Software and Systems Modeling Volume 4 Number 2, pages 189-198, May 2005

Problem frame semantics for software development

Jon G. Hall Lucia Rapanotti Michael Jackson
The Open University
Milton Keynes
MK7 6AA
{J.G.Hall, L.Rapanotti, M.Jackson}@open.ac.uk

Abstract

This paper presents a framework for understanding Problem Frames.

The semantics places Problem Frames within the framework for Requirements Engineering of Zave and Jackson, and its subsequent formalization in the Reference Model of Gunter et al. It distinguishes between problem frames, context diagrams and problem diagrams, and allows us to formally define the relationship between them as assumed in the Problem Frames framework.

The semantics of a problem diagram is given in terms of 'challenges', a notion that we also introduce. The notion of a challenge is interesting in its own right for two reasons: its proof theoretic derivation leads us to consider a challenge calculus that might underpin the Problem Frame operations of decomposition and recomposition; and it promises to extend the notion of formal refinement from software development to requirements engineering.

In addition, the semantics supports a textual representation of the diagrams in which Problem Frames capture problems and their relationship to solutions. This could open the way for graphical Problem Frames tools.

Keywords

Requirements Engineering, Problem Frames, Semantics, Reference Model, Framework

1 Introduction

Problem Frames [8, 9] classify software development problems. In their intention they are akin to design patterns [4, 10], but for describing problems, not solutions. Problem Frames structure the world in which the problem is located—the *problem domain*—and describe what is there and what effects a system located there must achieve. By emphasising problems rather than solutions, Problem Frames can exploit the understanding of a problem class, allowing a problem owner with specific domain knowledge to drive the Requirements Engineering process. The graphical notation is helpful here for communication between the software developer and the problem owner.

However, a primarily graphical basis has disadvantages [14]. In particular:

- it is hard to interpret graphical artefacts precisely, and misunderstandings can easily occur;
- it is hard to determine whether a description is complete and correct;
- it is hard to identify equivalent structures that could be used interchangeably.

Other graphical notation communities have found that a formal semantics contributes to a solution, see [7, 11] and papers in [3] for examples. The semantics presented in this paper could serve this purpose for Problem Frames. To the best of our knowledge, a semantics of Problem Frames is missing from the literature, although a partial formal

characterization of some early work on Problem Frames is given in [1].

Our semantics places Problem Frames within Zave and Jackson's framework for Requirements Engineering [15]. In particular we focus on the relationship between Problem Frames and expressions of the form

$$K, S \vdash R$$

(where K is a description of the problem domain, S is the specification of the solution, and R is the problem requirement). This relationship connects the Problem Frames framework to the Reference Model for requirements and specifications given by Gunter *et al* [5] and more recently, Hall and Rapanotti [6], in which control and other interaction properties are given expression.

The semantics has many interesting characteristics. For instance:

- It represents the structure of a Problem Frames artefact at a level above the detail of domain descriptions; for instance, it can accommodate the description of an 'operator' domain's behaviour as

  'The operator smokes'.

  In this sense, our representation of the Problem Frame artefacts comes from the tradition of the propositional calculus: we represent and manipulate only the structure above the level of domain descriptions. Of course, as with the propositional calculus—from which the predicate calculus is reached through the formalisation of propositions—so too, in principle, our semantics is extensible wherever more formal domain descriptions exist.

- We define a formal syntax for Problem Frames, underpinning the semantics and anchoring the graphical notation. This syntax opens the way for graphical Problem Frames tools and for electronic transformation and communication of Problem Frames artefacts.

- The meaning of a problem diagram is given in terms of 'challenges'—a formal notion that abstracts the solution S from $K, S \vdash R$ tuples. The notion of a challenge is interesting in its own right for at least two reasons. Its proof theoretic derivation leads us to consider a challenge calculus that might underpin the Problem Frame operations of decomposition and recomposition, as well as the associated correctness arguments; and it promises to extend the notion of (formal) refinement from program development through to requirements engineering.

- The formal semantics for Problem Frames provides a clear overall structure for artefacts and their relationships within a Problem Frames development. Although persuasive arguments for the soundness of a development can be based solely on its rigour or formality, a clear overall structure is always the essential foundation. Without a clear overall structure, even a collection of perfectly formal and convincing arguments at the detailed level will not add up to a demonstration that a development is sound.

This paper is organised as follows. Section 2 outlines the PF (Problem Frames) framework. A comprehensive introduction to PFs is beyond the scope of this paper and can be found in [9]. Section 3 recalls the Reference Model of [5]. Section 4 introduces the semantics. Finally, Section 5 briefly comments on the nature of the semantics and indicates the direction of future work.

2 The Problem Frames Framework

In this section we review some of the basic elements of the PF (Problem Frames) framework. To focus our review we describe the PF development of the following problem, a Chemical Reactor Controller:

> A computer system is required to control the catalyst unit and cooling system of a chemical reactor. An operator issues commands for activating or deactivating the catalyst unit; in response to such commands, the system instructs the

unit accordingly and regulates the flow of cooling water. A gearbox is attached to the system: whenever the oil level in the gearbox is low, the system should ring a bell and halt execution.

This problem is a simplified version of a chemical reactor described in [2, 12].

2.1 Problem Diagrams

Within the PF framework, a *problem diagram* defines the 'shape' of a problem by capturing the characteristics and interconnections of the parts of the world it is concerned with. A problem diagram also includes the requirements that constrain the relationships between these parts. The problem diagram for our Chemical Reactor Controller problem is shown in Figure 1.

The components are:

- A double-barred box (*Operation Machine*): the *machine domain*, i.e., the software system to be built together with its underlying hardware.

- A box (the *Operator* domain): the human operator. Human operators are regarded as *biddable* in the problem frames framework: they may obey stipulated procedures, but not reliably, and may generate events spontaneously.

- Other boxes (*Cooling System*, *Catalyst*, etc.): *given domains* representing parts of the world that are relevant to the problem. Each of these domains is *causal*: i.e., its phenomena are physical events and states, and are causally related;

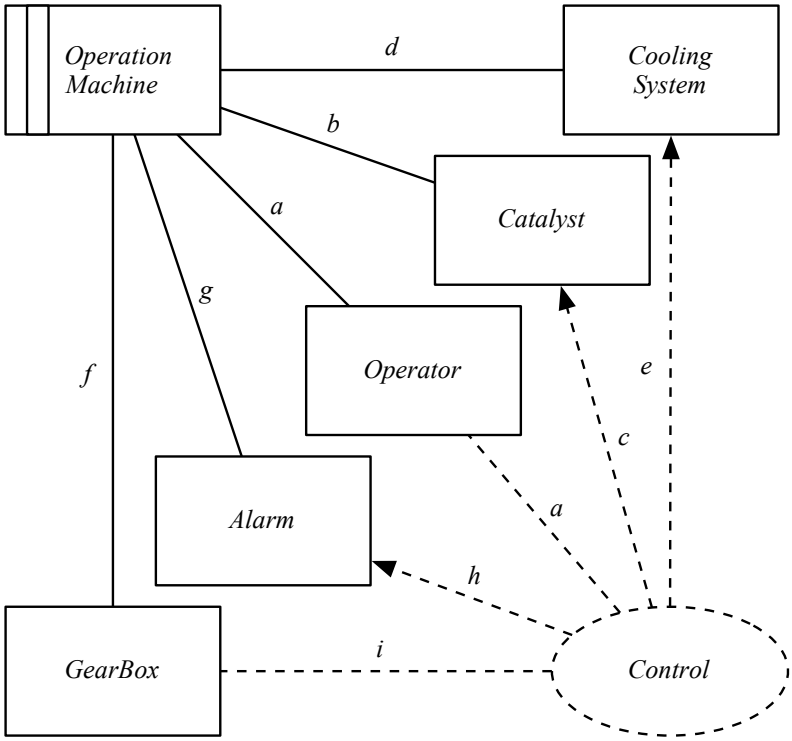
- A dotted oval (*Control*): the *requirement*: i.e., the condition in the problem domain that the Operation Machine must guarantee to qualify as a solution to the problem.

- The annotated connections between domains: these indicate *shared phenomena*, including events, operations and state information. In Figure 1, for example, the connection between the *Operation Machine* and the *Cooling System* is annotated by the set d containing the two phenomena $increase\_water_{act}$ and $decrease\_water_{act}$. These represent commands generated by the Operation Machine in order to increase or decrease the water level in the Cooling System. Phenomena shared between two domains are *observable* by both, but *controlled* by one of them only. For instance phenomena $increase\_water_{act}$ and $decrease\_water_{act}$ are both controlled by the *Operation Machine*: this is indicated by an abbreviation of the domain name followed by *!*, in this case *OM!*.

- The annotated connections between the requirement and the domains: a dashed line indicates that the requirement *references* its phenomena, while a dashed arrow indicates that the requirements *constrains* the phenomena. For instance, the *oil_level* in the gear box is referenced by the requirement, while the *bell_ringing* is constrained.

As well as biddable and causal, domains can also be lexical; a lexical domain reifies a data structure, in some cases a text. The phenomena of biddable domains are events; those of causal domains are states and events (generalised as causal phenomena); and those of lexical domains are symbolic.

Not included in the diagram, but an essential part of the problem definition, are the descriptions of all given domains and the details of the requirement. Typically, as the development progressed, detail of a machine specification would be added. All of these descriptions would be recorded elsewhere, rather than added to the diagram. An important point is that the framework does not prescribe the notation to be used for these descriptions; formal and informal descriptions are equally acceptable, provided they are precise enough to capture the characteristics of interest of the various respective domains. This freedom motivates the propositional nature of our semantics: the satisfaction of a description, for example by a domain, can be abstracted as a proposition.



$a : OP!\{open\_catalyst, close\_catalyst\}$
$b : OM!\{open\_catalyst_{act}, close\_catalyst_{act}\}$
$CA!\{is\_open_{sen}, is\_closed_{sen}\}$
$c : CA!\{open, closed\}$
$d : OM!\{increase\_water_{act}, decrease\_water_{act}\}$
$CS!\{is\_rising_{sen}, is\_falling_{sen}\}$
$e : CA!water\_level$
$f : GB!request\_service$
$g : OM!ring\_bell$
$h : AL!bell\_ringing$
$i : GB!oil\_level$

Figure 1. The Chemical Reactor Problem Diagram

The PF framework distinguishes two fundamental roles of descriptions:

- *indicative*, expressing what is given or assumed to be true, and
- *optative*, expressing what is required or desired to be true.

Descriptions of given domains, then, play an indicative role, while the requirement description and machine specification are optative.

Also provided separately from the diagram are the *designations* of phenomena, which ground the problem's vocabulary in the physical phenomena. For instance, through their designations, *open_catalyst* and *close_catalyst* will denote physical command actions that the operator can actually perform; is_rising_{sen} and $is\_falling_{sen}$ will denote states that can be captured by sensors.

2.2 Problem Frames

One of the aims of the PF framework is to identify basic classes of problem that recur throughout software development. [9] identifies five such *basic* problem classes, or *problem frames*, including the *commanded behaviour frame*, the *required behaviour frame* and the *information display frame*, which we use here (or later) for illustration. A problem frame acts as a template for recognising problems in its class. In form a problem frame is simply a problem diagram, annotated with *domain and phenomena markings*, and associated with a correctness argument template. To match a problem frame template, a problem must:

- match the problem topology—as captured in the problem frame diagram;
- match the domain characteristics—as captured by the domain markings in the problem frame diagram (B for biddable, C for causal, and Y for lexical);
- match the characteristics of the shared phenomena—as captured by the markings on the connections (E for events, C for causal and X for symbolic); and
- support a way of building and discharging a correctness argument which matches the correctness argument template for the problem class. For the commanded behaviour frame, the form of the argument will exploit explicitly stated causal properties of the controlled domain to show that the machine behaviour in terms of the phenomena E4, C1 and C2 will cause the required behaviour of the controlled domain in terms of the phenomena C3. The reader interested in the detail of the correctness argument template should consult [9].

The commanded behaviour frame is shown as the annotated problem diagram in Figure 2.

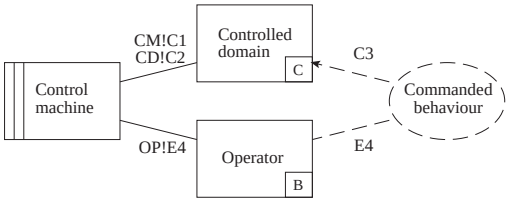


Figure 2. The Commanded Behaviour frame

By comparing Figure 2 and Figure 1, the reader will see that the commanded behaviour frame matches the projection of the problem diagram consisting only of these parts:

- The *Operation Machine*, matching the *Control Machine* in the frame.
- The causal *Catalyst*, matching the *Controlled domain* in the frame.
- The biddable *Operator*, matching the *Operator* in the frame. The operator controls and issues commands spontaneously as event phenomena, shared with the *Control machine*. These event phenomena are *open_catalyst* and *close_catalyst*.
- The requirement determining the response of the Controlled domain to the Operator's commands. The required behaviour is expressed in terms of the causal phenomena is_open_{sen} and is_closed_{sen}.

2.3 Problem Decomposition

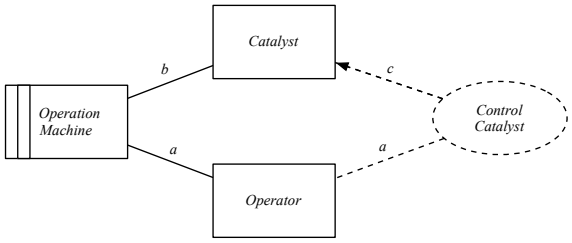
Most real problems are too complex to fit basic problem frames. They require, rather, decomposition into a collection of interacting sub-problems, each of which is smaller and simpler than the original. The projection of the problem diagram of Figure 1 that matches the commanded behaviour frame represents one such sub-problem.

Within the PF framework, we identify sub-problems from their problem frame templates, and instantiate the corresponding sub-problem diagrams. Our semantics allows us to express the relationship between problems and sub-problems; we will briefly illustrate this process for the chemical reactor.

The chemical reactor problem can be decomposed into three distinct sub-problems:

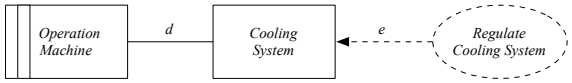
- a commanded behaviour frame allowing the operator to control the catalyst;
- a required behaviour frame regulating the water flow for cooling; and
- an information display frame for issuing a warning (and halting the system) when there is an oil leak in the gearbox.

To be able to apply the problem frames as templates, the domains and phenomena of the problem must match those of the frame. The instantiation of a problem frame for a specific problem results in the sub-problem diagram that was matched by the problem frame. In the Chemical Reactor problem, the instantiation of the commanded behaviour frame is the problem diagram of Figure 3; that of the required behaviour frame is given in Figure 4; and that of the information display frame in Figure 5.



$a : OP!\{open\_catalyst, close\_catalyst\}$
$b : OM!\{open\_catalyst_{act}, close\_catalyst_{act}\}$
$\quad CA!\{is\_open_{sen}, is\_closed_{sen}\}$
$c : CA!\{open, closed\}$

Figure 3. The *Control Catalyst* sub-problem, an instantiated commanded behaviour frame



$d : OM!\{increase\_water_{act}, decrease\_water_{act}\}$
$\quad CS!\{is\_rising_{sen}, is\_falling_{sen}\}$
$e : CA!water\_level$

Figure 4. The *Regulate Cooling System* sub-problem, an instantiated required behaviour frame

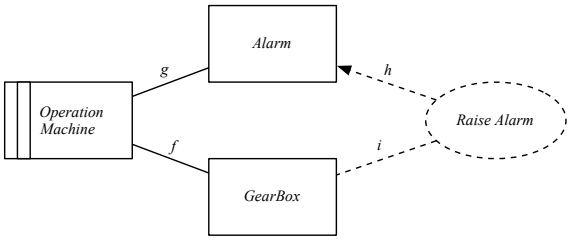
3 The Reference Model

In the section we briefly recall the Reference Model, which underpins our semantic characterisation of Problem Frames.

The Reference Model [15, 5, 6] is based on the widely accepted separation between the system (which we call the machine) and its environment (which we call the problem domains); on a number of key artifacts which pertain to the two; and on a vocabulary, which is used to describe the environment, the system and the interface between them. From a requirements engineering viewpoint, the key artefacts are the following[1]: the domain knowledge, K, is what we know about the environment; the requirement, R, is what the customer requires of a system working within the environment; and the specification, S, is a description of the system, which can be used for its implementation.

The Reference Model formalises the relation

[1] Other artefacts are defined in the Reference Model, which pertain to design and implementation. They are beyond the scope of this article, and are omitted here.



$f : GB!request\_service \quad g : OM!ring\_bell$
$h : AL!bell\_ringing \quad i : GB!oil\_level$

Figure 5. The *Raise Alarm* sub-problem, an instantiated information display frame

[15] between world, requirements and machine as:

$$K, S \vdash R$$

That is: a proof can be given that an implementation of S, introduced into a problem domain satisfying K, will guarantee satisfaction of the requirement R.

In the Reference Model, designations are used to ground terms in the real world [15]. The designations provide names to describe the environment, the system and their artefacts. The vocabulary is used to name phenomena, which are grouped into: those denoted by e, which belong to, and are controlled by, the environment; and those, denoted by s, which belong to, and are controlled by, the system (or machine). e is further partitioned into phenomena which are visible to (or observable by) the machine (e_v), and those that are hidden from it (e_h); s is further partitioned into phenomena which are visible to (or observable by) the environment(s_v), and those that are hidden from it (s_h). $e_v \cup s_v$ is the vocabulary of the interface between the environment and the machine.

For technical reasons, the Reference Model assumes that the specification, S, is written in the interface vocabulary: i.e., S can use only terms denoting phenomena in $e_v \cup s_v$. On the other hand, K and R can use terms denoting phenomena in $e \cup s_v$.

4 Towards a semantics for Problem Frames

In this section we describe the semantics in terms that we have already introduced. The semantics begins with a non-graphical representation language for the elements of the PF framework. This is based on the (uninterpreted) Domain and Requirements Description Language (*DRDL*) which is used to express requirements and domain properties—our equivalents of basic propositions. We then build problem diagrams and problem frames upon this, in the Problem Frames Description Language (*PFDL*). Then we define problem frame instantiation, and finish with the interpretation of decomposition and recomposition.

4.1 *PFDL*: a Description Language for Problem Frames

We introduce a textual notation for problem frames, the *Problem Frames Description Language* (or *PFDL*). The purpose of this notation is to provide a (textual) syntactic domain on which to define our semantic function.

As discussed earlier, we assume that some language (or a collection of languages) has been chosen for the description of domains, phenomena, requirements and specifications—these are the 'basic propositions' of our semantics. This we refer to as the *Domain and Requirement Description Language* (or *DRDL*, for short). We assume only that *DRDL*:

- allows the representations of phenomena and their relationships, can distinguish controlling influence over them from observation of their values, and can mark phenomena and domains appropriately;
- comes equipped with some notion of reasoning, which we will refer to as $\vdash_{DRDL}$ (or simply as $\vdash$ when the context makes this clear).

We do not require the reasoning of $\vdash_{DRDL}$ to be formal in any proof theoretic sense. It may be, for instance, that the reasoning takes a pragmatic

software engineering form, such as testing. We return to this point in Section 5.

4.1.1 Describing problem diagrams

The basic components of problem diagrams are domains (including machine domains), requirements and their connectivity through arcs and annotations. To be able to express this in *PFDL*, we assume the following sets, representing, respectively, all domains, all machine domains, all requirements and all phenomena:

$$\texttt{PFDomain}, \texttt{PFMachine}, \texttt{PFRequirement}, \texttt{PFPhenomena}$$

For the grounding of a problem, both domains and requirements must have names and descriptions in *DRDL*[2]:

$$\texttt{PFDomainName} \triangleq \texttt{PFDomain} \to \mathit{DRDL}$$
$$\texttt{PFDomainDescription} \triangleq \texttt{PFDomain} \to \mathit{DRDL}$$

$$\texttt{PFRequirementName} \triangleq \texttt{PFRequirement} \to \mathit{DRDL}$$
$$\texttt{PFRequirementDescription} \triangleq \texttt{PFRequirement} \to \mathit{DRDL}$$

Phenomena are also named and described in the *DRDL*:

$$\texttt{PFPhenomenaDesignation} \triangleq \texttt{PFPhenomena} \to \mathit{DRDL}$$

Phenomena are partitioned in sets of phenomena that can be either controlled or observed by each domain and the machine:

$$\texttt{PFControlled} \triangleq \texttt{PFDomain} \cup \texttt{PFMachine} \to \texttt{PFPhenomena}$$
$$\texttt{PFObserved} \triangleq \texttt{PFDomain} \cup \texttt{PFMachine} \to \texttt{PFPhenomena}$$

[2]We will often identify the domain with its name. This requires, of course, that the mapping from domains to names is injective. Similarly with requirements and phenomena.

Similarly, some phenomena can be either constrained or referenced by the requirement:

$$\texttt{PFConstrained} \triangleq \texttt{PFRequirement} \to \texttt{PFPhenomena}$$
$$\texttt{PFReferenced} \triangleq \texttt{PFRequirement} \to \texttt{PFPhenomena}$$

We will bundle all annotating functions into a *problem diagram annotation*:

$$\texttt{PFAnnotation} \triangleq (\texttt{PFDomainName}, \texttt{PFDomainDescription}, \texttt{PFRequirementName}, \texttt{PFRequirementDescription}, \texttt{PFPhenomenaDesignation}, \texttt{PFControlled}, \texttt{PFObserved}, \texttt{PFConstrained}, \texttt{PFReferenced})$$

(For brevity, if A is the problem diagram annotation, we will refer to the nine annotating functions as $A_1, \ldots, A_9$, where A_1 is the domain names annotation, A_2 is the domain descriptions annotation, and so on.)

A problem diagram is a set of domains, a single machine, a single (set of) requirement(s), and an annotation, i.e.:

$$\texttt{PFProblemDiagram} \triangleq (\mathbb{P}\,\texttt{PFDomain}, \texttt{PFMachine}, \texttt{PFRequirement}, \texttt{PFAnnotation})$$

The correspondence between the graphical elements of a problem diagram and their PFDL counterpart is illustrated in Figure 6.

As an example, let $(\{D_1, \ldots, D_6\}, M, R, A)$, where $A = (A_1, \ldots, A_9)$, be the problem diagram for the chemical reactor problem. Then its *PFDL* description, in tabular form, is shown in Figure 7.

4.1.2 Describing problem frames

As already mentioned, basic problem frames are templates for classes of problems. [9] provides a template language enriched over that for problem diagrams by the introduction of domain and phenomena markings. These markings must be matched by problem diagram elements for the problem frame template to apply.

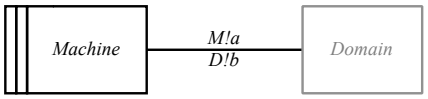
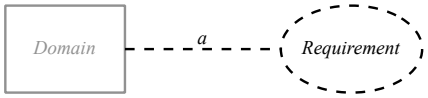
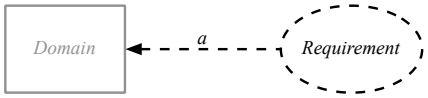
Graphical element	Element	Name	Controlled/ Constrained	Observed/ Referenced
	*M*	*Machine*	*a*	*b*
	*D*	*Domain*	*b*	*a*
	*R*	*Requirement*		*a*
	*R*	*Requirement*	*a*	

Figure 6. The relationship between the graphical elements of a problem diagram and their counterpart in the *PFDL*

Each domain can be marked as biddable (B), lexical (X) or causal (C):

$$\texttt{PFDomainMarking} \triangleq \texttt{PFDomain} \rightarrow \{\texttt{B}, \texttt{X}, \texttt{C}\}$$

Each (set of) phenomena can be marked as causal (C), symbolic (Y) or event (E):

$$\texttt{PFPhenomenaMarking} \triangleq \texttt{PFPhenomena} \rightarrow \{\texttt{C}, \texttt{Y}, \texttt{E}\}$$

The diagram of a problem frame is called a *frame diagram* and is a problem diagram augmented with markings for domains and phenomena:

$$\texttt{PFFrameDiagram} \triangleq (\texttt{PFProblemDiagram}, \texttt{PFDomainMarking}, \texttt{PFPhenomenaMarking})$$

A Problem Frame provides a *frame concern*, i.e., a template for correctness arguments that is common to all problems of that class. We assume that frame concerns are taken from another given set:

$$\texttt{PFFrameConcern}$$

and so we may define:

$$\texttt{PFProblemFrame} \triangleq (\texttt{PFFrameDiagram}, \texttt{PFFrameConcern})$$

A *PFDL* representation of the commanded behaviour frame of Figure 2 is given in Figure 8 (we have omitted the frame concern). The reader will note that the phenomena controlled by the operator are event phenomena (marking E), while all other phenomena are causal (marking C); such markings will constrain the applicability of the problem frame to a problem diagram.

4.2 Problem diagrams as challenges

As in program refinement [13], in which *pre* and *post* conditions define a 'challenge' to develop code to implement them, so a problem diagram, through its environment and requirements, defines a *challenge* to develop a machine specification to solve them. Whereas refinement is mature, and has formal notation capable of representing a challenge[3] we, as yet, do not. We will therefore be

[3]The reader may be familiar with the refinement notation $w : [pre, post]$ which stands for (all) code that, by changing only variables in w, will take as input any state satisfying *pre* and produce a state satisfying *post*.

	Name	Controlled/Constrained	Observed/Referenced
M	*Operation Machine*	$open\_catalyst_{act}$, $close\_catalyst_{act}$ $increase\_water_{act}$, $decrease\_water_{act}$ *ring_bell*	is_open_{sen}, is_closed_{sen} is_rising_{sen}, $is\_falling_{sen}$ *request_service* *open_catalyst*, *close_catalyst*
D_1	*Catalyst*	is_open_{sen}, is_closed_{sen} *open*, *closed*	$open\_catalyst_{act}$, $close\_catalyst_{act}$
D_2	*Cooling System*	is_rising_{sen}, $is\_falling_{sen}$ *water_level*	$increase\_water_{act}$, $decrease\_water_{act}$
D_3	*Reactor*	$\varnothing$	$\varnothing$
D_4	*Operator*	*open_catalyst*, *close_catalyst*	$\varnothing$
D_5	*GearBox*	*request_service* *oil_level*	$\varnothing$
D_6	*Alarm*	*bell_ringing*	*ring_bell*
R	*Control*	*open*, *closed* *water_level* *bell_ringing*	*open_catalyst*, *close_catalyst* *oil_level*

Figure 7. The Chemical Reactor problem diagram in tabular form

	Name	Marking for Domain	Marking for Controlled/ Constrained Phenomena	Marking for Observed/ Referenced Phenomena
M	*Control Machine*	$--$	$C1$	$C2, E4$
D_1	*Controlled Domain*	C	$C2, C3$	$C1$
D_2	*Operator*	B	$E4$	$C1$
R	*Commanded Behaviour*	$--$	$C3$	$E4$

Figure 8. *PFDL* **description of the commanded behaviour frame**

forced to improvise, notationally, somewhat.

We use the following definition as the basis for our semantics. Given a world description K, a requirement description R, and sets of shared phenomena c (controlled by the machine domain) and o (observed by the machine domain) over some *DRDL*, we define a 'challenge to find a satisfying specification' as

$$\begin{aligned} c, o : [K, R] \quad = \quad \{S \mid\ & S \text{ controls } c \\ & \wedge\ S \text{ observes } o \\ & \wedge\ K, S \vdash_{DRDL} R\} \end{aligned}$$

As such, $c, o : [K, R]$ stands for the set of all those specifications that control phenomena in the set c, and observe phenomena in the set o, and are capable, in the context of K, of discharging R, as per [15][5][6]. As indicated in the notation, the details of $\vdash_{DRDL}$ will be provided by the *DRDL*. We also note that $c, o : [K, R]$ can be empty (as when R is *false*, for instance), in which case there is no satisfying specification, i.e., the system cannot be built. Referring to the original Reference Model, $c = s_v$, i.e., the machine controlled phenomena shared with the environment, and $o = e_v$, the machine visible shared phenomena.

To derive the challenge which stands as semantics for a particular problem diagram, one must only look to its *PFDL* description. Given a problem diagram $(\{D_1, \ldots, D_n\}, M, R, A)$, with $A = (A_1, \ldots, A_9)$, in the semantics, for conve-

nience we define its challenge as

$$c, o : [DD_1 \wedge ... \wedge DD_n, DR]$$

where:

- $DD_i = A_2(D_i)$ — the problem diagram's domain descriptions;
- $DR = A_4(R)$ — the problem diagram's requirement description;
- $c = A_6(M)$ — those phenomena controlled by the problem diagram's machine domain;
- and $o = A_7(M)$ — those phenomena visible to the problem diagram's machine domain.

The reader will note that discharging the correctness argument associated with the diagram is part of the challenge: specifically, demonstrating that $K, S \vdash_{DRDL} R$, that S controls c, and that S observes o.

4.3 Instantiation as injection

A problem frame is a template for a class of problems. By 'applying' a problem frame to a problem diagram, we can derive a (sub-)problem diagram whose semantics is a challenge. This allows for a process of problem decomposition in which the sub-problem diagrams are projections of the original problem diagram through appropriately chosen problem frames.

Semantically, we can see the application of a problem frame to a problem diagram as matching domains and connections (i.e., the topology), and the types of the domains and phenomena. We say that a frame diagram $PF = (D, M, R, A, DM, PM)$ matches a problem diagram $PD = (D', M', R', A')$ when there is an injective correspondence $\iota : D \rightarrow D'$ which respects the topology, domain and phenomena types.

A problem frame instantiation is then the application of ι to (D, M, R, A, DM, PM).

4.4 Addressing composition and recomposition

Extending the analogy with program refinement, working towards an $S \in c, o : [K, R]$ will be an exploration of the design choices at each stage: because we work within all of software—including architectures, etc—and not just within code, the design space is possibly much larger than that in refinement.

Problem frames, as templates, allow us to navigate the design space by dividing up the problem. For example, in the Chemical Reactor problem, we have three sub-problem diagrams corresponding to the required behaviour, commanded behaviour, and information display frames. The decomposition of this problem into three sub-problems is illustrated in Figure 9, in which:

$$\begin{aligned}
c_1 &= \{open_catalyst_{act}, close_catalyst_{act}\} \\
o_1 &= \{is_open_{sen}, is_closed_{sen}, \\
&\quad open_catalyst, close_catalyst\} \\
c_2 &= \{increase_water_{act}, decrease_water_{act}\} \\
o_2 &= \{is_raising_{sen}, is_falling_{sen}\} \\
c_3 &= \{ring_bell\} \\
o_3 &= \{request_service\} \\
c &= c_1 \cup c_2 \cup c_3 \\
o &= o_1 \cup o_2 \cup o_3
\end{aligned}$$

As in refinement, given a possible decomposition through the application of problem frames, there will be a corresponding composition concern analogous to refinement's verification condition: we must be able to convince ourselves of the validity of the recomposition. Our point is that the backwards argument (upwards in Figure 9) identifies the sub-problems that must be solved, while the forwards argument (downwards in Figure 9) requires that composition concerns are identified and the associated proof obligations are discharged.

In the Chemical Reactor problem, for instance, there is a requirement that when the oil level in the gearbox is low, the machine should sound an alarm and halt (the information display sub-problem). On the other hand, the machine is always required to regulate the cooling system (the required behaviour sub-problem). The composition of these two apparently conflicting sub-problems must continue to guarantee the safe functioning of the reactor. This concern must therefore be addressed at the composition level, perhaps by reconsidering one or both of these requirements.

$$\frac{\begin{array}{r} c_1, o_1 : [\{Catalyst, Operator\}, Control\ \ Catalyst] \\ c_2, o_2 : [\{Cooling\ \ System\}, Regulate\ \ Cooling\ \ System] \\ c_3, o_3 : [\{Alarm, GearBox\}, Raise\ \ Alarm] \end{array}}{c, o : [\{Catalyst, Cooling\ \ System, Operator, Alarm, GearBox\}, Control]} \uparrow \text{Decomp.}, \downarrow \text{Recomp.}$$

Figure 9. The decomposition/recomposition relationship between the Chemical reactor problem diagram and its sub-problem diagrams

The indication that requirements are in conflict can be taken from the backwards development of the challenges. The binding of symbols that occurs throughout the tree through backwards development may lead to a situation in which the recomposition concern cannot be discharged. This is natural in such proof theoretic frameworks. By analogy, sometimes, when we start from a unprovable 'theorem', it is because the 'theorem' contains a contradiction and the 'theorem's' statement must be rethought. As with system development through problem frames, conflicting requirements may prevent the system being built. When we are made aware of such requirements—because the challenge cannot be met—we must reconsider them, and formulate different, non-conflicting requirements.

5 Discussion and Future Work

In this paper we have provided a semantics for elements of the PF framework. We have introduced the Problem Frames Description Language (PFDL), over a Domain and Requirements Description Language (DRDL), for the description of problem diagrams and problem frames. We have defined the meaning of a problem frame as a template for problem diagrams, and a semantics for problem diagrams within the Reference Model, given in terms of challenges of the form $c, o : [K, R]$, a notion that we have also introduced.

We stress that our semantics is based on a weak notion of correctness. In this we differ from the Reference Model semantics, which is based on universal quantification over all 'phenomenon sequences'. The actual wording used by the authors is:

> A requirement with an environment constraint should always be verified by a demonstration or proof that specified properties, conjoined with domain knowledge, guarantee the satisfaction of the requirement.

In our semantics we relax that notion of proof to its weakest (non-technical) meaning:

> facts, evidence, argument, etc. establishing or helping to establish a fact.

We might, with this definition, also admit proofs of the type 'an extensive search has found no case in which the system did not satisfy the requirements' (a testing 'proof'); or the 'customers were convinced by the arguments of the developers'; or 'a formal refinement of the requirements to code was performed, with all verification conditions being discharged'; or 'Dijkstra programmed it'; even 'we'll fix it in the next release' might do. Clearly, the nature of $\vdash_{DRDL}$ and that of the requirements are closely related: where the requirements include human safety (as may be the case in a critical system), the strength of $\vdash_{DRDL}$ must provide for an appropriate level of proof.

The development of the challenge expression $c, o : [K, R]$ needs many things. Most importantly, the availability of a 'weakest environment predicate transformer' (analogous to the weakest

precondition semantics of refinement) would place the semantic basis of the manipulation of such expressions on a much firmer footing, and even lead us towards a requirements engineering notion of 'refinement' that forms a lattice over such objects; given a problem, the refinement relation $\sqsubseteq$ would justify the relationship, summarised in Figure 10:

$$problem \sqsubseteq partial\_soln_0 \sqsubseteq \ldots \sqsubseteq specification$$

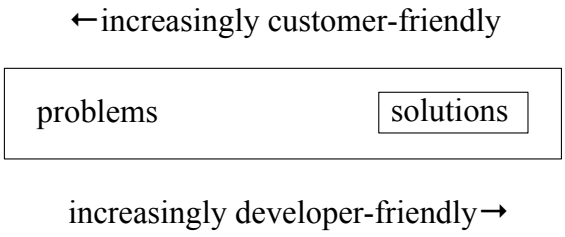


Figure 10. The move between problem and solution domains (after [13, Figure 1.6, page 9])

Finally, we observe that this paper provides a basis for a formal semantics for elements of the PF framework. The range of uses to which it can be put is still largely unexplored, and will provide a focus for further research.

6 Acknowledgements

We acknowledge the kind support of our colleagues, especially Bashar Nuseibeh and Robin Laney, in the Department of Computing, the Open University. Also, we thank the anonymous reviewers, whose comments have helped us to improve the paper greatly.

References

[1] D. Bjorner, S. Koussoube, R. Noussi, and G. Satchok. Michael jackson's problem frames: Towards methodological principles of selecting and applying formal software development techniques and tools. In *1st IEEE International Conference on Formal Engineering Methods*. IEEE Computer Society Press, 1997.

[2] O. Dieste and A. Silva. Requirements: Closing the gap between domain and computing knowledge. In *Proceedings of SCI2000, Vol. II (Information Systems Development)*, 2000.

[3] A. Evans, J.-M. Bruel, R. France, K. Lano, and B. Rumpe. Making UML precise. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[5] C. Gunter, E. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 3(17):37–43, 2000.

[6] J. G. Hall and L. Rapanotti. A Reference Model for Requirements *Engineering*. In *Proceedings of the 11th International Joint Conference of Requirements Engineering*, 2003.

[7] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[8] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles, and Prejudices*. Addison-Wesley, 1995.

[9] M. Jackson. *Problem Frames*. Addison-Wesley, 2001.

[10] S. Konrad and B. H. Cheng. Requirements patterns for embedded systems. In *IEEE Joint International Conference on Requirements Engineering*, 2002.

[11] P. G. Larsen, N. Plat, and H. Toetenel. A formal semantics of data flow diagrams. *Formal Aspects of Computing*, 6(6):586–606, 1994.

[12] N. Leveson. Software safety: why, what and how. *ACM Computing Survey*, 18(2):125–163, 1986.

[13] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[14] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[15] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.