaptations). The domain view represents the environment in which the system operates. The goals view expresses the requirements of the system and links them to the operations necessary for their achievement. The operation view represents the operations of the system. The adaptation view describes the adaptations that must be performed if some conditions are satisfied (e.g., goals are violated, certain events takes place).

As presented in Figure 1, the graphical interface of the FLAGS Designer is composed of three areas. The Diagram (1) shows one of the view the designer is currently creating, the Palette (2) contains the elements (Nodes, Links, and

Compartments) that can be drag and dropped in the diagram, and the Properties View (3) allows the designer to visualize and edit the properties of an element, which has been selected in the diagram. Note that the elements of the model can be divided in three categories: Node, Compartments and Links. Node are the main elements of the FLAGS model, such as goals, adaptation goals, operations, and so forth. Links represent all possible connections between the elements of the model (e.g., an operationalization is a link between a goal and an operation). The Compartments correspond to other additional properties of an element of the model (e.g., the definition of a goal).



Property	Value
Contr	
Description	Perform puncturing
Id	0
Leaf	false
Name	G1
Operat	
Priority	2
Ref	Refinement Link 0
Son Of	

Fig. 1. The Goal model for the puncturing case study.

In the following we describe how the FLAGS Designer supports the creation of all different views of a model. As an example, we refer to the FLAGS model we used to represent the puncturing in the percutaneous cryoablation. This surgical action is used to destroy small cancers located on the surface of the kidney and comprehends three phases. First, one or more needles must be inserted in the tumoral mass, then an iceball must be created by injecting hydrogen in the cancer through the needles, and, finally, all needles must be extracted after the iceball is defrosted.

3.1 The Domain View

The elements of the domain may belong to one the following types: entity, event, or agent. Both entities and events can be characterized by simple attributes (e.g., int, string) or may refer to other entities/events already defined in the model. Attributes and references also have a cardinality (e.g., at most one, exactly one, some, any). Entities are objects providing an informative content and may evolve through a set of states, when one or more operations are performed. The state of an entity is determined by the value assumed by its attributes. Events are instantaneous objects, corresponding to something that may happen during the execution of the system.

Agents can be software components, external devices and humans in the environment, responsible for the satisfaction of some goals. Agents can monitor and control entities and events in the environment and may be responsible for the realization of a goal, depending on the objects they can monitor/control.

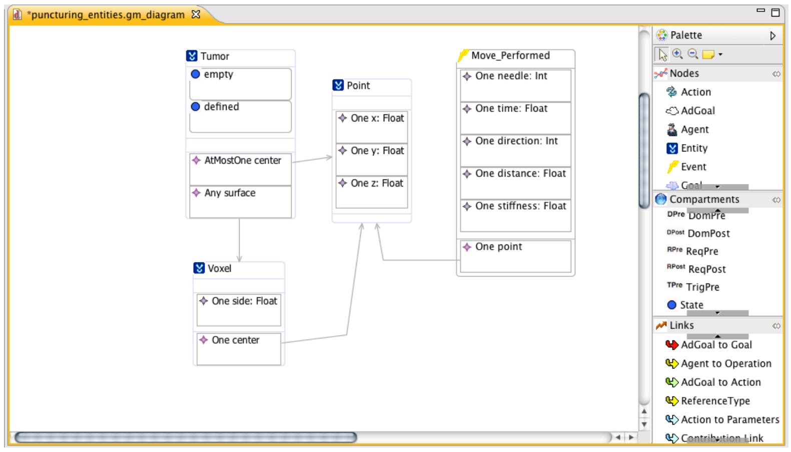


Fig. 2. A subset of the entities of the domain of the puncturing case study.

Figure 2 provides a subset of the domain model for the puncturing case study. From Figure 2 we can see event *Move_Performed* and entities *Tumor*, *Point* and *Voxel*. In our example, we did not model agents since our main focus is on automating a surgical action, instead of reasoning on the responsibility of different agents.

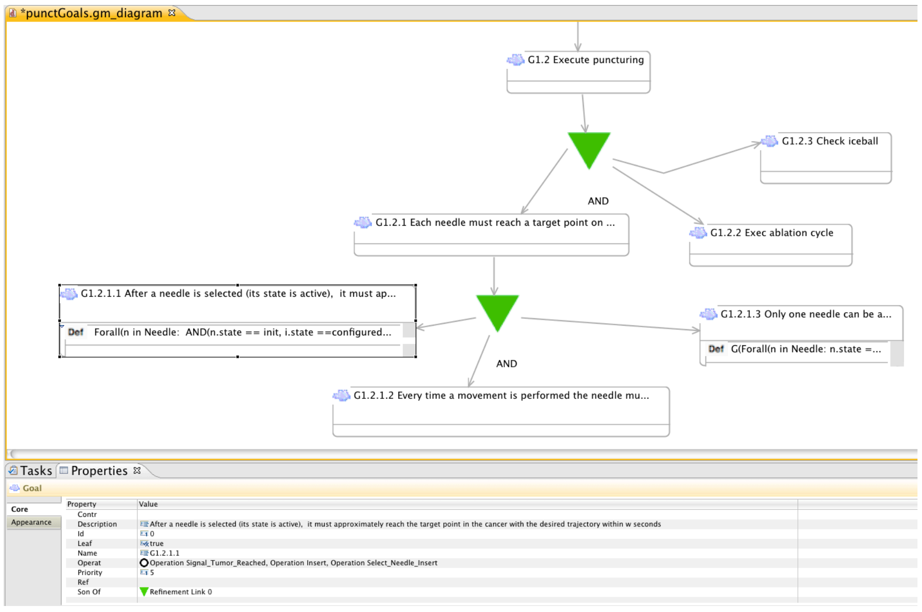
Entities are characterized by three groups of compartments: the first one identifies the states through which an entity can transit, the second one detects the attributes, and the last one contains the references to other entities/events in the model. Since events do not have a state, they are just characterized by the last two groups of compartments.

Event *Move_Performed* is generated every time a needle moves, either because it is being inserted or it is being extracted. It is described by a set of attributes: the needle that has been moved (*needle*), the *time* at which the movement took place, the *direction* of the movement (-1 if the needle is inserted or 1 if the needle is extracted), the distance covered by the needle, and the *stiff-*

ness perceived during the movement. Event *Move_Performed* has also reference *point* which indicates the arrival point of the movement. Entity *Point* is just characterized by its coordinates (x, y, and z) and does not have a state. A *Tumor* is characterized at most by a center (reference to a *Point*) and a surface (i.e., a set of *Voxel*[2]). Entity *Tumor* may pass through state *empty*, when its surface and center has not been detected yet, and *defined*, when its surface and center is detected through an ultrasound scan.

3.2 The Goal view

The Goal view defines the main objectives the system should meet. Goals can be refined into several conjoined subgoals (AND-refinement) or into alternative combinations of subgoals (OR-refinement). The satisfaction of the parent goal depends on the achievement of all (for AND-refinement) or at least one (for OR-refinement) of its underlying subgoals. The refinement of a goal terminates when it can be "operationalized", or , in other words, it can be associated with a set of operations. All goals that cannot be refined anymore are called leaf goals and represent the requirements of the system. Goals are formally expressed in terms of crisp or fuzzy properties and are associated with a priority depending on their criticality.

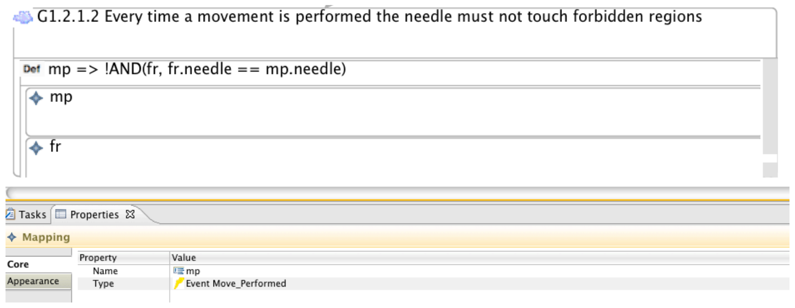


*punctGoals.gm_diagram
G1.2 Execute puncturing
G1.2.3 Check iceball
AND
G1.2.1 Each needle must reach a target point on ...
G1.2.2 Exec ablation cycle
G1.2.1.1 After a needle is selected (its state is active), it must ap...
Def Forall(n in Needle: AND(n.state == init, i.state ==configured...
G1.2.1.3 Only one needle can be a...
Def G(Forall(n in Needle: n.state =...
AND
G1.2.1.2 Every time a movement is performed the needle mu...
Tasks
Properties
Goal
Core
Appearance

Property	Value
Contr	
Description	After a needle is selected (its state is active), it must approximately reach the target point in the cancer with the desired trajectory within w seconds
Id	0
Leaf	true
Name	G1.2.1.1
Operat	Operation Signal_Tumor_Reached, Operation Insert, Operation Select_Needle_Insert
Priority	5
Ref	
Son Of	Refinement Link 0

Fig. 3. Decomposition of goal G1.2 for the puncturing case study.

[2] A voxel is a volume element, representing a value on a regular grid in three dimensional space

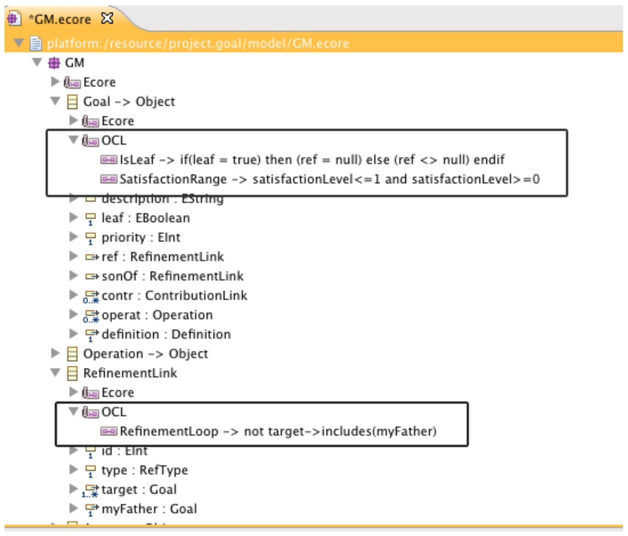


```
G1.2.1.2 Every time a movement is performed the needle must not touch forbidden regions
Def mp => !AND(fr, fr.needle == mp.needle)
  mp
  fr
```

```
Tasks  Properties
Mapping
Core         Property   Value
Appearance   Name       mp
             Type       Event Move_Performed
```

Fig. 4. The definition of goal G1.2.1.2

Figure 3 partially shows the goal view of the puncturing, where each goals is characterized, on top, by its name and description. In particular, it shows goal G1.2 (Execute puncturing) that can be decomposed (AND-refinement) into 3 sub-goals. First, all necessary needles must reach a specific position in the tumoral mass (G1.2.1). When all needles have been inserted, the ablation cycle must be performed (G1.2.2). After an iceball has been created, the doctor must verify whether the tumor has been completely destroyed (G1.2.3). In its turn, goal G1.2.1 can be further AND-decomposed into goals G1.2.1.1, G1.2.1.2 and G1.2.1.3. G1.2.1.1 specifies that all the needles must be inserted in the tumoral mass. G1.2.1.2 asserts that every time a movement is performed the needle must not touch forbidden regions, such as ribs or nerves. G1.2.1.3 states that only one needle can be active at each time instant.



```
*GM.ecore
platform:/resource/project.goal/model/GM.ecore
  GM
    Ecore
    Goal -> Object
      Ecore
      OCL
        IsLeaf -> if(leaf = true) then (ref = null) else (ref <> null) endif
        SatisfactionRange -> satisfactionLevel<=1 and satisfactionLevel>=0
      description : EString
      leaf : EBoolean
      priority : EInt
      ref : RefinementLink
      sonOf : RefinementLink
      contr : ContributionLink
      operat : Operation
      definition : Definition
    Operation -> Object
    RefinementLink
      Ecore
      OCL
        RefinementLoop -> not target->includes(myFather)
      id : EInt
      type : RefType
      target : Goal
      myFather : Goal
```

Fig. 5. OCL constraints defined on the FLAGS model.

The Properties view allows the users to configure other additional attributes of a selected goal, such as its description, priority, whether it is a leaf goal or not, or its operations (in case of a leaf goal). For example, Figure 3 shows the properties view of G1.2.1.1, which is a leaf goal (attribute *leaf* is true) and has the maximum priority (i.e., 5) since it is one of the most important goals to achieve. The operations associated with G1.2.1.1 are *Insert*, *Signal_Needle_Extracted*, and *Select_Needle*, which we will explain in detail in the following subsection.

Each leaf goal has one compartment composed of some sub-compartments. The first sub-compartment contains the definition the goal, which is specified in the FLAGS language and may use some variables that refer to other elements of the FLAGS model. For example, Figure 4 shows the definition of goal G1.2.1.2, which asserts that every time a movement is performed (variable *mp*), no forbidden region must be touched (variable *fr*) by that needle. The other sub-compartments represent the mapping between the variables used in the definition and the elements of the FLAGS model. For example, variables *mp* and *fr* refer respectively to event *Move_Performed* (as shown in Figure 4), and event *FR_TouchedEv*.

Figure 5 presents some OCL constraints specified for the FLAGS model. For example, in the first group we assert that a leaf goal cannot be refined by other goals (*isLeaf*), while the second constraint asserts that a goal cannot refine itself (i.e., it cannot be in both attributes *myFather* and *target* of a refinement link).

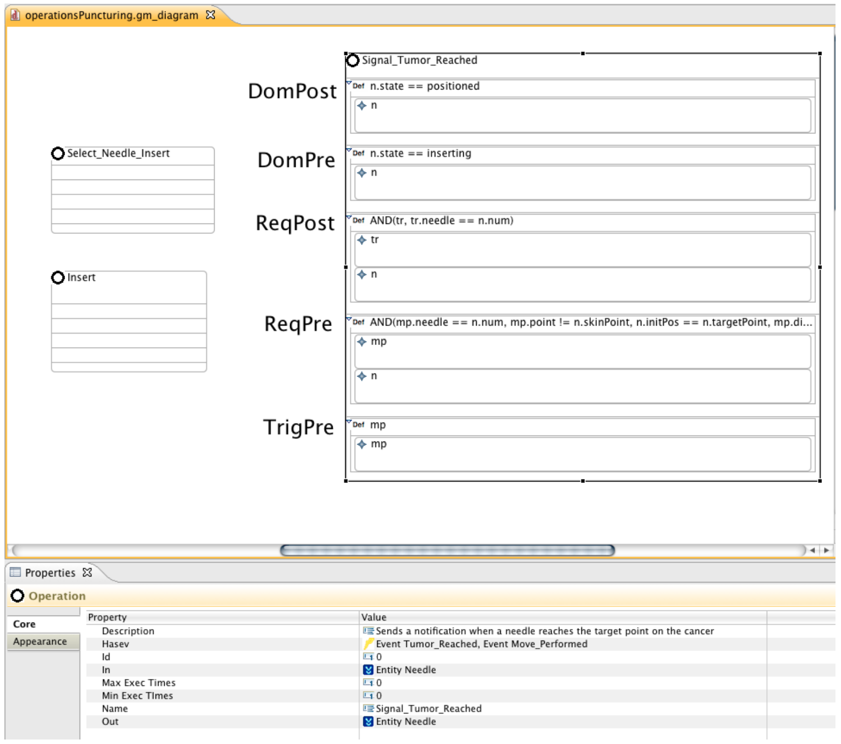
3.3 The Operation view

The Operation view describes the operations of the system. An operation is an input-output relationship over a set of objects. Operations are specified depending on their effects on the domain: domain pre- and post-conditions (*DomPre* and *DomPost*). A domain pre-condition characterizes the state before applying the operation, a domain post-condition defines a relationship between the state before and after applying the operation. Operations are also specified through required pre-conditions (*ReqPre*), triggering pre-conditions (*TrigPre*) and required post-conditions (*ReqPost*). Required pre-conditions define those states in which the operation is allowed to be applied. Triggering conditions define those states in which the operation must be immediately applied, provided the domain precondition is true. Required post-conditions define additional conditions the application of an operation must satisfy.

Figure 6 represents some operations of the FLAGS model of the puncturing: *Select_Needle_Insert*, *Insert*, and *Signal_Tumor_Reached*[3]. Operation *Signal_Tumor_Reached* is triggered every time a needle is moved (see *TrigPre* compartment), under the condition that the point in which the needle is positioned equals the target point on the cancer (see *DomPre* compartment). This operation generates event *Tumor_Reached* that refers to the same needle that has been previously moved (see *ReqPost*). This operation causes a change of state for entity

[3] For reasons of space we just enlarged the compartments of operation *Signal_Tumor_Reached*

Needle that transits from state *inserting* to state *positioned* (see *DomPre* and *DomPost* compartments). This Properties View allows to configure other additional attributes of an operation, such as its input and output entities/events,, a descriptions and a minimum and maximum time it can be executed.



```
DomPost  Def n.state == positioned
         n
DomPre   Def n.state == inserting
         n
ReqPost  Def AND(tr, tr.needle == n.num)
         tr
         n
ReqPre   Def AND(mp.needle == n.num, mp.point != n.skinPoint, n.initPos == n.targetPoint, mp.di...
         mp
         n
TrigPre  Def mp
         mp
```

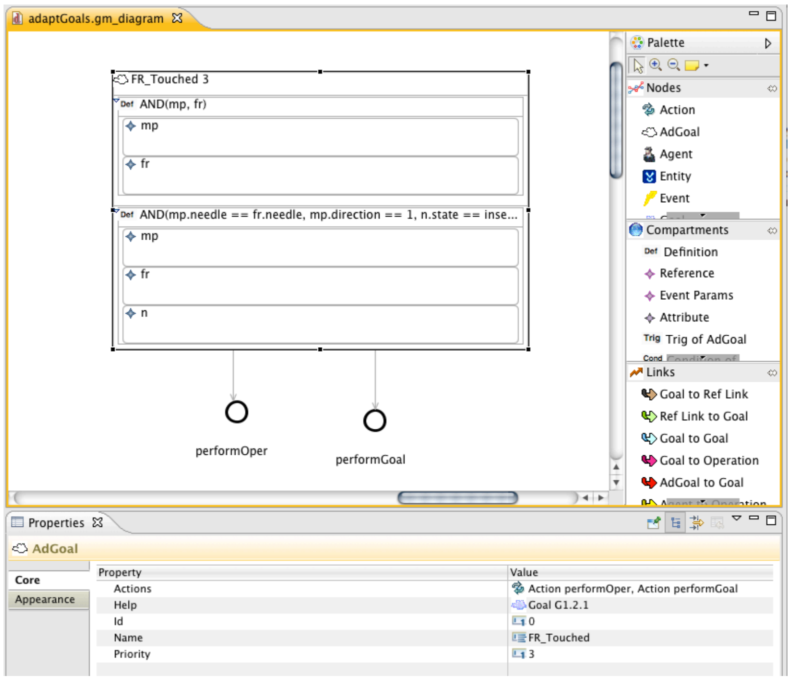
Property	Value
Description	Sends a notification when a needle reaches the target point on the cancer
Hasev	Event Tumor_Reached, Event Move_Performed
Id	0
In	Entity Needle
Max Exec Times	0
Min Exec TImes	0
Name	Signal_Tumor_Reached
Out	Entity Needle

Fig. 6. A subset of the operations of the puncturing case study.

Note that all pre- and post-conditions are represented through a suitable compartment. Each of these compartments is composed of one sub-compartment that contains the definition of the pre- or post-condition (expressed in the FLAGS language) and other sub-compartments that map each variable used in the definition of the pre- or post-condition with an entity/event in the domain model.

3.4 The adaptation view

Adaptation goals define the adaptation capabilities embedded in the system at requirements level. They are characterized by a condition and a trigger. A condition specifies a set of properties of the system (e.g., satisfaction levels of conventional goals, adaptation goals already performed) or of the environment (e.g., constraint on the domain) that must be true to activate an adaptation goal. A trigger is associated with one or more events that activate the execution of the adaptation goal if the corresponding conditions are satisfied too.



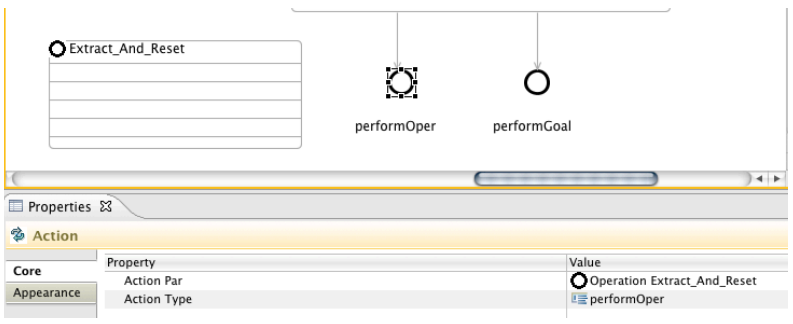
Property	Value
Actions	Action performOper, Action performGoal
Help	Goal G1.2.1
Id	0
Name	FR_Touched
Priority	3

Fig. 7. Adaptation goal FR_Touched.

Adaptation goals are associated with a set of actions to be carried out when adaptation is required. These actions can simply change the way goals are achieved, by performing some goals/operations, without modifying the existing FLAGS model, or, alternatively, they can modify the model, by, for example, adding/removing goals and operations. Adaptation goals also have a priority to select one among different adaptations that may be triggered at the same time.

Figure 7 shows adaptation goal *FR_Touched*, which is performed to achieve goal G1.2.1 even if a needle touches a forbidden region. In this case, the needle is extracted and inserted again in a different position. The properties view allows the users to configure a set of attributes of an adaptation goal, such as the goal it helps (G1.2.1 in the example), its priority (i.e., 3) and the adaptation actions to be performed. The trigger and conditions are represented through two compartments. The structure of each compartment is similar to that used to express the pre- and post-conditions of the operations. Adaptation goal *FR_Touched* is triggered by events *Move_Performed* and event *FR_TouchedEv*. They indicate that adaptation is performed after a needle is moved and a forbidden region has been touched. The condition verifies that the event *FR_TouchedEv* refers to the same needle that performed the movement ($fr.needle == fr.needle$), and the needle is being inserted ($mp.direction = 1$).

As shown in Figure 8, this adaptation goal is operationalized through two adaptation actions: perform operation *Extract_And_Reset* and perform goal G1.2.1. Operation *Extract_And_Reset* has been added only for adaptation purposes. It extracts the needle that touched a forbidden region and changes its configuration parameters. Action perform goal G1.2.1 re-performs the operations associated with goal G1.2.1 (e.g., a needle can be selected again to be inserted) to enforce its satisfaction.



Property	Value
Action Par	Operation Extract_And_Reset
Action Type	performOper

Fig. 8. Adaptation action perform operation *Extract_AND_Reset* .

4 Related Work

In our previous work [9] we already compared the FLAGS model with other approaches used to represent the requirements of self-adaptive systems. FLAGS is one of the most expressive models that is able to represent almost all features of a feedback loop. Differently from the other modeling approaches, FLAGS also supports the evolution of the system by offering the possibility to change the requirements model at runtime.

Other tools [10,11] have been already provided to model and manage the requirements of the system. Objectiver [10] is a graphical designer to create and manage the requirements expressed according to the KAOS methodology. Objectiver does not provide support to model the adaptation capabilities of the system or transform goals into runtime objects. Instead, it focuses on the requirements elicitation and can transform the KAOS model into textual requirements documents conforming to existing standards, such as the IEEE Software Requirements Specification (IEEE 830-1998).

TAOM4E [11] relies on the TROPOS [12] methodology to translate early requirements into the code of BDI (Belief-Desire-Intention) agents. The tool has been released as an Eclipse plug-in by the Software Engineering group at FBK IRST in Trento. Similarly to the FLAGS Designer, TAOM4E is a research prototype that has been designed according to the MDA principles. Despite these commonalities, the FLAGS Designer has a different objective, since it is aimed to collect requirements to engineer the feedback loop.

5 Conclusions

This paper describes an eclipse plug-in to design and manage the requirements of the system together with its adaptation capabilities. This is still a preliminary solution towords the final objective of providing a complete transformation of requirements into the code of the feedback loop and support the evolution of requirements at runtime. The adoption of other existing Eclipse projects, such as EMF, GMP, and Xtext, aided the development process and fostered the usability and extensibility of the FLAGS Designer. The FLAGS Designer is still in its first release and needs further improvements. The users must be guided during the design of the requirements and must receive suggestions regarding the parts of the model that still need to be defined. In this first release the correctness check of the goals definition has not been integrated with the rest of the designer and must be performed separately. In this phase we have just provided support for the automatic generation of monitors [13] that assess the satisfaction of fuzzy requirements at runtime. However to achieve the final objectives of engineering the feedback loops, we also need to provide automatic mechanisms to convert adaptation goals into adaptation actions that are activated at runtime, when necessary.

References

1. Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/`
2. Graphical Modeling Project (GMP). `http://www.eclipse.org/modeling/gmp/`
3. Xtext Language Development Framework. `http://www.xtext.org`
4. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-Driven Adaptation. In: Proc. of the 18th Int. Requirements Engineering Conf., (2010) 125–134
5. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley (2009)
6. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Proc. of Future of Software Engineering. (2007) 259–268
7. ISUR Project Website. `http://www.isur.eu`
8. Object Constraint Language Version 2.2. `http://www.omg.org/spec/OCL/2.2/`
9. Pasquale, L.: A Goal Oriented Methodology for Self-Supervised Service Compositions. PhD thesis, Politecnico di Milano (2011)
10. Objectiver. `http://www.objectiver.com/`
11. Fondazione Bruno Kessler: Tool for Agent Oriented Modeling TAOM4E. `http://selab.fbk.eu/taom/`
12. Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and Analyzing Early Requirements in TROPOS. RE **9**(2) (2004) 132–150
13. Pasquale, L., Spoletini, P.: Monitoring Fuzzy Temporal Requirements for Self-Adaptive Systems: Motivations, Challenges and Experimental Results. In: Proceedings of SOCCER'11. (2011) (to appear)