Enhancing Problem Frames with Scenarios and Histories: a Preliminary Study

Vieri Del Bianco
CEFRIEL
Via Fucini, 2
20133 Milano (Italy)
+39 02 66804978
delbianc@cefriel.it

Luigi Lavazza
University of Insubria and CEFRIEL
Via Mazzini, 5
21100 Varese (Italy)
+39 0332 218930
luigi.lavazza@uninsubria.it

ABSTRACT
Problem frames are a very interesting approach to requirements modelling that is gaining increasing attention and popularity. This paper reports a few preliminary investigations concerning the possibility of enhancing the problem frames methodology with concepts derived from requirements modelling techniques based on scenarios and histories. The goal of this research is to make problem frames even more appealing for the software developers, who are generally familiar with the ideas underlying scenario-based modelling.

The results presented here are encouraging: a well known problem, often used to illustrate problem frames (the sluice gate control) was studied with the help of scenarios, modelled by means of histories. Scenarios were also employed to support the merging of sub-problems. In these activities the proposed approach was quite helpful. Nevertheless, some research is still needed to adequately explore the actual applicability and value of the proposed approach.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications – languages, methodologies.

General Terms
Design, Languages.

Keywords
Problem frames, scenarios, histories.

1. INTRODUCTION
The Problem Frames approach [1] has the potential to dramatically improve the early lifecycle phases of software projects. Problem frames (PF) drive developers to understand and describe the problem to be solved, which is crucial for a successful development process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWAAPF’06, May 23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005...$5.00..

Nevertheless, PF are not (yet) as popular as they deserve in industrial software development processes. On the contrary, more intuitive approaches, like those based on the description of scenarios by means of sequences of events (also known as histories) are much more widely employed. This kind of approach dues its success mostly to the fact that it is rather intuitive; by the way it must be observed that also formal notations, based on temporal logic, allow the modeller to specify sequences of events and states (i.e. specific evolutions in time of the system); these sequences actually describe instances of histories. Moreover, some formal methods are able to check whether a given history is compatible or not with a given model. Assuming that every history represents a sequence of events that is (or is not) acceptable for the system, history checking can be used to test the given model.

In general a set of histories is not a complete description of a problem domain or of user requirements; therefore it is normally not possible to conclude that a model is correct only on the basis that it rejects unacceptable histories and allows acceptable ones. For this reason, it is not advisable to base requirements modelling exclusively on scenario description.

The basic idea investigated in the paper is to employ scenario modelling based on histories in conjunction with problem frames, histories being given in terms of domain phenomena. Histories are expected to increase the intuitiveness of PFs, which are responsible for providing a more rigorous requirements modelling framework than scenario-based modelling.

The paper is organized as follows: the proposed approach is described in section 2, by means of a case study. Section 3 sketches how the proposed approach can be employed in the development phases. Tool support is then briefly discussed. Finally some conclusions are drawn.

2. PROBLEM FRAMES AND SCENARIOS
In order to illustrate the applicability of scenario-based modelling in conjunction with problem frames, in this section we model the well known problem of the sluice gate control. The problem is defined as follows [1]. 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 of 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 at the top
and bottom of the gate travel; at the top it is fully open, at the bottom it is fully shut. The connection to the computer consists of four pulse lines for motor control, two status lines for gate sensors, and a status line for each class of operator commands. The PF diagram for the sluice gate problem is reported in [1].

Figure 1. The sluice gate commanded behaviour frame

2.1 Sluice gate control

In this section several scenarios concerning the sluice gate control are reported.

2.1.1 Scenario 1

A typical scenario is the following:

1. The gate is shut.
2. The operator issues the command to raise (open) the gate.
3. The control machine activates the motor.
4. After a while the gate is open.
5. The control machine stops the motor.
6. The gate is still in the open position.

The scenario can be formally written as follows:

\[
\begin{align*}
&\text{GM!Shut}(t_0) \\
&\text{SO!Raise}(t_1) \\
&\text{SC!Clockw}(t_2) \\
&\text{SC!On}(t_3) \\
&\text{GM!Rising}(t_4) \\
&\text{GM!Top}(t_5) \\
&\text{SC!Off}(t_6) \\
&\text{GM!Open}(t_7)
\end{align*}
\]

where \( t_{i+1} \geq t_i \), \( \forall i \), and no phenomenon occurs at any \( t \) such that \( t_i < t < t_{i+1} \).

The fact of explicitly ordering phenomena in time, and in particular the association of every phenomenon occurrence with a time instance, suggests the opportunity of specifying constraints on time delays. For instance, the following relations are expected to hold among the time of occurrence of the phenomena in the sluice gate control PF.

\[
\begin{align*}
t_2-t_1 &= \text{SC_reaction_time} \\
0 &< t_3-t_2 < d \ (d \ being \ the \ maximum \ time \ needed \ by \ the \ controller \ to \ issue \ two \ consecutive \ command \ towards \ the \ Gate\&Motor) \\
t_4-t_3 &= \text{GM_reaction_time} \\
t_5-t_4 &= \text{Sluice_opening_time} + \text{GateSensor_reaction_time} \\
t_6-t_5 &= \text{SC_reaction_time}
\end{align*}
\]

Figure 2. The revised sluice gate PF

In this case, timing relationships are quite important. In fact, the behaviour of the system is determined by the stop command being issued before the gate is completely open (i.e. \( t_5-t_4 = \text{Sluice_opening_time} \)).

All the other constraints remain the same as in the scenario 1.

Note that GM!Rising is actually a state (as is GM!Open and all the phenomena in b), therefore \( t_4 \) is the time when the Gate enters the Rising state: the state will hold until \( t_7 \).

Timing constraints can also be useful to reason about the performance of the system. In the case above, \( t_5-t_1 \) indicates the time required to open the gate. This can be checked against the needs of the user, in order to understand if the system is fast enough. If it is not, the timing constraints indicate how to make the system faster: it is possible to shorten the reaction time of the machine or –more likely– to get a more powerful motor for the gate.

2.1.2 Scenario 2

Let us now consider a second scenario:

1. The gate is shut.
2. The operator issues the command to raise (open) the gate.
3. The control machine activates the motor.
4. After a while –before the gate is completely open– the operator issues a stop command.
5. The control machine stops the motor.
6. The gate is still in an intermediate position.

The scenario can be formally written as follows:

\[
\begin{align*}
&\text{GM!Shut}(t_0) \\
&\text{SO!Raise}(t_1) \\
&\text{SC!Clockw}(t_2) \\
&\text{SC!On}(t_3) \\
&\text{GM!Rising}(t_4) \\
&\text{SO!Stop}(t_5) \\
&\text{SC!Off}(t_6) \\
&\text{GM!Still}(t_7)
\end{align*}
\]

In this case we need the new phenomenon GM!Still in order to represent the effect of the SC!Off command. In fact, none of the phenomena belonging to the set b given in Figure 1 represents correctly the state of the gate at the conclusion of the scenario. This problem can be solved quite easily by inserting “Still” in the set of possible states of the Gate & motor domain, as shown in Figure 2.
2.1.3 Scenario 3
Let us now consider a third scenario

1. The gate is shut.
2. The operator issues the command to lower (shut) the gate.
3. The control machine –which “knows” the state of the gate– does nothing.
4. The gate is still in the shut position.

The scenario can be formally written as follows:

\[
GM!Shut(t_0) \\
SO!Lower(t_1) \\
GM!Shut(t_2)
\]

In this case it is essential to specify the time constraint \( t_2-t_1 > SC\_reaction\_time \).

In fact, for any \( t_2 \) such that \( t_2-t_1 \leq SC\_reaction\_time \), the gate is always shut at \( t_2 \) (see also the previous scenario).

Actually, the sequence of phenomena here is meant to imply that no SC!On occurs in the \( t_1-t_2 \) interval. This is a rather indirect (hence not very intuitive) way of specifying the behaviour of the system. A reasonable alternative is to impose that the following sequence is prohibited:

\[
GM!Shut(t_0) \\
SO!Lower(t_1) \\
SC!Clockw(t_2) \text{ or } SC!Anti(t_2) \\
SC!On(t_3)
\]

Also prohibited is the following

\[
GM!Shut(t_0) \\
SO!Lower(t_1) \\
SC!On(t_2)
\]

Note: UML 2.0 [2] has introduced negative traces (i.e. prohibited sequences), therefore we expect that their use will become popular in the near future.

2.1.4 Scenario 4
We now consider the following sequence:

1. The gate is shut.
2. The operator issues the command to raise (open) the gate.
3. The control machine activates the motor.
4. After a while –before the gate is open– the operator issues a lower (close) command.
5. The control machine first stops the motor, then inverts the movement, and finally restarts the motor.

The scenario can be formally written as follows:

\[
GM!Shut(t_0) \\
SO!Raise(t_1) \\
SC!Clockw(t_2) \\
SC!On(t_3) \\
GM!Rising(t_4) \\
SO!Lower(t_5) \\
SC!Off(t_6) \\
GM!Still(t_7) \\
SC!Anti(t_8) \\
SC!On(t_9) \\
GM!Falling(t_{10})
\]

The following timing relations are expected to hold:

\[
\begin{align*}
t_5-t_4 &< \text{Sluice\_opening\_time} \\
t_6-t_5 & = SC\_reaction\_time \\
t_7-t_6-t_0 & = GM\_reaction\_time \\
0 &< t_8-t_6 < d \\
0 &< t_9-t_8 < d
\end{align*}
\]

It is interesting to note that –thanks to the last two constraints– GM!Still could actually follow SC!Anti, and even SC!On. These scenarios should be included in the possible ones. From the point of view of expressiveness, it would be preferable to be able to have a single scenario definition with variants, so that the three cases can be all represented.

A constraint on \( t_9-t_6 \) (e.g., the motor has to stay still for a while between subsequent movements in opposite directions) could be expressed quite easily, if needed.

2.1.5 Scenario 5
The scenario is meant to specify what happens if the operator generates a sequence of commands that does not make sense.

A sequence that is representative of the “bad” behaviour of the operator is as follows:

1. The gate is shut.
2. The operator issues the command to raise (open) the gate.
3. Immediately after such command, the operator issues very fast a sequence of random commands.
4. The control machine ignores all the commands except the first one.

This type of scenario can be formally written as follows:

\[
GM!Shut(t_0) \\
SO!Raise(t_1) \\
SO!Lower(t_2) \\
SO!Stop(t_3) \\
SO!Lower(t_4) \\
SC!Clockw(t_5) \\
SC!On(t_6)
\]

Where the following timing relations hold:

\[
\begin{align*}
t_2-t_1 &< SC\_reaction\_time \\
t_3-t_1 &< SC\_reaction\_time \\
t_4-t_1 &< SC\_reaction\_time \\
t_5-t_1 & = SC\_reaction\_time
\end{align*}
\]

The scenario indicates the behaviour of the machine in response to a sequence of commands too close to each other. In this case the controller ignores the commands that are too close to the first one. Alternatively, it could be possible to specify that the commands are buffered and executed as soon as it is reasonable and safe (for the mechanical parts) to do it.

2.2 Sluice gate monitoring
Scenario 3 showed that the operator can issue “useless” commands. Although the sluice controller can filter these commands (as well as potentially harmful ones, as shown in scenario 5), it is advisable to inform the operator of the state of the gate and motor, so that he/she can evaluate what commands are appropriate at any time. For instance, if the operator is
informed that the gate reached the fully open position, the operator probably won’t attempt to further raise the gate, even if the Sluice Controller permits such an action.

The problem of informing the operator of the state of the gate and motor can be modelled by means of an information display PF, as shown in Figure 3.

Figure 3. The sluice gate monitoring PF
A possible scenario concerning the sluice gate monitoring is the following.

GM!Shut(t0)
GM!Rising(t1)
SM!MovingUp(t2)
GM!Still(t3)
SM!Still(t4)
GM!Rising(t5)
SM!MovingUp(t6)
GM!Open(t7)
SM!FullyOpen(t8)

The only timing constraints are that

t2-t1 = t6-t5 = t8-t7 = SM_reacttion_time.

Note that the phenomena in set b (i.e. GM!{Open, Shut, Rising, Falling, Still}) are states. The sluice monitoring machine actually needs to react to state changes. Therefore we can consider the phenomena in set a as events, even though they are named as the states in b.

SM!{MovingUp, MovingDown, Still, FullyOpen, FullyShut} are events that inform the operator of the changes in the state of the gate. In practice it could be preferable that these events trigger states of a display interface (e.g., every event switches on a lamp corresponding to a state). However, the presence of such an interface does not change the essence of the problem.

The two PFs shown in Figures 2 and 3 represent projections of the unique problem of monitoring and controlling the sluice gate. Therefore they have to be merged in order to represent the whole problem. When merging the problem projections it is interesting to understand whether scenarios merge as well. This issue is explored in next section.

2.3 Sluice gate monitoring and control
The result of merging the PFs shown in Figures 2 and 3 is reported in Figure 4.

<table>
<thead>
<tr>
<th>Scenario from Sluice control problem</th>
<th>Scenario from Sluice monitoring problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>The gate is shut.</td>
<td>The gate is shut (this is a state: it lasts until the next state change).</td>
</tr>
<tr>
<td>The operator issues the raise command.</td>
<td></td>
</tr>
<tr>
<td>The control machine activates the motor.</td>
<td></td>
</tr>
<tr>
<td>The gate starts opening (new state: gate opening).</td>
<td></td>
</tr>
<tr>
<td>The monitor reports that the gate is moving up.</td>
<td></td>
</tr>
<tr>
<td>Before the gate is open the operator issues a stop command.</td>
<td></td>
</tr>
<tr>
<td>The control machine stops the motor.</td>
<td></td>
</tr>
<tr>
<td>The gate stops in an intermediate position (new state: gate still).</td>
<td></td>
</tr>
<tr>
<td>The monitor reports that the gate is still.</td>
<td></td>
</tr>
<tr>
<td>The monitor reports that the gate is still.</td>
<td></td>
</tr>
</tbody>
</table>

The merging of the histories representing the two scenarios is represented in Table 1.
The resulting integrated scenario can be described as follows (phenomena from the monitoring problem are in italics, phenomena common to the two problems are underlined):

- **GM!!Shut**
- **SO!!Raise**
- **SC!!Clockw**
- **SC!!On**
- **GM!!Rising**
- **SM!!MovingUp**
- **SO!!Stop**
- **SC!!Off**
- **GM!!Still**
- **SM!!Still**

... Table 1 is obtained considering the cause-effect relationships. For instance “The gate starts opening (new state: gate opening)” follows “The control machine activates the motor” because the opening is caused by the activation of the motor. However, the resulting ordering of events is only partial, since there are pairs of events that are not related by a cause-effect relationship. For instance, the event “Before the gate is open the operator issues a stop command” is neither related to “The control machine activates the motor” nor to “The monitor reports that the gate is moving up”. In this case “Before the gate is open the operator issues a stop command” and “The monitor reports that the gate is moving up” could occur in any order. It is therefore advisable to analyse time constraints in order to understand what are the possible ordering of these events and of the events that are caused by them.

The analysis proceeds as follows (note that phenomena names are used).

- **GM!!Rising** (event “The gate starts opening”) occurs at T0, therefore **SM!!MovingUp** (event “The monitor reports that the gate is moving up”) occurs at T0+SM_reaction_time.
- **SO!!Stop** (event “Before the gate is open the operator issues a stop command”) occurs at T0+k (k being unbound). **SC!!Off** (event “The control machine stops the motor”) occurs at T0+k+SC_reaction_time, and **GM!!Still** (event “The gate is still in an intermediate position”) follows at T0+k+SC_reaction_time + GM_reaction_time.

Therefore, **SM!!MovingUp** will happen after **GM!!Still**, if SM_reaction_time > k + SC_reaction_time + GM_reaction_time. This is not very good, since the operator would be informed that the gate is moving after the gate has already stopped again. The good news is that scenario analysis can reveal the possibility that the merging produces also such a rather surprising scenario. In this way the designer of the system is warned that employing a slow monitoring machine could invalidate the information delivered to the operator.

The example above showed that the practice of accompanying problem frames with scenarios works also when modelling sub-problems and when they are then merged into a unique problem description. The approach shown is applicable in general; provided that scenarios are specified as histories of phenomena. In fact, it is our opinion that scenarios can be helpful in this type of activity. When two PFs are merged, their scenarios also have to merge: if they do not, this is a warning that there is some problem with the composition of the subproblems.

### 3. METHODOLOGY CONSIDERATIONS

Scenario descriptions can be employed in two ways, corresponding to different phases of the software development:

- Scenarios can be employed to support requirement definition (i.e. they can be used to build a model of the problem domain and of the user requirements).
- Scenarios can provide the starting point for the design activities.

The following sections explore these two possible ways of exploiting scenarios in the context of problem frames.

#### 3.1 Support for requirements modelling

##### 3.1.1 Scenarios support Problem Frames

Scenarios can be employed to support requirement definition (i.e. elements and characteristics of both the environment and the user requirements can be identified by analysing a representative set of scenarios). Here we do not go into a detailed discussion of this issue. Rather, we report some observations that support the claim.

1. History elements are domain phenomena. Every element of a history must be an event generated in a domain or a domain state. Therefore, writing histories helps in identifying domains and their characteristics. As an example, consider Scenario 1: the second element of the history “The operator issues the command to raise (open) the gate” brings immediately to the identification of the domain “Sluice Operator” and of the phenomenon “Raise command” (SO!!Raise). Of course it is possible that most domains and phenomena have already been identified and described when scenarios are investigated. However also in these cases scenarios can help, as demonstrated by Scenario 2, which suggested that state “Still” should be added to the description of domain “Gate & motor”.

2. It is often the case that two subsequent phenomena are linked by a cause-effect relationship. If the phenomena belong to different domains, then the former is a shared phenomenon. For instance, in scenario 1, **SC!!Off** follows **GM!!Top**. A little of analysis shows quite easily that **GM!!Top** is a shared phenomenon between the machine and the Gate & motor domain. The same procedure let us conclude that also **SC!!Off** is a shared phenomenon.

3. Sub-sequences that appear in several histories suggest rules of behaviour. For instance, the pair (**GM!!Shut**, **SO!!Raise**) is usually followed by the sequence **SC!!Clockw**, **SC!!On**, **GM!!Rising**. This suggests that the system model should include the following rules: a) whenever the gate is shut and the raise command is issued, the machine issues the Clockw and On commands; b) as a consequence of Clockw and On commands the gate starts rising.

As a complement to the previous considerations, it can be observed that it is possible to classify scenarios according to the phenomena they address. The scenarios reported in section 2 consider phenomena of all domains. It could be possible to restrict the attention to the phenomena concerning a single domain; for
instance, histories made of phenomena concerning only the gate & motor could provide a description of the problem domain.

3.1.2 Problem Frames support scenarios
Scenarios are usually described as sequence diagrams. The latter include object instances and message passing. This practice has an inherent problem: you can coherently use object instances and message passing only if you have already described a set of classes, this description being the result of the same analysis to which scenarios are expected to contribute. This problem is generally solved by interleaving scenario description and class definition. Nevertheless, class diagrams tend to be quite detailed, and too less agile to support the description by means of scenarios (e.g., you have to stop too often to detail classes). In the context of “traditional” UML practice, some methodologies (like the “Robustness analysis”) have been introduced in order to provide a lightweight preliminary class description needed to support scenario-based modelling. In the case of the robustness analysis a set of stereotyped classes (interface, control, and entity classes) is employed.

We propose PF diagrams as a way to provide a context for scenarios. With respect to class diagrams PFs have the advantage of being more lightweight, while they are more rigorous than the “lightweight” descriptions proposed by robustness analysis and similar proposals. It is still true that the analyser should develop the PFs and the scenarios in parallel.

3.2 Support for design
Scenario descriptions and use cases, possibly modelled graphically with sequence diagrams and state diagrams a la UML, are generally considered a good starting point toward the actual design, implementation and test of the software system.

The scenario descriptions presented in this paper correspond to a high level description of the system requirements and specifications, at the same level of abstraction of black-box system-level user goal use cases described in [3]. For instance, the actors participating to use cases can be mapped to domains, the only notable exception being lexical domains.

A use case can generally be characterized by means of one or more scenarios, possibly including fault scenarios (where errors and unusual conditions are addressed) and negated scenarios (i.e. scenarios that should not happen). The same applies to scenarios in the context of PFs.

The decomposition of a problem into smaller problems (problem frame projections) does not invalidate the analogy with the system-level use cases (see section 2.3).

Once problem frames have been enriched with a set of scenarios, we have the same level of information, or better, than provided by classic use cases; the path to code can then be shaped on the basis of one of the current methodologies based on use cases and scenarios.

To name just a few, the Unified Process ascribes to use cases a major role in capturing functional requisites and proposes them as the starting point to development [5]. Formal scenarios and sequence diagrams can be used to generate and synthesize the skeleton code of the system; a recent proposition of this consolidated concept can be found in [6]. Indeed it is possible to use sequence diagrams as a starting point to synthesize state diagrams of the system under development, as demonstrated in [7]. On the other side of the spectrum, scenarios, use cases and histories are used in agile development processes; the information provided by scenarios and problem frames is wider than what is needed from most agile processes (Kent Beck's eXtreme Programming being one of the lightest) [8][9].

An interesting theme to investigate would be how to integrate problem frames with IID (Iterative and Incremental Development) Processes, which assumes that requisites as well as most of the other artefacts of the system should be defined, built and/or refined in incremental steps (or even continuously).

Finally, there exist approaches to development that are tailored directly on problem frames, based on UML's sequence diagrams and use cases [3], or with an explicit mapping to architectural patterns [4].

4. TOOL SUPPORT

Tool support is essential for the practical applicability of the approach in industrial settings.

Since there is little or no tool support for problem frames, here we do not discuss support for the requirements modelling approach integrating PFs and scenarios. Instead, we concentrate on the existing support for scenarios alone.

Support for modelling scenarios is provided by several tools developed as part of well-known methodologies. Among these are the tools supporting UML sequence diagrams and SDL message sequence charts. More interesting are facilities provided by formal methods: quite noticeable is the possibility of checking histories against system models (i.e. some tools allow the user to verify that a given history is compatible, or incompatible, with a given specification). In this way it is possible to test the correctness of the specification with respect to the scenarios that must, or must not, be allowed by the system. History checking of TRIO specifications [11] is an example of such techniques.

5. CONCLUSIONS

The main goal of this paper is to start exploring the possibility of employing scenario modelling based on histories in conjunction with problem frames, in order to increase the intuitiveness of PFs, thus making their usage more appealing to professional software developers.

A few preliminary investigations concerning the proposed approach have been reported. The results presented here are encouraging, and tend to support the idea that integrating scenarios into PFs is a viable practice.

More research is needed to assess the full potential of the proposed approach, and to test its pros and cons in software development activities of non trivial size and complexity.

6. REFERENCES


