A Meta-model for Problem Frames: Conceptual Issues and Tool Building Support

P. Colombo, L. Lavazza, A. Coen-Porisini
Dipartimento di Informatica e Comunicazione
Università degli Studi dell'Insubria
Varese, Italy
{pietro.colombo, luigi.lavazza, alberto.coenporisini}@uninsubria.it

V. del Bianco
Systems Research Group, CASL
University College Dublin
Dublin, Ireland
vieri.delbianco@ucd.ie

Abstract—**Problem frames are an approach to requirements modeling that is gaining increasing attention and popularity. The approach provides useful concepts and methodological guidelines. However, problem frames are not equipped with an expressive and complete notation and they lack tools support. These limitations can be addressed by introducing a suitable meta-model to formally define the notation. In this way it is also possible to identify the aspects that are not covered by the problem frames notation and to provide hooks for user-defined extensions. The meta-model is expected to support the underlying analysis methodology, and the following design and verification phases. Furthermore, it can provide the basis for building a tool supporting both the editing of problem frames and the other activities associated with the approach (frame concern, composition, correctness argument, etc.). This paper presents a meta-model that addresses the former issues and was used for building a tool with the EMF/GMF technology.**

Keywords- meta-modeling; modeling tools; problem frames.

I. Introduction

Problem Frames (PFs) [1] are an approach to requirements analysis and modeling that drives the analyst to model the problem in terms of (physical) problem domains, their properties, the information they exchange and the user requirements. The solution of the problem is specified in terms of a machine, whose behavior is defined so that the interaction of the machine with the given environment satisfies the requirements.

PFs allow analysts to analyze complex problems by decomposing them into simpler ones; these basic problems are modeled according to basic patterns (i.e., the frames, which represent common, well understood problems). Then the analyst can show that the user requirements are satisfied by the outcome of the previously defined modeling activity; finally, the various problem frames are composed into a complete description.

While a great effort has been dedicated to define the PF methodology, little attention was given to the definition of an expressive and complete notation and to tools supporting PFs. For instance, PFs do not provide any language for describing the properties of problem domains, or for specifying the desired behavior of the system: the analyst has to select and use a language among the available ones (in [1] Jackson uses state-charts, pseudo-code, and natural language). Problem Frames also lack automated support: no tool is available for defining, analyzing, or composing PFs.

The aforementioned problems can be solved with the help of a meta-model that defines precisely the Problem Frames concepts, supports the methodology, and provides the basis for the construction of tools.

An initial proposal of a meta-model for Problem Frames was presented by the authors of this paper in [21]. The usage of the proposed meta-model in the construction of a prototype tool using the meta-model in combination with EMF [13] and GMF [14] was also discussed.

The meta-model presented did not cover a very important part of the Problem Frame methodology, namely the correctness argument [1]. This paper is an extended version of [21]. Besides refining the material already presented in [21], here we illustrate and discuss the usage of the meta-model in describing the requirements, domain characteristics (with special reference to behavioral properties), and machine specifications. These are the ingredients for building correctness arguments, that is, for showing that the proposed machine specification satisfies the requirements in the problem domain.

A PF-based development process is introduced as well, in which PF models are exploited also in the design and verification phases.

The paper is organized as follows: Section II provides a brief introduction to Problem Frames; Section III illustrates the proposed meta-model, while Section IV describes the UML definition of the meta-model and exemplifies the usage of the meta-model in describing a problem. Section V illustrates the usage of the meta-model for expressing requirements and describing machine and problem domain behavior, and the support to correctness arguments. Section VI describes the construction of a tool based on the meta-model, exploiting the EMF/GMF methodology. In Section VII a PF-based development process is introduced; Section VIII accounts for related work; finally Section IX draws some conclusions.

II. Problem Frames

Problem Frames are based on the concept that user requirements are about relationships in the real world and not about functions that the software system must perform. The desired relationships in the real world are achieved with the help of a machine; however, in the requirements analysis phase, the Machine is only specified as far as its role in the real world is concerned: only the interface between the machine and the problem domain needs to be specified, while the machine internals are left unspecified, since they will be addressed in the design phase.

Thus, the first task is to understand and represent the context in which the problem is set: the context diagram shows the various problem domains in the application environment along with their connections, and the Machine and its connections to (some of) the problem domains. A domain is simply a part of the world that we are interested in. It consists of phenomena such as individuals, events, states, relationships, and behaviors. An interface is a place where domains overlap, so that the phenomena in the interface are shared, thus allowing connection and communication between domains. A set of shared phenomena is controlled by a domain and is observed by other domains.

Problem diagrams add requirements to context diagrams. Requirements are attached to domains and specify conditions involving the phenomena of those domains (possibly including the private, non-shared ones).

An interface that connects a problem domain to the Machine is called a specification interface. The goal of the analyst is to develop a specification of the behavior that the Machine must exhibit at its interface in order to satisfy the user requirements. A PF is a description of a recognizable class of problems, and thus in some sense problem frames are problem patterns.

Figure 1 shows an example of a *commanded behavior* frame: "there is some part of the physical world whose behavior is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly [1]".

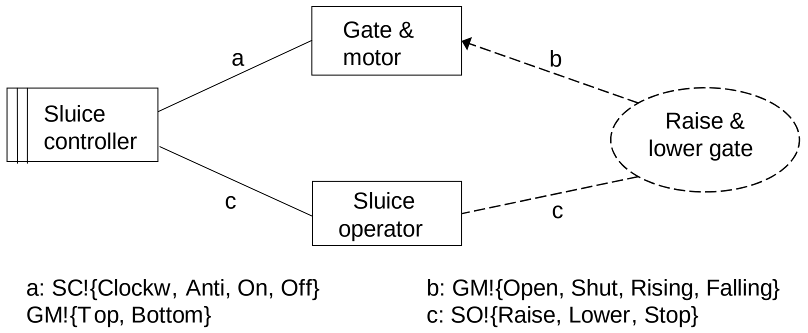
Gate & motor
a
b
Sluice controller
Raise & lower gate
c
Sluice operator
c
a: SC!{Clockw, Anti, On, Off}
GM!{Top, Bottom}
b: GM!{Open, Shut, Rising, Falling}
c: SO!{Raise, Lower, Stop}

Figure 1. The sluice gate commanded behavior frame.

Such a frame (illustrated in Figure 1) is described using a simple example concerning the specification of a controller that operates a sluice gate. A small sluice, with a rising and a falling gate, is used in a simple irrigation system. A computer system is needed to raise and lower the sluice gate in response to the commands issued by an operator. The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors both at the top and the bottom of the gate travel: when the top sensor is active the gate is fully open, when the bottom sensor is active, it is fully shut. The connection to the computer consists of four pulse lines for motor control, two status lines for the gate sensors, and a status line for each class of operator commands. The position of the gate is defined as the fraction of space occupied by the gate: when it is open Position=0, when it is closed Position=1. Finally, the top and the bottom sensors are active when Position becomes less than 0.05 and greater than 0.95, respectively.

The PF diagrams involves three domains: the Sluice Controller, which is the machine that will be developed to satisfy the requirements; the Gate & motor, which is the domain to be controlled (it is a causal domain since its properties include predictable causal relationship among its causal phenomena); the Sluice Operator, which is a biddable domain indicating a user without a positive predictable behavior (that is, the user can issue commands but cannot be constrained to act in any way).

It has to be assured that requirements, domain and specification descriptions fit together properly. Addressing this issue (the "frame concern") must result in a 'correctness argument' showing that the proposed machine will make the requirements satisfied in the problem domain [1].

In the case of the commanded behavior frame, we have to assure that only sensible and viable commands are executed. Requirements can be expressed as effects on the problem domain caused directly by the user's commands or by other events, such as reaching the completely open or closed position. According to Jackson, these effects can be expressed in a rather straightforward way by means of state machines. Also the behavior of the problem domain can be represented by means of a state machine, showing the states of Gate & motor, and specifying the reactions to external commands, as well as the evolution in time of the domain. For instance, the behavior of the Gate & motor domain is specified by the state machine reported in Figure 2 (taken from [1]); state 5 is an 'unknown' state, which should never be reached in normal operations; in fact, the gate would probably break if entering this state were attempted.

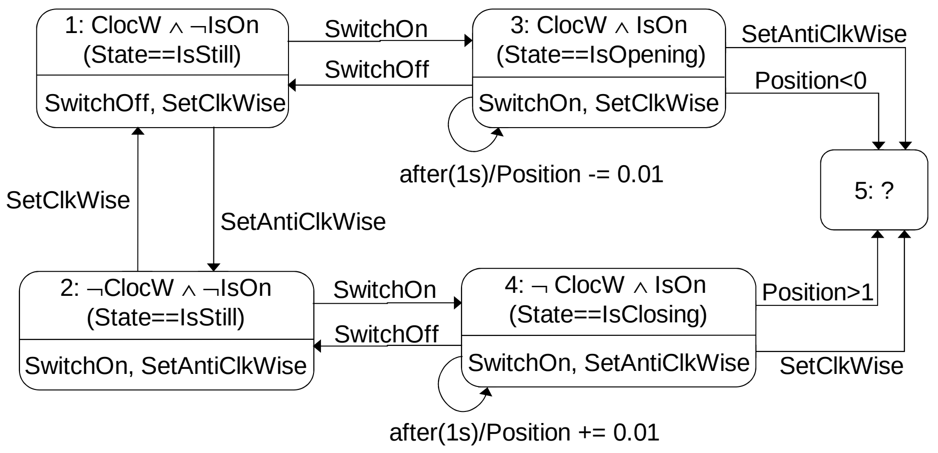

Figure 2. The specification of the Gate & motor domain.

III. The Meta-Model

In order to build a tool that supports the PFs approach, several aspects need to be defined. Our approach consists in introducing a meta-model that supports the definition of all the aspects necessary to specify both notational and methodological concepts.

Notational concepts are used to represent the structural elements of a problem, the behavioral properties associated with such elements, the user's goals, and the machine specification. In our case the notational concepts have to support the representation of the Problem Frames diagrams as specified in [1].

Methodological elements are a collection of concepts, rules and suggestions that drive requirement analysis. For instance, phenomena that are internal to a domain are modeled, although they do not appear in problem diagrams, because they can be useful to define shared phenomena and the domains behavior.

Moreover, the definition of the meta-model allows us to identify possible inconsistencies, weaknesses or incomplete definitions in the notation, and therefore to propose solutions to address such issues. The meta-model introduces the elements needed to describe the following concepts:

- The basic structural elements and connections associated with a problem.
- The dynamic and behavioral properties associated with structural elements.
- The goals of the user, i.e., the user requirements.
- The specification of the solution, i.e., of the machine.
- The decomposition criteria.

The Problem Frames specific elements to be addressed are the following:

- A *Problem Domain* represents a physical domain of the environment where the problem is located, whose properties can be either given or designed by the user. A *Machine Domain* is a computer that interacts with the Problem domains in a way that satisfies the requirements.
- *Phenomena* are properties of a domain and can be classified as Entities, Events or States.
- *Interfaces* are connections between Domains characterized by shared phenomena.
- A *Shared phenomenon* is controlled by a domain and observed by one or more other connected domains.
- The *behavior* of a domain is specified in terms of the involved domain's phenomena. Even though the PFs methodology does not prescribe a notation for describing the behavior, the meta-model should be able to explicitly indicate the existence of a behavioral specification element and which phenomena are involved.
- *Requirements* are associated with domains; requirements are described in terms of domains' phenomena; in particular, they should be modeled as capable of referring to and constraining phenomena.
- *Machine specification*s specify the properties of the machine's interface with the problem domains.

In the next section the meta-model is described using UML [16]. This choice is motivated by the expressiveness of the language and by its diffusion in the communities of analysts and designers. Moreover, the serialization of a UML model via XMI [17] is recognized as a valid description of the meta-model by frameworks –like EMF [13]– that support the generation of tools.

IV. UML description of the meta-model

A UML Class Diagram that introduces the essential features of the meta-model is shown in Figure 3.

The root element of the model is named *PFsModel*. It is composed of entities representing essential structural concepts such as *Domain* and *Interface*.

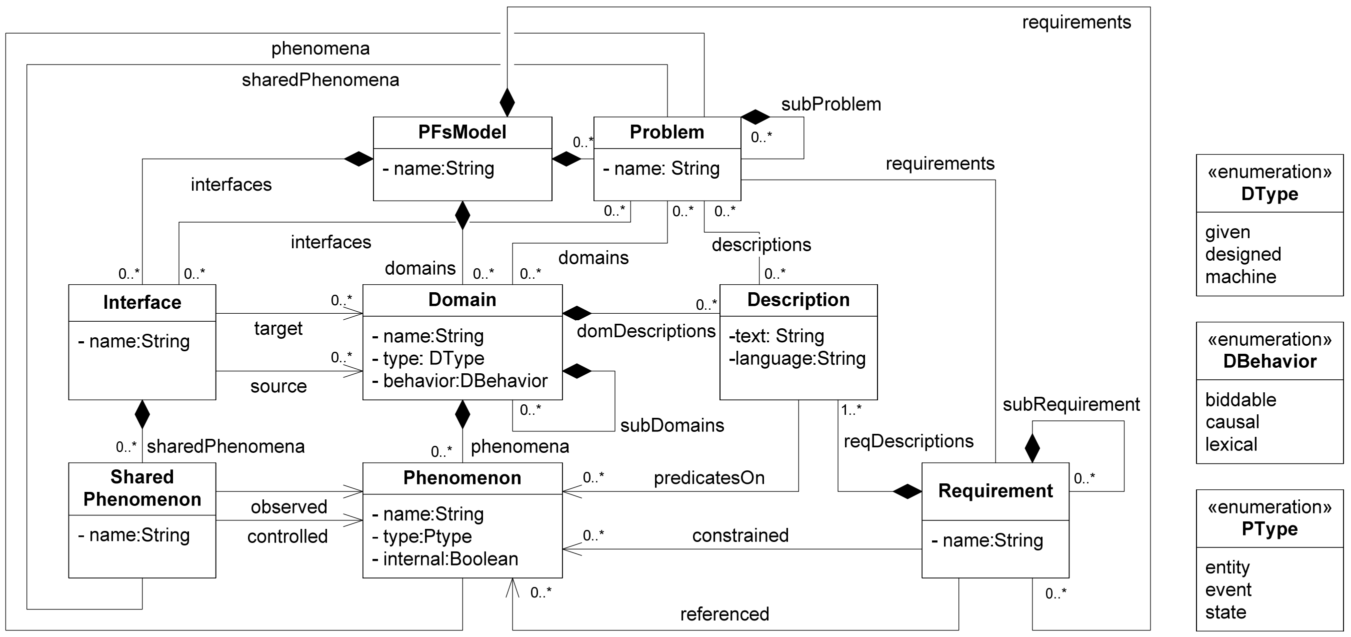

Figure 3. The UML Class Diagram defining the meta-model.

Domain is characterized by an attribute *name* (for identification purposes), an attribute *type* (to express whether the domain represents the machine, or it is given or designed), an attribute *behaviour* (to specify whether the domain is lexical, biddable or causal). The attributes *type* and *behavior* are typed by means of two enumerative data types named *DType* and *DBehavior*, respectively. Some constraints are defined on the values associated with these attributes. More specifically, a machine domain is always a causal domain, and a designed domain is never biddable. These properties are specified by means of OCL [15] constraints:

```
context Domain inv:
 ((self.type=Dtype::machine) implies
       (self.behavior=Dbehavior::causal)) and
 ((self.type=Dtype::designed) implies
       (self.behavior<>Dbehavior::biddable))
```

Another constraint imposes that domains have unique names:

```
context Domain inv:
Domain.allInstances()->forAll(p1, p2 | p1 <> p2
implies p1.name <> p2.name)
```

Similar rules are defined to assure that distinct elements of a model are given different names.

Domain is composed of sub-domains and internal phenomena. *Phenomenon* is characterized by the attributes *name* and *type*. According to Jackson, the latter is used to express whether a phenomenon is a state, an event, a value, etc. The attribute *type* is typed by means of the enumerative data type *PType*; the Boolean attribute *internal* specifies whether the phenomenon is owned and controlled or just visible by the connected domain. Also in this case some constraints are introduced. More specifically, a lexical domain cannot be characterized by causal phenomena such as events or states.

```
context Domain inv:
 self.phenomenon->forAll(p |
  self.behavior=Dbehavior::lexical implies
  (p.type<>Ptype::event and p.type<>Ptype::state))
```

Domains can be connected by means of the element *Interface*. Two directional association relationships named *source* and *target* connect the class *Interface* to the class *Domain*. Notice that the terms ‘target’ and ‘source’ do not imply that the interface has an orientation. A constraint is defined in order to assure that the involved domains are distinct:

```
context Interface inv: self.target <> self.source
```

An *Interface* exists when one or more phenomena are shared between two domains. The shared phenomenon concept is represented in the meta-model by the homonymous class. In the proposed meta-model, whenever a phenomenon (for instance, SC!Off in Figure 1) is shared, a corresponding phenomenon is created and added to the phenomena of the connected domain (in our example, a non internal phenomenon is added to the *gate&Motor* domain). An instance of *SharedPhenomenon* is also created, and connected to the instances of the corresponding phenomena (in our example, *controlled* will identify the *Off* phenomenon of *sluiceController*, while *observed* will identify the *Off* non internal phenomenon of *gate&Motor*). This rather baroque representation is motivated by the goal of using the meta-model for the development of a tool based on GMF/EMF technology. In fact, in order to guarantee that an element of a model is accessible for editing, such technology imposes that the element belongs to a containment hierarchy having the diagram being edited as root. In order to satisfy such constraint, we identified the following solution: domains contain phenomena, and interfaces contain shared phenomena that in turn refer to the phenomena (both controlled and visible) of domains.

An additional constraint assures that the usage of the relationships *controlled* and *observed* is consistent with the value of the attribute *internal*.

```
context SharedPhenomenon inv:
 (self.controlled.internal=true and
 self.observed.internal=false and
 self.controlled.name=self.observed.name) and
 self.name=self.observed.name
```

The following constraint states that every instance of SharedPhenomenon is properly connected.

```
context SharedPhenomenon inv:
 (self.controlled.domain=self.interface.target and
 self.observed.domain=self.interface.source) or
 (self.controlled.domain=self.interface.source and
 self.observed.domain=self.interface.target)
```

The proposed solution supports the association of a different, specific editor with each element of the meta-model: the editable elements are those recursively contained in the element; the elements that are reachable from the considered element via non containment relationships can be accessed by the editor in a read-only manner. As an example, the proposed solution supports the definition of a diagram editor for domains and another editor for interfaces. With the former, internal phenomena may be added to the domain instance, which is the root element of the diagram. With the interface editor,, shared phenomena that refer to the internal phenomena of the involved domains are added to an instance of *Interface*. No internal phenomena can be added to a domain with the interface editor, being only possible to refer to existing instances.

The specification of the behavior of the domains is supported by means of the element *Description*. *Description* allows the model to be extended, i.e. several kinds of elements can be attached to this element for specifying the behavior by means of *ad hoc* notations. In fact, descriptions can be expressed with different notations such as state machines, natural language, formal languages, modeling languages like UML or SysML, etc. This can be done by importing elements from the meta-models of external notations, and by connecting them to *Description*; both the choice of which elements to import and the definition of the associations depend on the involved notation. Extending the PFs meta-model is out of the scope of this work and therefore the meta-model simply provides two attributes named *text* and *language*. The former describes the behavior by means of a textual description, while the latter indicates in which language the description is written.

Descriptions predicate on the phenomena of a domain (both controlled and visible); this concept is expressed by means of a directional association named *predicatesOn* between the classes *Description* and *Phenomenon*.

PFsModel also includes class *Requirement*, whose instances are crosscutting elements that specify static or dynamic properties with reference to the structural elements of a model. Class *Requirement* represents the user requirements, expressed by predicating on the domains' phenomena. The class is characterized by the attribute *name*, and by the relationships *constrained* and *referenced*, which express whether the requirement constrains the phenomena or just observes them. The specification of requirements is supported by the class *Description*. In fact, in the meta-model, the requirements, the machine specification and the behavior of domains are all represented by the same element *Description*.

PFsModel also introduces concepts that aim at supporting problem decomposition. More specifically, the problem concept is defined by the class *Problem*, while the decomposition is represented by the *subProblems* relationship. Other relationships are defined in order to express that a problem is characterized by requirements and domains, which are interconnected by means of interfaces. Notice that such relationships are simple associations, i.e. an instance of *Problem* is associated with instances of other elements that are contained in an instance of *PFsModel*. Such relationships support both the decomposition of a problem, and the definition of multiple views. A problem may involve only a subset of the domains (and of the corresponding phenomena) of a model: instances of *Problem* may be considered partial views on the model, consisting of subsets of the elements contained in an instance of the *PFsModel* class.

Figure 4 reports a fragment of the instance of the proposed meta-model that describes the sluice gate control problem. In particular, the model contains the Gate&Motor and SluiceController domains and their internal phenomena. Moreover, interface 'a', which connects the two domains (see Figure 1), is also shown: the interface involves a set of shared phenomena, each one corresponding to an internal phenomenon of the controlling domain and an external phenomenon, which is observed by the other domain participating in the interface. For instance, the phenomenon 'on', controlled by the SluiceController is made visible to the Gate&Motor domain through interface 'a' and the shared phenomenon 'onSP'.

The description is actually a bit redundant, with each phenomenon represented several times; however, this kind of organization was practically imposed by the constraints due to the usage of GMF.

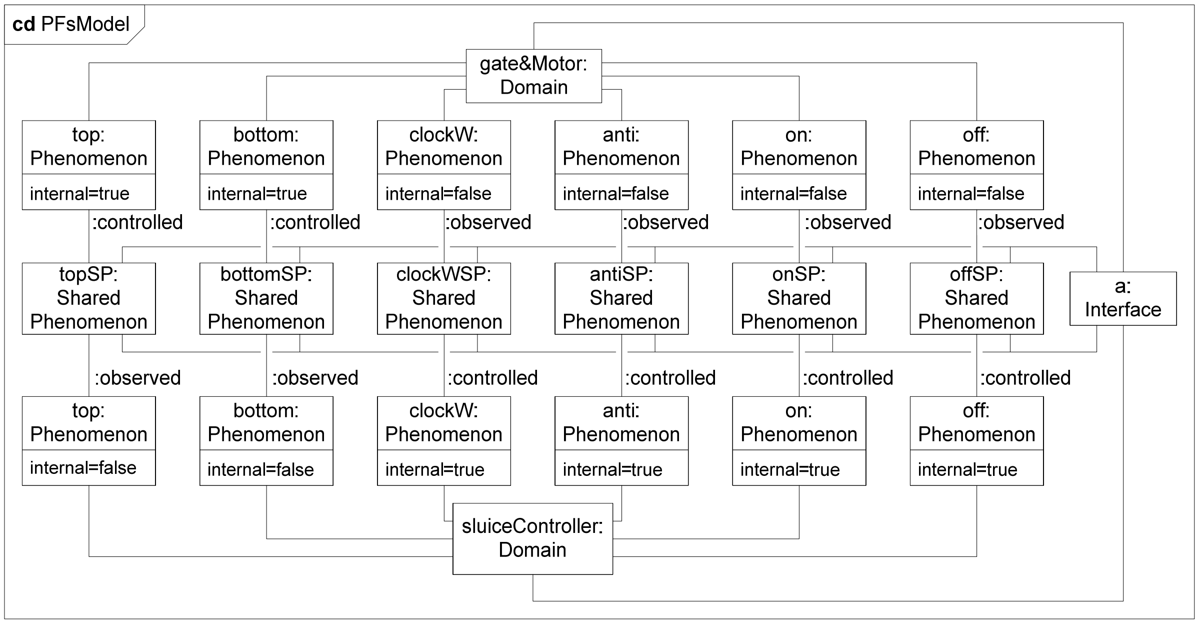

Figure 4. An instance of the meta-model (fragment).

V. Representing requirements and reasoning about correctness

A fundamental part of the problem frame methodology deals with correctness arguments.

With problem frames, the idea is that requirements are described as relations that the user wants to be established among domains in the problem environment. Requirements are therefore given by means of optative descriptions involving problem domain elements. For instance, a requirement of the sluice gate control system is that when a Raise command is issued and the state of the system makes the required operation sensible and viable, the command is executed, i.e., the gate starts rising.

Some characteristics of the relevant domains belonging to the problem environment have also to be described, because they contribute to the actual behavior of the proposed solution. For instance, the fact that the gate starts moving when the motor is set on, or that the Bottom signal is issued when the gate reaches the closed position clearly contribute to the behavior of the system. The behavior of given domains is specified by means of indicative descriptions.

The machine is the hardware/software part of the proposed solution. Its behavior is defined via suitable specifications that involve only the machine interface. The machine specification must guarantee that the interaction of the machine with the problem domain causes the required relations in the problem environment to hold.

The correctness argument must convince that the proposed machine satisfies the requirements in the problem domain.

Figure 5 illustrates a piece of the correctness (or adequacy) argument for the sluice gate control problem. Figure 5 is an adaptation from [1]. According to [1], in this kind of problem, requirements (1) state what commands are sensible in which situations and (5) what effects they should cause in the problem domain if they are viable. The specifications of the machine (2 and 3) define what is the reaction of the machine to commands (including those that are not sensible or viable). The description of the behavior of the problem domain describes how the domain state and behavior are affected by what the machine does at their shared interface.

Figure 5 provides an excerpt of the just mentioned description concerning a specific case (i.e., what happens when the Raise command is issued and the Gate is closed). The correctness argument shows the domain behavior resulting from the commands and that the final state of the system complies with the requirements, which prescribe the consequences of commands. In order to support this type of argument, the meta-model must be able to support the proper description of requirements (in terms of phenomena of the problem domains), of domain behaviors (in terms of their own phenomena and phenomena that are visible because shared by other domains, including the machine) and of machine specifications (in terms of phenomena shared through its interfaces; talking about machine's internal phenomena is strictly forbidden in this phase).

Figure 3 shows that the meta-model includes "descriptions" that belong either to domains (including the machine) or to requirements. These descriptions consist of text, written in some language, and of references to the phenomena that are mentioned in the description itself.

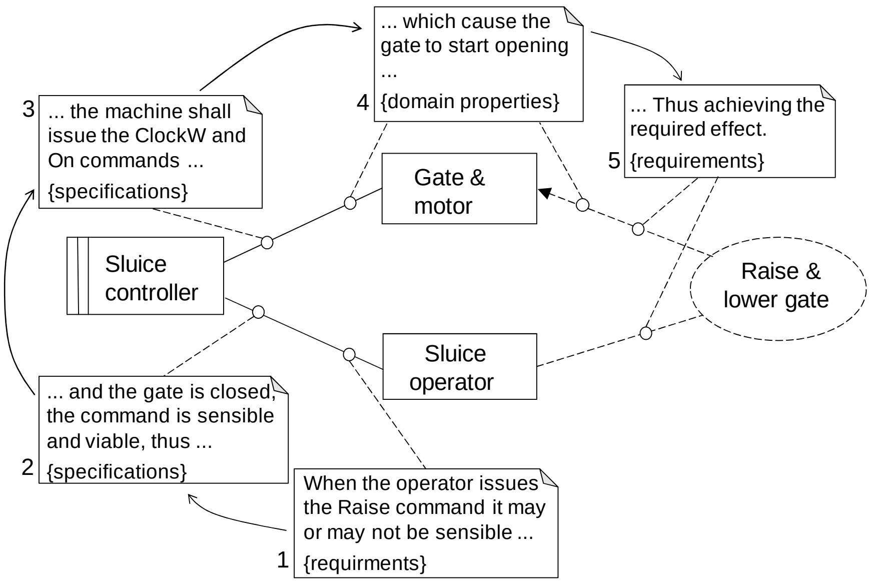
... which cause the gate to start opening ...
4 {domain properties}
3 ... the machine shall issue the ClockW and On commands ...
{specifications}
... Thus achieving the required effect.
5 {requirements}
Gate & motor
Sluice controller
Raise & lower gate
Sluice operator
... and the gate is closed, the command is sensible and viable, thus ...
2 {specifications}
When the operator issues the Raise command it may or may not be sensible ...
1 {requirments}

Figure 5. An adequacy argument.

Figure 6 shows a fragment of an instance of a meta-model, reporting the requirement that prescribes the effect of a Raise command when the gate is closed. In Figure 6 the textual description of the requirement is written in plain English. Therefore, it brings no meaning to a possible tool using the meta-model (unless, perhaps, sophisticated artificial intelligence techniques are used; we do not consider this possibility). However, the underlined words in the descriptions have a specific meaning, comprehensible by a tool using the meta-model: they correspond to the references to phenomena having the same names. Therefore, when the analyst that is defining the model of a system selects the phenomena that are relevant for a requirement, he/she is also determining the vocabulary that can be used in the textual description of the requirement.

Note that the analyst, when describing requirements, has to classify every phenomenon connected to a requirement as 'referenced' or 'constrained', according to the notation defined in [1].

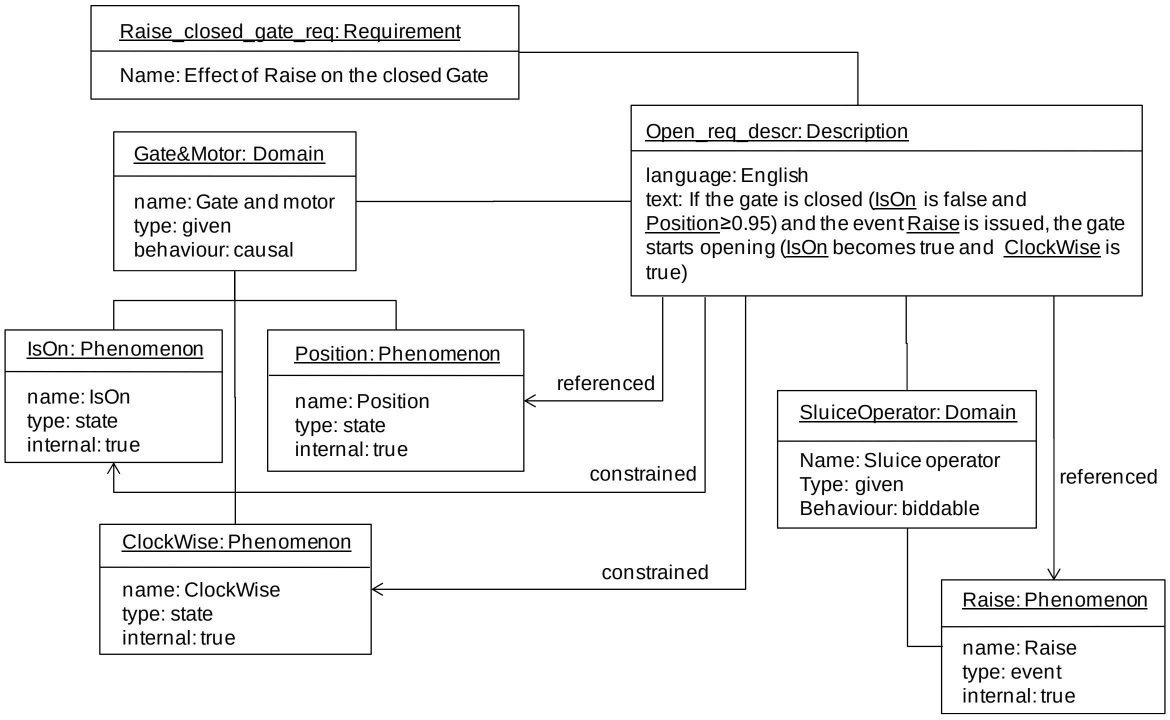

Figure 6. A fragment of meta-model instance that specifies a requirement.

Figure 7 shows a piece of the description of a domain, namely the Gate and motor. It is possible to see that this type of description works exactly like a requirement description. The only difference is that –following Jackson– we do not distinguish referenced phenomena from constrained ones. However, it must be noted that since external phenomena are always referenced, it is possible to omit their classification.

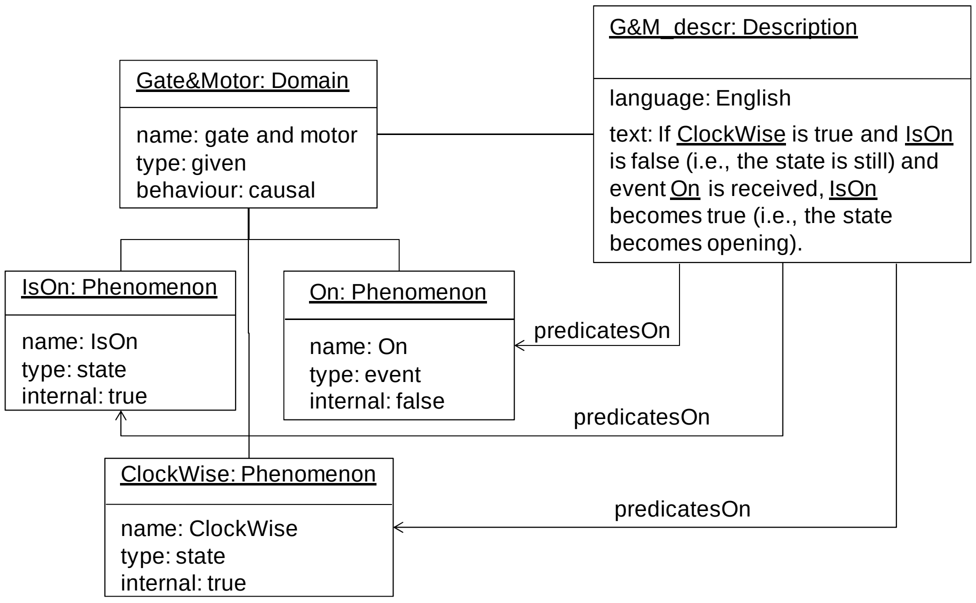

Figure 7. A fragment of meta-model instance that specifies the behaviour of the Gate&Motor given domain.

Finally, Figure 8 reports the specifications of the machine. The diagram is similar to those describing requirements and domain behavior. However, more instances of domains and phenomena are involved, since the specifications deal with phenomena from three different domains (the machine, the operator and the Gate and motor).

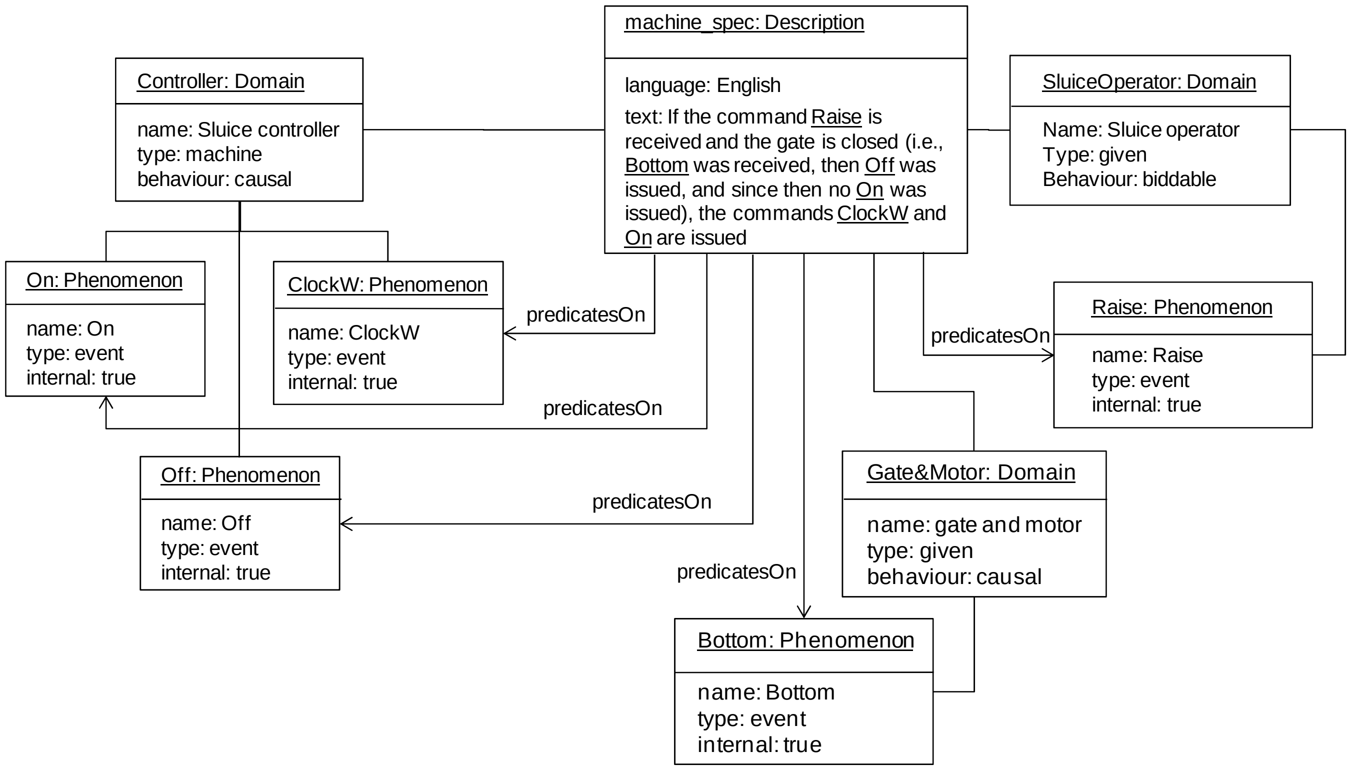

Figure 8. A fragment of meta-model instance that specifies a piece of the machine specifications.

The above reported descriptions provide the information needed to build correctness arguments. In principle, it could be possible to develop a tool that assists the user in building such arguments. In fact, the tool could help the analyst in selecting domains and phenomena that are relevant to the argument. For instance, given the Raise command, the tool could automatically select the involved requirements, the triggered machine reactions, etc., thus providing the user with the 'bricks' that can be used to build the argument.

In [1] Jackson uses a few different notations, and leaves the analysts free to choose the notation they like the most. Accordingly, the proposed meta-model allows the analyst to use any notation: it is only necessary to specify the 'language' attribute of the descriptions.

The fact that the description of domains, requirements and the machine can be used to build correctness requirements, suggests that better results could be achieved if a formal notation is used. In order to illustrate this possibility, in what follows we rewrite the descriptions already reported in Figure 6, Figure 7 and Figure 8 using event calculus (EC).

The EC is a system of logical formalism, which draws from first-order predicate calculus [24]. EC has already been used for describing and reasoning about event-based temporal systems, and has been used in conjunction with problem frames [23][22]. Of the several variations of EC that have been proposed, the version discussed in [25] was used in [22]; we also use that version.

Domain behaviour

If ClockWise is true and IsOn is false (i.e., the state is still) and event SwitchOn is received, IsOn becomes true (i.e., the state becomes opening).

```
HoldsAt(ClockWise ∧ ¬IsOn,t) ∧ Happens(On,t) →
HoldsAt(IsOn, t+1)
```

The state IsOn persists until SwitchOff is issued

```
HoldsAt(IsOn,t) ∧ ¬Happens(Off,t) →
HoldsAt(IsOn,t+1)
```

Requirements

If the gate is closed (IsOn is false and Position≥0.95) and the event Raise is issued, the gate starts opening (IsOn becomes true and ClockWise is true).

```
Holds(¬IsOn ∧ Position≥0.95,t) ∧
Happens(Raise,t) → Holds(IsOn ∧ ClockWise, t+1)
```

Machine specifications

If the command Raise is received and the gate is closed (Bottom was received, then Off was issued, and since then no On was issued), the commands ClockW and On are issued

```
Holds(Closed,t) ∧ Happens(Raise,t) →
Happens(ClockW, t+1) ∧ Happens(On, t+2)
```

```
Holds(Closed,t) ← Happens(Bottom,t1) ∧
Happens(Off,t2) ∧ t1<t2<t ∧ ¬∃t3
(Happens(On,t3) ∧ t2<t3<t)
```

Starting from descriptions written in EC, correctness arguments can be built, also with the help of reasoning tools.

In [22] if an event or a fluent is a part of an interface, its name is parameterized –under some circumstances– with the name of the interface. For example, `Happens(e1(p),t1)` indicates that the event e1 is generated by a controlling domain at the interface p at the time t1. Similarly when describing the effect of an event on a fluent that is controlled by a domain, the fluent name is parameterized with the name of the domain. For example, `Initiates(e1(p),f2(D),t)` indicates that when the event e1 occurs at the interface p, the fluent f2 controlled by Domain D becomes true. Our meta-model is defined so that all the mentioned fluents or events correspond to specific phenomena, therefore they are unambiguously and precisely characterized in terms of the domain they belong to and the interfaces they participate into.

VI. From the meta-model to the tool

The meta-model presented above was used as a basis for the development of a tool supporting the editing of Problem Frames as well as other aspects of the approach.

The proposed solution exploits the Eclipse Graphical Modeling Framework (GMF) [13], a "state of the art" technology for the definition of model editors in the Eclipse development framework [18]. GMF provides advanced services that guide the developer in the definition of visual editors starting from a meta-model. The generated editors also provide different kinds of advanced services such as diagram editing, validation, transformation, and support for a standardized XMI model serialization format.

GMF provides both a generative component and a runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) [12] and the Eclipse Graphical Editing Framework (GEF) [11].

EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF consists of three fundamental parts [10]:

- The Core framework: it includes a meta-model (Ecore) for describing models and runtime support for change notification and XMI serialization.
- The Edit framework: it includes generic reusable classes for building editors for EMF models.
- The Codegen framework: it provides code generation facilities to build a complete editor for an EMF model.

EMF supports the definition of OCL constraints by providing a framework usable for property validation. EMF also provides tools for the automatic definition of basic editors that aim at visualizing and manipulating models (instances of the meta-model). GEF is a framework to be used in conjunction with GMF to create graphical editors characterized by a model-view-controller architecture. The development of diagram editors that handle EMF models based on the direct usage of GEF is an onerous activity, since it requires an in-depth knowledge of the architecture and the API of both GEF and EMF. In order to ease the development of graphical editors, the capabilities of EMF and GEF were composed and made available through the GMF infrastructure. In fact, GMF combines the advantages of EMF and GEF, and provides tools that aim at simplifying and automating the generation of diagram editors. The usage of such technologies provides several advantages to the designer:

- A collection of reusable components for graphical editors, such as geometrical shapes, icons, etc.
- A standardized model to describe diagram elements. Diagram elements are described by means of graphical models that define both the characteristics of the visual elements shown in a diagram, and the mechanisms through which it is possible to access them.
- The separation of semantic aspects from diagrams. Semantic elements are defined in an Ecore model, and they are accessed by means of EMF, while diagram models are directly managed by GMF.

The generated editors are open, thus the interested user can access the generated source code in order to modify or extend the functionalities of the editor. Moreover, the generated editors are Eclipse plug-ins; hence extensions can exploit the standard Eclipse mechanisms.

A PF editor has to support problem analysis according to the various concepts of the PFs approach. We decided to partition the required functionalities into several editors, since the involved activities are fairly independent and use different notations. For instance, the specification of the requirements (or of the machine) uses a notation that is different from the one used for defining the problem structure.

Although functionalities are allocated to distinct editors, all the editors operate on the same model. In other words, multiple views insist on the same elements. For instance, a dedicated editor for domain behavior specification is opened whenever the user double clicks a domain instance in the problem editor. Several problems may arise when supporting diagram partitioning: editor instances have to cooperate and to stay constantly synchronized with the state of the global model.

We identified the following distinct editors: A context editor, for editing context diagrams; A problem editor, for editing problem diagram; A domain editor, for specifying the internal structure of a domain (internal phenomena as well as internal sub-domains may be defined); A domain specification editor, for describing the behavior of a domain; An interface editor, for specifying shared phenomena between the domains; A requirement editor, which supports the specification of the user requirement.

The editors were defined according to the typical GMF building process [13]. First of all, an EMF model was defined; in particular, the meta-model proposed in the Section IV –including the properties expressed via OCL– was defined via EMF Ecore technology. The EMF model describes the global model shared among the different editors. Then, a framework supporting the manipulation of the previously defined model was automatically generated using EMF.

All the previously introduced diagram editors were defined starting from the model and the generated editing code. The same GMF process was applied to the definition of each diagram editor.

A graphical model for the representation of diagram elements was defined, using the GMF graphical model creation wizard. A visual layout was defined –according to Jackson's notation [1]– for the elements of the EMF model of Problem Frames, and the access points and services to modify the attributes of each element were also defined.

The definition of the tool models for the manipulation of diagram elements exploited the GMF tool model wizard. The Problem Editor was defined so that all the elements that can be visualized (e.g., domains, requirements and interfaces) can also be edited, while the Interface Editor supports the definition of shared phenomena among domains that are given.

A mapping model specifies which graphical elements can be used in each diagram, and which tool is used for the manipulation of such elements. The definition of the mapping model was performed in part by using the GMF mapping wizard, and in part by configuring the generated model. The resulting model relates the elements of the models defined in the previous steps. Moreover, it supports editor partitioning: this model was used to specify that the interface element of the Problem editor has to be shown in the canvas of an Interface editor.

A generator model was defined for each mapping model specified for the editors by using the GMFGen Model tool. Such model supported the definition of code generation criteria, such as the specification of the serialization formats for diagrams.

Once all the generator models were defined, dedicated tools were used for automatic code generation. Then the generated source code was extended by implementing advanced functionalities, mainly concerning the partitioning of diagrams.

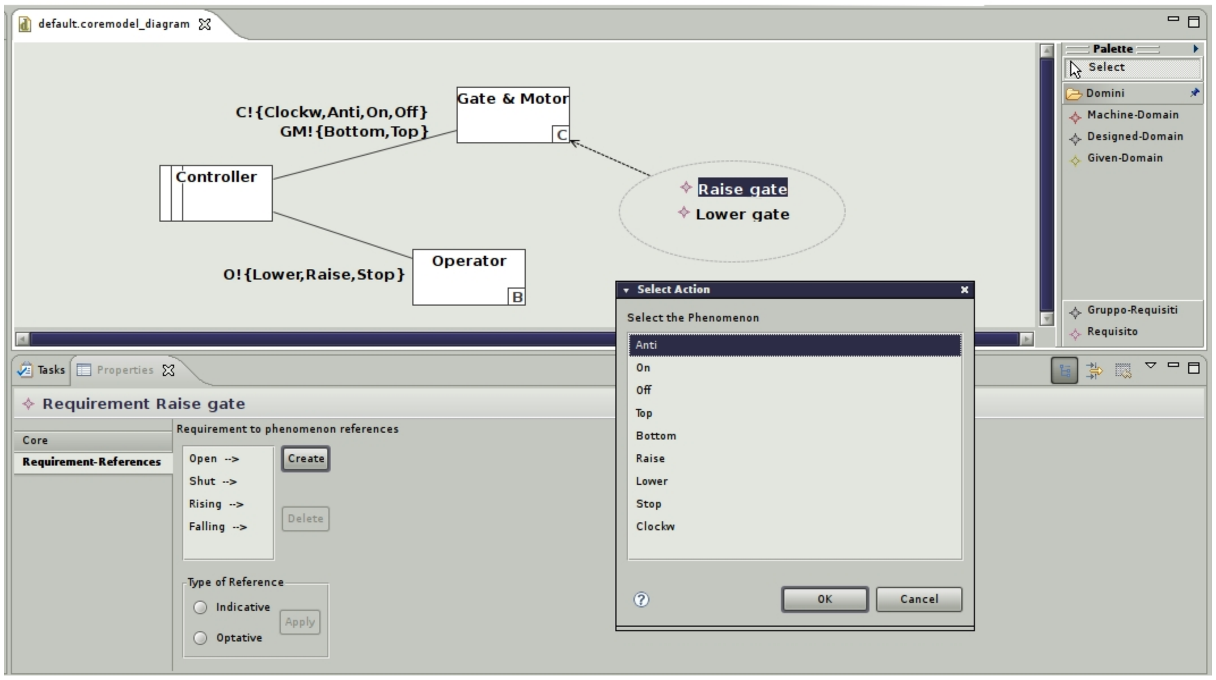

Figure 9. A snapshot of the generated tool.

Figure 9 shows a snapshot of the generated PFs editor, which is composed of the previously described editors and coordinates the activities performed by them. The current prototype is an editor that supports the PF notation. The aspects that are strictly related to the notations/languages adopted for the specifications, like developing correctness arguments, are only partly supported. Future work includes completing the support for problem composition and developing a full support for notation dependent activities.

Our experience with GMF technology was not fully satisfactory. EMF and GEF frameworks are becoming *de facto* standards for the definition of Eclipse based editors, but their combination in GMF appears not mature yet. In particular, problems arise as soon as one tries to define non-trivial editors characterized by features such as diagram partitioning, multiple views, and synchronization of different diagrams. More specifically, compilation errors, and the lack of support for a few needed functionalities, which are not properly implemented, oblige the user to manually patch the generated code and to implement the missing functionalities. Such activities are furthermore complicated by the high complexity of the structure of the automatically generated code, and by the poorly documented API of GMF.

GMF also constrains the structure of the meta-model. The worst limitation we found concerns the elements that can be edited in a diagram: they have to belong to a composition hierarchy rooted in the element associated with the editor. We addressed such issue by means of an extensive (and unusual) usage of composition relations, and by adding additional elements, which, as in the previously discussed case of the *SharedPhenomenon* class, increase the complexity of the model.

VII. A PROBLEM FRAME-BASED DEVELOPMENT PROCESS

Currently, the tool described above supports problem frame modeling only in writing syntactically correct diagrams. However, the formal descriptions of requirements and specifications could be easily exploited to verify properties of the model, in particular to prove that the specification of the machine actually satisfies (some of the) requirements when used in the modeled environment. To this end, the PF editor could be used in combination with an event calculus off-the-shelf tool, as in [23]. The resulting process (see Figure 10) would lead to reliable requirements specifications, whose most important properties would have been formally proved.

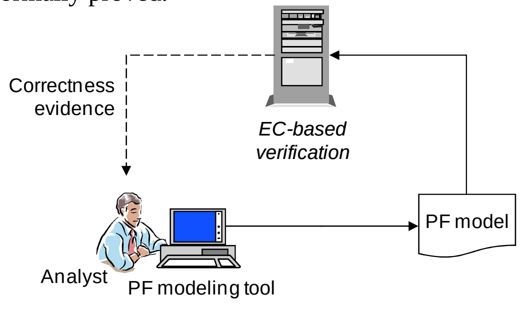

Figure 10. Problem frame editing and verification.

The process described in Figure 10 had already been envisaged by the authors for UML specifications [27], in the context of a whole UML-based development process.

The PF models that result from the modeling and verification activities are the base for the following development phases. Therefore, we need to understand to what extent the valuable information embedded in the PF models can be exploited in the rest of the development process.

The notation used to model problem frames is not suited to support the design and implementation phases, thus we have to translate the PF diagrams into a more implementation-oriented notation. Since the authors have already shown that problem frames can be successfully used in conjunction with UML [5][26], it is quite natural to choose UML as the design notation to be used in combination with PF-based requirements specifications. The usage of UML is also eased by several EMF/UML2 based Eclipse projects (such as ATL [29] and the other transformation engines developed in the context of the Eclipse Model-to-model transformation project [28]) which support the generation of UML models from meta-model instances.

A possible problem frame-based development process is schematically described in Figure 11. The idea is to exploit to the maximum possible extent the knowledge about the environment and the machine embedded in the PF diagrams.

A first step concerns the design phase: problem frames can suggest which architectural structures are best suited for implementing the machine. In [4], Hall et al. show how each problem frame can be implemented with an appropriate design structure, while in our preceding work [6] we show how to use UML and SysML to represent PF models, thus building a starting point for the following design phase.

A PF model is often also useful for understanding the scenarios of the system (especially if complemented with UML sequence diagrams, as in [26]). Scenarios are on their turn strictly connected with testing activities: in fact, in functional testing at least one test case must be written for each scenario. A scenario involves actions, activities and events originated by both problem domains and the machine, which are described in PF diagrams: the latter can thus be used to devise test cases. Moreover, since executing test cases involves exercising some domain behavior, if the test has to be carried out in a laboratory, the problem domain behavior must be simulated: in this case, the PF diagrams provide an accurate specification of the domain behavior to be simulated. Finally, the requirements specify the expected outcome for each scenario, i.e., the oracle of the test case. In conclusion, the PF diagrams contain the whole knowledge needed to define a complete testing environment.

To summarize: PF models can be used to schematically define the software architecture, to provide domain simulators properties, and to derive functional tests cases. A tool able to understand the element constituting a PF model (based on the previously presented meta-model), could generate automatically –through model transformations– three different models: the formal model of the system, used to understand whether the systems fulfills the required proofs of correctness; the design model, used as a starting point to develop the system; the test model, used to verify the implementation.

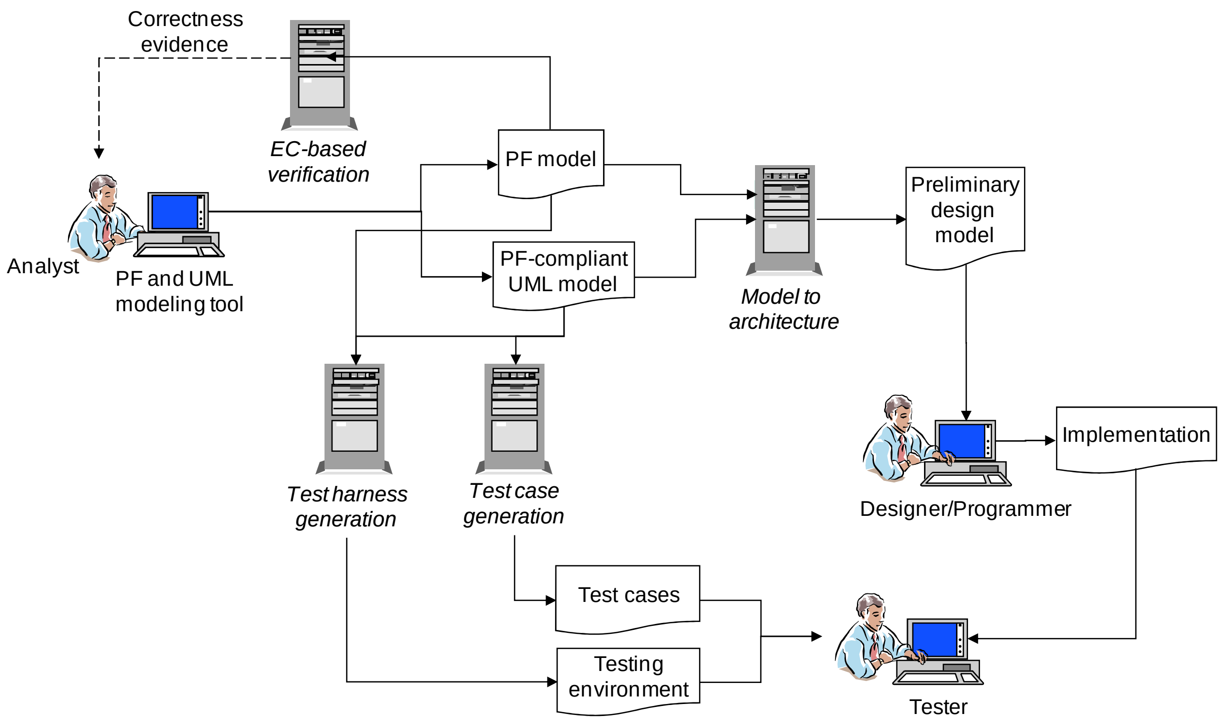

Figure 11. The PF customized development process.

VIII. Related Work

The existing PFs meta-models describe the PF domain with different objectives, which are reflected on the meta-model structure. The meta-model described in [2], [7], [8] is highly detailed, as the one presented in [9]; a less detailed meta-model can be found in [20], while a very concise meta-model is described in [10].

The meta-model presented in [2], [7], [8] describes the main relationships among most of the concepts introduced in Problem Frames methodology. This meta-model suffers from some inadequacies: some of its concepts are exclusively dedicated to represent methodological concepts, such as frame flavors and frame concerns, which we just keep out of the meta-model and out of the tool's responsibilities, leaving them to the user. Another problem (from our specific point of view) with the meta-model is the very fine granularity of the concepts presented, sometimes introducing inheritance hierarchies. Unfortunately, the management of generalization hierarchies is quite cumbersome in GMF. In practice, when working with GMF it is necessary to deal with meta-models that represent the relevant information without employing generalization/ specialization.

The ontology of the Problem Frames proposed in [3], [9] captures even more concepts than the meta-model defined in [2], [7], [8], and it is more abstract, since it does not provide any meta-model (a meta-model is always an ontology, but the vice versa is not guaranteed). From our point of view this ontology presents the same problems as the meta-model introduced in [2], [7], [8]. Moreover, it does not address the specification of behaviors and requirements, which are clearly relevant to the user.

The essential meta-model proposed in [10] is oversimplified: it does not include all the concepts that are expressed in PF diagrams, and some important pieces of information are missing. For instance, the meta-model includes a relation between domains for specifying that the involved domains overlap, but this relation does not indicate which phenomena are shared by the overlapping domains; this information is elsewhere in the model and can be retrieved in a rather complicated way. For these and other similar reasons, the meta-model presented in [10] –which, in fact, was defined to support requirements progression– is not adequate for the construction of tools.

With respect to the approaches to meta-modeling mentioned above, our approach is more pragmatic: on one hand, we strived to provide a synthetic though fairly complete description of the problem frames notation, including a few elements that –although not strictly belonging to the notation– are necessary to support the methodology of PF; on the other hand, we kept the meta-model compliant with the requirements of the EMF/GMF development method. The result was that we were able in a relatively short time to create a prototype of a tool fully supporting the problem frame notation, and well on the way of supporting the PF methodology.

In [20] a work very similar to the one reported here is described: the meta-model is expressed in UML, with added constraints in OCL, and the resulting meta-model is –quite comprehensibly– similar. However, there are also relevant differences. Our meta-model was specifically structured to be used in an EMF/GMF framework: this is reflected in the meta-model itself, e.g., composition relations and decorators are often used instead of inheritance relations. There are also some semantic differences between the two meta-models: the meta-model in [20] does not account for sub-problems and descriptions involving multiple frames. Finally, we have implemented the PF editor: a full working prototype of a PF modeling tool, able to generate UML compatible models.

Another project that exploits the GMF framework is UML2Tools: the Eclipse Ecore UML2 meta-model is used as a basis for building a tool for editing UML2 class, state, component and activity diagrams [19]. Although the goal and the approach of UML2Tools are similar to ours, it does not support a common model shared by the diagram editors (e.g., the editor of class diagrams, the editor of state diagrams, etc.): instead, each editor deals with a distinct instance of the model. In practice the user is not allowed to define a single coherent model: multiple independent diagram-specific models have to be created.

An alternative approach to directly supporting PF notation is to integrate PFs concepts and methodology in the usage of well known modeling languages, like UML and SysML. Such approach has been proposed in [5] and [6].

IX. Conclusions and Future Work

There are several reasons for defining the meta-model of Problem Frames. The first one is that the meta-model helps defining the notation in a precise way; this activity is much needed, since the Problem Frames approach provides essentially methodological guidelines and concepts, but does not precisely define the notation. A second motivation is that the meta-model supports the (semi-automatic) construction of a tool, and tool availability is an essential condition to promote the usage of Problem Frames in industrial software processes. A third motivation is that a precise model (based on a defined meta-model) can be used to automate model transformations, thus feeding other development phases, such as formal verification of the specifications (to prove that the specifications satisfy the requirements), development and test. Finally, a tool based on the meta-model provides a sort of training environment that is compliant by construction with the problem frames approach. Such environment is expected to favor the learning of the PF based requirements analysis techniques, to allow users of the PF approach to evaluate both the tool and the approach, and to stimulate the suggestion of improvements. This paper reports the definition of a meta-model for problem frames that can effectively be used as a basis for the construction of a tool. The proposed meta-model represents all the elements of the PF notation, but leaves the support of a few methodological issues to the initiative of the user. The effectiveness of the meta-model was demonstrated by building a prototype tool with GMF. This activity was also an occasion to evaluate the GMF technology, which appears still rather immature, since a few essential features (such as editing the same subset of elements in two different editors) are neither well supported nor documented.

The main goal of the work reported here was to define a meta-model that could be used as a basis for developing a tool supporting the problem frames technique. While achieving such goal, we put aside a couple of issues that will be object of future work. A first issue concerns the definition of a way to integrate *Descriptions* with the rest of the model: in essence, the issue is that the *text* attribute of *Descriptions* should be connected to the *predicateOn* links to *Phenomena*.

In other words, the occurrence of a phenomenon's name in the text of a description should be recognized as a reference to an instance of *Phenomenon*.

A second important issue involves implementing full-fledged problem composition and decomposition mechanisms, thus testing the ability of the meta-model to support this very relevant part of the problem frames method.

ACKNOWLEDGMENTS

The research presented in this paper has been partially funded by the project "Elementi metodologici per la specifica, la misura e lo sviluppo di sistemi software basati su modelli", funded by the Università degli Studi dell'Insubria.

Massimiliano Bosetti contributed to the development of the prototype tool using the proposed meta-model and approach.

REFERENCES

[1] Jackson, M.: Problem Frames - Analysing and Structuring Software Development Problems. Addison-Wesley ACM Press (2001)

[2] Lencastre, M., Boetlho, J., Clericuzzi, P., and Araújo, J.: A Meta-model for the Problem Frames Approach. In 4th Workshop in Software Modeling Engineering (WiSME'05), Montego Bay, 3 October 2005.

[3] Chen, X., Jin, Z., and Yi L.: An ontology of problem frames for guiding problem frame specification. In: 2nd International Conference-Knowledge Science, Engineering and Management, 2007.

[4] Hall, J.G., Rapanotti, L., and Jackson, M.: Problem frame semantics for software development, Software System Modeling. 4, 189-198 Springer-Verlag (2005)

[5] Lavazza, L. and del Bianco, V.: Combining problem frames and UML in the description of software requirements. In Fundamental Approaches to Software Engineering (FASE 2006), March-April 2006, Vienna.

[6] Colombo, P., Del Bianco, V., Lavazza, L., and Coen-Porisini, A.: A methodological framework for SysML: a Problem Frames-based approach. In 14th Asia-Pacific Software Engineering Conference (APSEC 2007), 5-7 Dec. 2007, Nagoya, Japan.

[7] Lencastre, M., Araujo, J., Moreira, A., and Castro, J.: Analyzing crosscutting in the problem frames approach. Proceedings of the 2006 international workshop on Advances and applications of problem frames, Shanghai, China, ACM, pp. 59--64, 2006.

[8] Lencastre, M., Araujo, J., Moreira, A., and Castro, J.: Towards aspectual problem frames: an example. Expert Systems, vol. 25, pp. 74-86, 2008.

[9] Jin, Z. and Liu, L.: Towards automatic problem decomposition: an ontology-based approach. Proceedings of the 2006 international workshop on Advances and applications of problem frames, Shanghai, China, ACM, pp. 41-48, 2006.

[10] Seater, R., Jackson, D., and Gheyi, R.: Requirement progression in problem frames: deriving specifications from requirements. Requirements Engineering, vol. 12, pp. 77-102, 2007.

[11] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., and Vanderheyden, P.: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. IBM Redbooks, 2004.

[12] Graphical Editing Framework (GEF), http://www.eclipse.org/gef [June 16, 2009]

[13] Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf [June 16, 2009]

[14] Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmf [June 16, 2009]

[15] Object Constraint Language Specification, version 2.0, OMG formal/06-05-01, 2006

[16] OMG, UML Superstructure Specification, v. 2.1.2. formal/2007-11-02, 2007

[17] OMG, MOF 2.0/XMI Mapping, v2.1.1, formal/2007-12-01, 2007

[18] des Rivières, J. and Wiegand, J.: Eclipse: A platform for integrating development tools, IBM Systems Journal, Vol 43, No 2, 2004

[19] Model Development Tools (MDT), http://www.eclipse.org/modeling/mdt/ [June 16, 2009]

[20] D. Hatebur, M. Heisel, and H. Schmidt: A Formal Metamodel for Problem Frames. Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, Toulouse, France: Springer-Verlag, pp. 68-82, 2008.

[21] P. Colombo, V. del Bianco, L. Lavazza, A. Coen-Porisini, "Towards a Meta-model for Problem Frames: Conceptual Issues and Tool Building Support", *The 4th Int. Conf. on Software Engineering Advances – ICSEA 2009,* September 20-25, 2009 - Porto, Portugal.

[22] Thein Than Tun, Tim Trew, Michael Jackson, Robin Laney and Bashar Nuseibeh: Specifying features of an evolving software system, Software Practice and Experience 2009; 39.

[23] Classen A, Laney R, Tun TT, Heymans P, Hubaux A.: Using the event calculus to reason about problem diagrams, Proceedings of the 3rd International Workshop on Applications and Advances of Problem Frames, Leipzig, 10 May 2008, ACM, 2008.

[24] Kowalski R, Sergot M.: A logic-based calculus of events. New Generation Computing 1986; 4(1).

[25] Miller R, Shanahan M.: The event calculus in classical logic—alternative axiomatisations. Journal of Electronic Transactions on Artificial Intelligence 1999; 3.

[26] Del Bianco, V., Lavazza, L., Enhancing Problem Frames with Scenarios and Histories in UML-based software development, Expert Systems – The Journal of Knowledge Engineering, Special issue on applications and advances in problem frames, February 2008 - Vol. 25 n. 1 Pag. 28-53 – Blackell publishing.

[27] Del Bianco, V., Lavazza, L., Mauri, M.: Model Checking UML Specifications of Real-Time Software, The Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland, 2–4 December, 2002.

[28] M2M, http://www.eclipse.org/m2m/

[29] Frédéric Jouault and Ivan Kurtev, "Transforming Models with ATL", in Satellite Events at the MoDELS 2005 Conference, Springer LNCS, Vol. 3844/2006.