Technical Report No: 2004/13

Dynamic Assembly of Problem Frames

Leonor Barroca
José L. Fiadeiro
Michael Jackson
Robin Laney

22nd April 2004

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Dynamic Assembly of Problem Frames

L.Barroca[1], José L. Fiadeiro[2], M.Jackson[1], R. Laney[1]
[1]Department of Computing
The Open University
Milton Keynes MK7 6AA, UK
{l.barroca, m.jackson,r.c.laney}@open.ac.uk
[2]Department of Computer Science
University of Leicester
Leicester LE1 7RH, UK
jose@fiadeiro.org

ABSTRACT

This paper addresses the support of modular, compositional and incremental analysis and design of software systems by the assembly of problem frames. We use coordination-based techniques to put in place an architectural layer in which, for each subproblem, we provide a description of the machine and the way it is interconnected with the components of the problem domain to fulfill given customer requirements. Composition in this architectural layer is dynamic in the sense that it is not constrained to follow pre-established decomposition structures; instead, it allows fully incremental development. The architectural layer provides a basis on which we will be able to support reconfiguration of the system at execution time by the addition or substitution of new machines or new problem domain components resulting from new requirements.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.10 [**Software Engineering**]: Design; D.2.11 [**Software Engineering**]: Software Architectures; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; K.6.3 [**Management of Computing and Information Systems**]: Software Management

General Terms

Management, Design, Reliability, Languages, Theory, Verification.

Keywords

Composition, Coordination, Evolution, Requirements, Software Architectures.

1. INTRODUCTION

Problem decomposition is a key activity in software engineering. Addressing individual well-contained parts of a system that deal with specific features of the problem domain, and integrating these parts incrementally, is a major step in dealing with complexity in software development. This is not an easy task and it requires a good understanding of what the domain problems are, and the best ways of breaking them into sub-problems.

It is now widely recognised that a good decomposition structure also needs to support a natural and easy evolution of software systems that can accommodate the addition of new, or changes to old, requirements. Hence, it is important that the structure that emerges from requirements decomposition can be reflected in the solution domain so that the actual software system in place can evolve in a way that is *compositional* with respect to the changes that take place in the application domain. In this context, compositionality means the ability to evolve the whole system by operating on individual parts as black-boxes, i.e. without interfering with their internal structure, what sometimes is called Plug-and-Play.

Recent work [e.g. 3], suggests that, to ensure the required degree of compositionality, one should work at the architectural level of systems, i.e. the layer that provides the transition between the requirements and the solution operating in its domain [20] (see Figure 1). More precisely, work on architectures and connectors [1] has shown that, by promoting interactions to first-class citizens as architectural connectors, one can operate directly over the mechanisms that coordinate the way system components interact. This allows for the addition of new requirements to be performed incrementally by operating independently over the connectors and the components in a given configuration, thus ensuring compositionality. Moreover, such reconfiguration steps can be performed at run-time, thus catering for important business or organisational properties like time-to-market and guaranteed minimal levels of service.

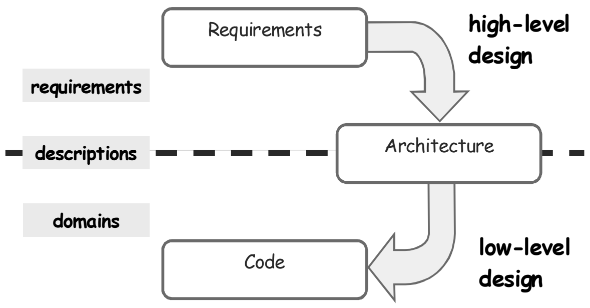

Figure 1: The architectural layer

Different strategies have been rehearsed for mapping this architectural decomposition into the code level, some of which are discussed in [3] that are based on method-call-interception. Current research on reflective middleware [8] is also enhancing reconfigurability at platform level. The main challenge has remained to relate architectural structure with requirements decomposition. This is the area that this paper addresses.

Problem frames, as introduced in [13], have proved to provide good support for the decomposition effort by making explicit how given components of a software system interact with the application domain in order to satisfy given customer requirements. In previous work [4] we looked at how problem frames can be enriched with representation schemes based on the coordination-based approach to architectures that was developed in [10] and adopted for software evolution in [3]. In this paper, we show how that approach can also deal with compositionality in requirements evolution, including incremental integration of new requirements.

Section 2 provides an overview of problem frames, makes clear the scope of the application of coordination primitives and architectural modelling techniques, and presents the example that we adopt for illustrating our approach. Section 3 presents coordination-laws and coordination-interfaces, and applies them to the definition of problem frames. Section 4 addresses compositionality, i.e. the ability to map the composition of problem frames to a corresponding composition of architectural elements. Section 5 deals with dynamic composition and the way it supports incremental analysis and design. Finally, section 6 concludes and presents related and future work.

2. BACKGROUND

2.1 Problem Frames

Problem frames is an approach to problem analysis and description [13]. It recognises that problems can usually be categorised as a set of commonly occurring patterns for which the same type of models can be used. The approach emphasises the relationships of systems to the real world domains where they live. Problem frames encapsulate both real world and system objects, and describe the interactions between them.

A simple problem frame is typically represented by a problem diagram showing one machine, one problem domain, and the shared phenomena between them. The Machine Domain represents the piece of software that the customer requires and the platform on which it executes in order to bring about some desired effects. The Problem Domain is that part of the world in which those effects are perceived by the customer. The Requirements are the properties that the customer wants to observe, in terms of phenomena shared with the problem domain, as a result of the effects brought about by the software as it executes and interacts with the domain.

In order to illustrate our approach, we look at a sluice gate that is controlled manually by an operator [13]:

> *A rising and falling gate is used in an 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 controlled by clockwise and anticlockwise pulses. There are sensors at the top and bottom of the gate travel indicating when the gate is fully opened and fully shut. The operator commands are to raise, lower, or stop the gate.*

Problem analysis is essentially concerned with the description of the relationships among the phenomena that are shared between these different domains.

In the example, we have two Problem Domains; one – *Gate&Motor* – is concerned with the gate, its motor, and the way it can be observed and operated; the other – *Operator* – is concerned with the commands from the operator. The Machine is the *Gate_Controller*, i.e. the computer system that is needed to control the gate. It shares with *Gate&Motor* the events that it controls – the commands *onClockw* (to raise the gate), *onAnti* (to lower the gate) and *off* (to stop the gate) for operating the gate as made available through the motor (see Figure 2). The *Gate&Motor* shares with the customer the observations of the state of the gate as made available through the sensors – being fully *up* or *down* as indicated in Figure 2. Although not represented in the diagram we also rely on the operator to observe the state of the gate through the sensors. The *Gate_Controller* observes the commands from the operator – *raise, lower, stop* as depicted in Figure 2, and reacts accordingly; these commands can also be observed by the customer. No relationship is represented between the *Operator* and *Gate&Motor*: the goal of the software system is, precisely, to mediate the interactions between these two Problem Domains.

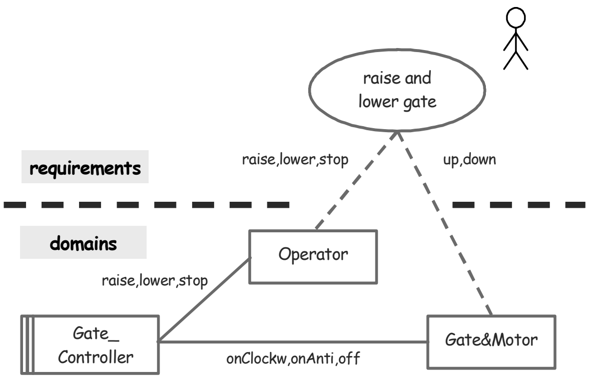

Figure 2: Problem diagram for an occasional sluice gate

Design of a software system cannot be undertaken on the basis that a problem decomposition structure has been and will remain fixed. It is important that software development puts in place design structures that are flexible enough to accommodate changes in the problem decomposition when and as they occur. In other words, incremental development must be perceived as a run-time, on-line activity.

2.2 The Architectural Layer

The machine represented in a problem frame is a piece of software that we want to superpose, possibly at run-time, over the components that are part of the problem domain. The machine interacts with the problem domain components through the declared shared phenomena, so that new behaviour can emerge that satisfies user requirements. Our approach to making design incremental, and supporting evolution in a compositional way, is to introduce a design layer in which we provide a model of the machine and the way it is interconnected with the components in the problem domain. This is what, in the introduction, we have called the architectural layer. The modelling primitives that we propose for this architectural layer are based on the separation between Coordination and Computation concerns [11].

As motivated in [4], Coordination is intrinsic to the way problem frames are used for decomposing and analysing problems. The Machine is a computational device that is superposed on the domain to coordinate the joint behaviour of its components. The computations of the Machine are of interest only to the extent that, when interacting with the components of the domain, they enable required properties to emerge. Hence, the central concern for evolution must be the explicit representation of the mechanisms that are responsible for the coordination of the interaction between the Machine and the Domain.

Figure 3 depicts the wider software development context that results from the introduction of the architectural layer.

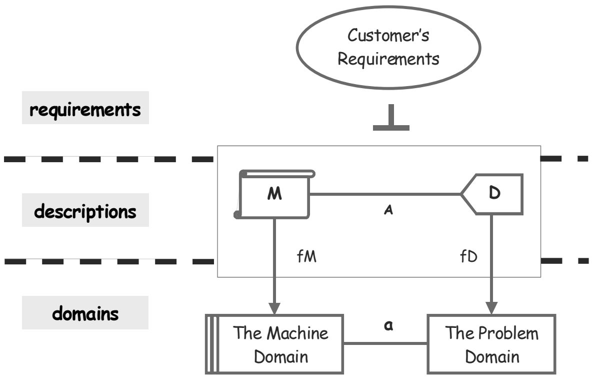

Figure 3: Architectural context

By *M* we mean the description of the behaviour of the execution of the software system (Machine) on a particular platform. Our approach is to view *M* as the body of an architectural connector [1] that has a role *D* for every entity of the Problem Domain with which the software must interact. This is what in the next section we will call a *coordination law* (M) and its *coordination interfaces* (D).

By *D* we mean the model that abstracts the properties of the Problem Domain that concern the way the Machine can interact with it (D points to M as it represents the domain as seen by the machine). By *A* we denote the way the models *M* and *D* are related. This is where we separate coordination from computation: in *A/D* we wish to place only the aspects that concern the way the machine interacts with the domain. These include the phenomena through which this interaction takes place, and the assumptions on which it is based, i.e. the properties that are assumed of the domain in the way the computations taking place in the Machine are programmed. This explicit externalisation of the coordination aspects is what we call a *coordination interface* of the coordination law.

The coordination interfaces D make explicit properties of the Problem Domain entities with which the Machine interacts. These properties should be taken together with the properties of the Machine when proving satisfaction of customer requirements. For a fixed set of requirements, the stronger D is, the weaker M must be, i.e. the easier/cheaper it is to construct the Machine; but, on the other hand, the weaker D is (and the stronger M is), the easier/cheaper it will be to accommodate changes in the entities of the Problem Domain that are interacting with the Machine. Deciding on how strong D should be, i.e. how much to tailor the software to the current entities of the Problem Domain or leave room for variation is, therefore, critical for development/evolution costs.

For instance, in terms of the sluice-gate, and for the sake of argument (none of the authors being able to claim expert knowledge on sluice-gates…), one may think that the customer has always worked with motors that cannot be changed direction without being stopped first; the customer may consider that (perhaps because he also produces the motors) this is how the Problem Domain will remain, in which case it makes sense to include this property in D (i.e. one that is ensured by the Problem Domain) and develop M under these assumptions; if, however, the customer does not want to remain tied to this particular brand of motors and he knows that the market is moving to motors that behave differently, he may wish not to include this property in D and, therefore, pay for a more sophisticated piece of software that will work together with other kinds of motors.

Because M and D are only (abstract) descriptions of component behaviour, the vertical relationships between the description and the domain worlds also play a central role in our approach. They make explicit how the actual components (software or otherwise), in their respective domains, *fit* these descriptions. This is important so that we know what are the properties of the entities in the Machine and Problem Domain on which the good behaviour of the current system relies; i.e. it satisfies all customer requirements. If requirements stop being satisfied, we should be able to trace the cause to the breakdown of one or more of these vertical relationships, e.g. the Machine was installed in a new platform that no longer validates the description given through M, or a physical component started malfunctioning.

By *fM* we mean the *fit* that must exist between the description of the Machine and the behaviour of its deployments. Indeed, there is normally a gap between the high-level specification of component models and their implementation in any particular technology. This gap is currently being filled by the code of the fine-grain parts that have to glue the domain components with the architectural framework provided by the underlying technology (CORBA, J2EE, .NET, *inter alia*). Even when using design patterns to structure the code, the final result mixes the code that implements the domain logic and the infra-structural glue code. The lack of a clear separation results in software components that are very difficult to maintain and evolve. The fit *fM* is what supports the required separation between the code that implements the domain logic (as modelled through the body of the connector) from infrastructural code that is platform dependent.

By *fD* we mean the fit between the model (coordination interface) and the Problem Domain. Depending on the nature of the domain that the software system is controlling, this fit may or not be of a formal nature. For instance, the Machine may be controlling another software system (e.g., monitoring some of its properties), in which case *fD* can be cast in a semantic domain common to the two language/platform couples.

Satisfaction of customer's requirements is established on the basis of the triple *M/A/D* and the correctness of the fits *fM/fD*. Requirements are typically expressed in a logical formalism (L, $\vdash$) as a sentence *R*. A mapping *P(M/A/D,fM,fD)* into *L* is thus required that is correct with respect to the semantics of (L, $\vdash$), characterising customer satisfaction as:

$$P(M/A/D,fM,fD) \vdash R$$

The set of properties *P(M/A/D,fM,fD)* will contain those that describe the behaviour of the Machine, as captured through M, and those that describe the expected behaviour of the entities in the Problem Domain with which the Machine interacts, as captured through the interfaces D. However, there are additional sets of properties that need to be considered.

One the one hand, *P* must involve the properties of the domain that can provide an adequate bridge between the phenomena at *a* (i.e. those shared between machine and domain), as abstracted through *fM/fD*, and the phenomena shared with the customer. For instance, in the case of the sluice gate, one would probably rely on properties relating the sensors and the motor, such as:

```
onClockw ⊃ (up unless off)
onAnti ⊃ (down unless off)
```

We are using here the syntax of a temporal logic [18] that we will not formalise in this paper:

- The first property reads: "After the *onClockw* command is issued on the motor, the event *up* will be eventually observed unless the *off* command is issued in the meanwhile". In other words, if the motor is started clockwise and left undisturbed for long enough, the event *up* will eventually be observed through the sensors.

- The second property is similar to the first; it states that if the motor is started anti-clockwise and left undisturbed for long enough, the event *down* will be eventually observed through the sensors.

Any formalisation of the properties of a physical domain is an approximation to the reality, and different approximations are appropriate for different problems, of course. One may very well need to evolve the mapping due to the realisation that the approximations being made are not good enough or valid anymore.

The same applies to the platform properties that are subsumed by *fM* such as those that relate to transaction management, concurrency, distribution, etc. For instance, in the case of the sluice gate one might rely on the fact that *Gate_Controller* cannot issue commands concurrently:

```
onClockw ⊃ ¬off ∧ ¬onAnti
onAnti ⊃ ¬off ∧ ¬onClockw
off ⊃ ¬onAnti ∧ ¬onClockw
```

Summarising, we are making explicit which are the properties that we should check when we change the platform in which the software is running in order to ensure that *R* will still be obtained at the other end of the implication; in other words we are trying to prevent situations in which software is reused without caring about the platform for which it has been developed.

Finally, we need to account for the properties of the admissible configurations, i.e. those that are related with the way machines are installed and interconnected to the other entities in the Problem Domain. For instance, in order to avoid unwanted interactions, precedence relations may have to be imposed on different machines that are controlling some part of the domain. Another example concerns initialisation conditions: because we want to support incremental and dynamic composition of machines, they cannot be modelled for specific decomposition structures and states; it is for the configuration process to make sure that machines are installed with a correct view of the current state of the system.

In this paper, we will only address the definition of coordination laws and interfaces for given problem domains and their composition to support incremental developm,ent. All the other aspects will be left for a subsequent paper.

3. COORDINATION

Our approach to the representation of the coordination aspects involved in problem frames is based on the *coordination technologies* presented in [3]: a set of principles and modelling techniques that were first proposed in [2] based on the notion of *coordination contract*, or *contract*, for short. The purpose of coordination contracts is to make available in modelling languages, like the UML [5], the expressive power of *connectors* in the terminology of Software Architecture [20]. The overall goal is to provide a means for the interaction mechanisms that relate system components to be externalised from the code that implements these components. It makes the relationships explicit in system models so that they can be evolved independently of the computations that the components perform.

Contracts prescribe certain coordination effects (the *glue* of architectural connectors in the sense of [1]) that can be superposed on a collection of partners (system components) when the occurrence of one of a set of specified *triggers* is detected in the system. Contracts establish interactions at the instance level when superposed on a running system. In the terminology that we used in the previous section, they sit in the Machine domain as part of the code. At the level that concerns us in the paper, the triple *M/A/D*, the architectural primitive that abstracts the properties of the glue from the code that implements it in contracts, is called a *coordination law* as described below.

3.1 Domain Assumptions

In the description of a coordination law, the natures of the components over which the law can be instantiated are identified as *coordination interfaces* (the *roles* of the connector type in the sense of [1]). These are defined so as to state *requirements* placed by laws on the entities that can be subjected to its rules and not as a declaration of features or properties that entities offer to be coordinated.

In the occasional sluice gate example (Figure 2) each domain model is abstracted as a coordination interface. The idea is to declare what the Machine is expecting from each problem domain in the way it has been designed to control it. Two primitives are made available in coordination interfaces for that purpose: *services* and *events*.

- Services identify operations that the domain must provide for the Machine to invoke.
- Events identify state transitions that the Machine must be able to detect in the problem domain. These act as triggers for the contract that is being put in place to react and activate a *coordination rule* as discussed below.

For instance, as depicted in Figure 2, *Gate_Controller* shares with *Gate&Motor* the commands *onClockw*, *onAnti* and *off* for operating the gate as made available through the motor. This means that we need a coordination interface that captures essential properties of the way *Gate_Controller* expects the motor to behave:

```
coordination interface motor
services     onClockw, onAnti, off
properties
    (onClockw∨onAnti) ⊃ (¬(onClockw∨onAnti) before
off)
end interface
```

This example illustrates that coordination interfaces are not just declarations of features (signatures) but include properties as well. These properties capture semantic aspects of the roles that components are expected to play in the Problem Domain; they are the ones that establish the correctness of every fit *fD.* That is to say, such properties express requirements on the behaviour of the components that are under the coordination of the machine.

This explicit representation of the roles expected of the components of the Problem Domain is extremely important: on the one hand, it allows us to detect mismatches that result from changes occurring in the Problem Domain and cause the fit *fD* to become incorrect; on the other hand, it establishes the criteria for components in the Problem Domain to be replaced without upsetting the interconnections with the Machine.

Returning to our example, we chose a property expressed in the syntax of a temporal logic to illustrate the kinds of assumptions that can be made and the implications that they have. The property reads: "After an *onClockw* or an *onAnti* command is accepted, no more such commands will be accepted before an *off* command is accepted". In physical terms, this means that the domain couple *Gate&Motor* is being assumed to prevent these events from occurring. One may think, for instance, of a motor that provides three buttons, one for each operation, and that the *onClockw* and *onAnti* buttons become locked once pressed and will only be unlocked after the *off* button is pressed.

Summarising, the importance of recording this property is because, according to the good judgment of the customer, software (the Machine) will have been designed precisely for this kind of motors; hence, if for some reason another generation of motors replaces these, we are saying that the software may no longer lead to the satisfaction of customer requirements and something must be done. For instance, most modern Hi-Fi equipment stopped having an input for turntables; a special adaptor is now required to connect the turntable to the pre-amplifier.

The sluice-gate example introduces another Problem Domain – the *Operator*. Its relationship with *Gate_Controller* can be characterised by the commands *raise*, *lower* and *stop* that it issues and to which *Gate_Controller* is required to react by activating the services of the motor.

```
coordination interface operator
events       raise, lower, stop
end interface
```

Notice that *Gate_Controller* cannot control the *Operator*; hence, it does not require any services that it can invoke. Indeed, *Gate_Controller* is purely reactive to *Operator*; this is why it requires that it can observe the three given commands that *Operator* can issue.

3.2 Modelling the Machine

The effects that the software system is required to bring about are described through the *coordination rules* of the law that describes the behaviour of the Machine. In the case at hand, this means activating the services of the *Gate&Motor* on request from the *Operator:*

```
coordination law Uncontrolled_Remote
partners     mt: motor; op: operator
rules
    1  when  op.raise
         do  mt.onClockw;
    2  when  op.lower
         do  mt.onAnti;
    3  when  op.stop
         do  mt.off
end law
```

Each coordination rule is of the form:

```
when  trigger
with  condition
  do  set of operations
```

Under the *when* clause, the trigger to which the contracts that instantiate the law will react is specified as a Boolean condition defined over the events declared in the interface and conditions over the internal state of the law. Under the *with* clause we specify a guard, a Boolean condition defined over the internal state of the law that, if false, leads to a rejection of the trigger. The reaction to be performed is identified under the *do* clause as a set of operations, each of which is either a service declared in the interface or an update on the internal state of the law. The whole interaction is handled as a single transaction, i.e. its execution is atomic.

The use of this law in a specific configuration of the Problem Domain requires two fits, one for each interface (see Figure 4): *fG&M* maps the services required by the *motor* interface to actions of *Gate&Motor*, and *fOp* maps the events of the *operator* interface to the actions with the same names as identified in the Problem Frame. These maps are identities because, for simplicity, we have chosen the same names on the coordination interfaces and the Problem Domain. However, they may be much more involved, especially when the interface is abstracting phenomena that must be mapped to more complex domain actions or observations. What is important is that the map establishes the relationships required and that the fit is proved correct, i.e. that the properties required in the interface can be shown to hold in the Problem Domain.

In this example, the coordination effects that are put in place simply transfer the commands issued by the operator to the motor without any additional control.

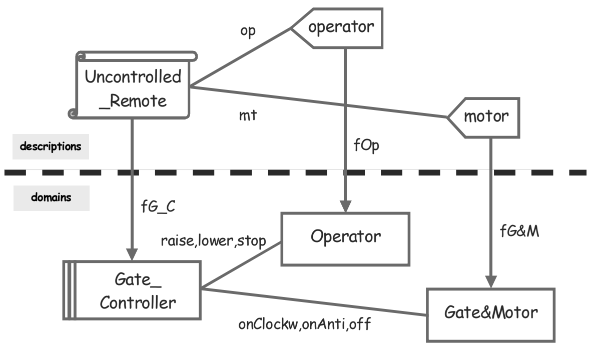

Figure 4: Uncontrolled interaction

4. INTEGRATION OF NEW CONCERNS

Because the operator commands the sluice machine directly with no external control, there is nothing to prevent undesirable sequences of commands: commands may be issued when they make no sense or when they are not viable. There is a case here for separation of concerns to detect and prevent the unwanted behaviour from the operator separately from the way the motor is operated.

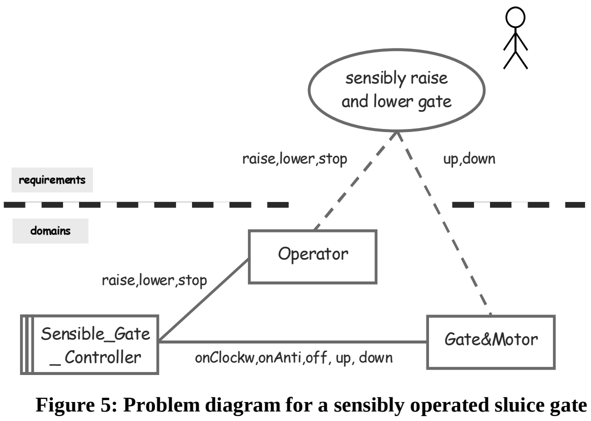
sensibly raise and lower gate
raise,lower,stop
up,down
requirements
domains
Operator
raise,lower,stop
Sensible_Gate _Controller
onClockw,onAnti,off, up, down
Gate&Motor

Figure 5: Problem diagram for a sensibly operated sluice gate

In order to allow only sensible commands to be actually transmitted to the gate, we want to be able to superpose another coordination mechanism over the interaction of the operator with *Gate&Motor*. Sensible commands correspond to only issuing a *lower* command when the motor is stopped and the gate has not reached the bottom, issuing a *raise* command only when the motor is stopped and the gate has not reached the top, and issuing a *stop* command only when the motor is in operation. A new set of requirements leads to the new problem frame of Figure 5.

Notice that the new Machine – *Sensible_Gate_Controller* – is now required to share with *Gate&Motor* the observations of the state of the gate as made available through the sensors – being fully *up* or *down* as indicated in Figure 5. Hence, the new coordination law requires a more sophisticated interface, which can be given by

```
coordination interface motor&sensor
services     onClockw, onAnti, off
events       up, down
properties
    (onClockw ∨ onAnti) ⊃
            (¬(onClockw ∨ onAnti) before off)
    onClockw ⊃ (up unless off)
    onAnti ⊃ (down unless off)
end interface
```

Notice the new events to be mapped to the sensors. Notice also that the properties relating the motor to the events need to be included now: the motor started clockwise will lead to a *up* event if not interrupted (and the same for anticlockwise and the *down* event).

The new controlled mode of operation is captured by the coordination rules of the law specified below. Notice that the law allows the motor to be stopped before the gate is completely closed or open, which may be useful for emergency situations, and started in either direction.

```
coordination law Sensibly_Operated
partners     mc: motor&sensor; op: operator
attributes   stopped, open, shut: bool
rules
    1  when  mc.up
         do  shut:=false
    2  when  op.raise
       with  stopped ∧ shut
         do  mc.onClockw ∥ stopped:=false ∥ open:=true
    3  when  mc.down
         do  open:=false
    4  when  op.lower
       with  stopped ∧ open
         do  mc.onAnti ∥ stopped:=false ∥ shut:=true
    5  when  op.stop
       with  ¬stopped
         do  mc.off ∥ stopped:=true
end law
```

Notice the use of attributes at the level of the law. They are used as internal representations that the *Sensible_Gate_Controller* makes of the state of the *Gate&Motor* in order to control its behaviour as required. These attributes are just a prosthesis that relates to the nature of the formalism that is being used for describing the behaviour of the Machine; they are not features that are required of the code that lies in the Machine and, therefore, can be ignored by the fit *fM*. The Boolean attribute *open* is true as long as the gate is not totally down; *shut* is true as long as the gate is not totally up.

This new law restricts the effectiveness of the *raise* and *lower* commands of the operator to the states in which the motor is stopped and the gate is not (totally) open and (totally) shut, respectively; in the other states, the commands are not passed over to the motor. This is the result of having specified with-clauses (guards) in the rules that are triggered by commands issued by the operator. The *stop* command is only transmitted to the motor when the motor is operating.

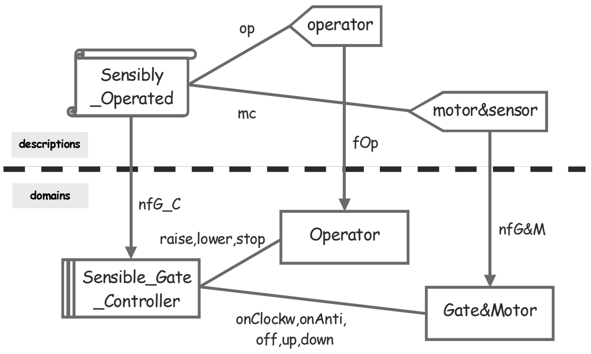

Figure 6: Sensibly operated interaction

The fact that, in a configuration, we make these interconnections explicit allows us to evolve from an uncontrolled to a sensibly operated mode by simply replacing one machine by the other as long as the sensors are indeed available in the Problem Domain as required by the new coordination interface *motor&sensor*. This results in a new fit *nfG&M*. In Figure 6, this change can be observed by having replaced *Gate_Controller* by a new machine *Sensible_Gate_Controller* that must fit the new law *Sensibly_Operated*. Naturally, the Problem Domain components themselves – *Operator* and *Gate&Motor*– have not changed.

This example illustrates the kind of compositionality that we motivated in the introduction: changes in a specific requirement involve only the corresponding machine, i.e. the coordination aspects of the global system. However, the example can also be used to illustrate a more sophisticated kind of compositionality: the one that supports incremental development.

Indeed, one may wish to regard this new law as resulting not from a revision of previous requirements but, instead, from the addition of a new requirement. In other words, one may wish to consider that, having detected that the operator may act in non-sensible ways under the uncontrolled law, the customer issues new requirements to prevent such unwanted behaviour. Rather than reformulate the whole problem and the corresponding Machine, one may wish to proceed incrementally and add a new problem frame over the same Problem Domain but only for the new set of requirements. The machine in this new frame acts as a filter that detects the events from the sensors and establishes the state of the gate.

This new problem frame leads us to a new coordination law:

```
coordination law Sensibly
partners     sr: sensor; op: operator
attributes   stopped, open, shut: bool
rules
    1  when  sr.up
         do  shut:=false
    2  when  op.raise
       with  stopped ∧ shut
         do  stopped:=false ∥ open:=true
    3  when  sr.down
         do  open:=false
    4  when  op.lower
       with  stopped ∧ open
         do  stopped:=false ∥ shut:=true
    5 when op.stop
      with ¬stopped
        do stopped:=true
end law
```

Notice that the coordination interface that this law requires, besides the *operator*, as before, involves only the sensor:

```
coordination interface sensor
events       up, down
end interface
```

This is because, as indicated in the new problem frame (see Figure 7), the *Operator_Controller* shares with *Gate&Motor* only the observations provided by the sensors. No direct action must be performed on *Gate&Motor*; the coordination is performed by preventing the *Gate&Motor* from reacting to the requests of the *Operator* in the undesirable states.

We now have two laws directly sharing an interface – *operator* – and, coordinating it with two different components of the Problem Domain – the sensors and the motor, both of which are part of *Gate&Motor*. Hence, we require two fits – *fS&M* and the previous *fG&M* – to the same Problem Domain component – *Gate&Motor*, resulting in two machines controlling the same domain simultaneously.

The semantics of this simultaneous application of the two laws, as shown in Figure 7, is the one that results from the union of the sets of coordination rules: the reaction to two triggers that are both true in a given state is guarded by the conjunction of the with-clauses and performs the union of their synchronisation sets (do-clauses). For instance,

```
when op.raise
with stopped ∧ shut
  do stopped:=false ‖ open:=true
```

from *Sensibly* (2) and

```
when op.raise
  do mc.onClockw
```

from *Uncontrolled_Remote* (1) compose to give

```
when op.raise
with stopped ∧ shut
  do mc.onClockw ‖ stopped:=false ‖ open:=true
```

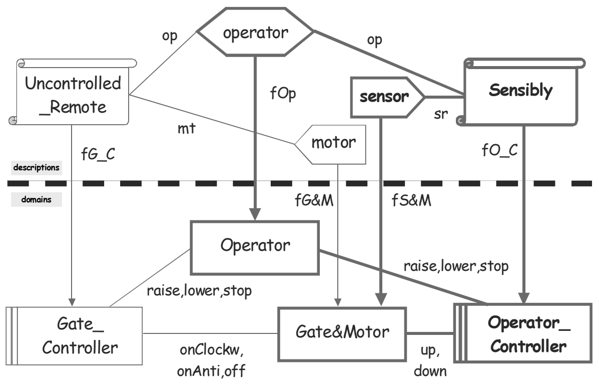

Figure 7: Decomposed sensibly operated interaction

This semantics captures the parallel composition of the corresponding machines synchronised at the events that they share. It is easy to see that this composition of *Uncontrolled_Remote* and *Sensibly* results in a set of coordination rules equivalent to that of *Sensibly_Operated* (2).

It is not possible to present herein the semantics of coordination laws that justifies this operation of parallel composition. The reader interested in a mathematical justification can consult [10] for a complete account of composition of architectural connectors in a categorical setting.

5. COMPOSITION OF CONCERNS

This incremental approach to problem analysis, as illustrated in the previous section, is one of the advantages we see in the combination of the coordination-based modelling primitives with problem frames. In this section, we focus more specifically on the composition aspects. For that purpose, let us analyse another version of our case study.

Other concerns of the sluice gate problem that need to be dealt with are related to ensuring that the *Gate&Motor* is not damaged due to not respecting certain restrictions of operation. The two following restrictions are examples of good functioning of the *Gate&Motor*:

1. There is a minimum period – *switchtime* – that has to be allowed for change of direction of the gate.
2. There is a maximum period – *motorlimit* – that should not be exceeded between detecting an *up* or a *down* event from the sensors and switching the motor off.

We start from Figure 7 and the simultaneous application of the two coordination laws, Uncontrolled_Remote and Sensibly, that guarantee only sensible commands from the operator. Our approach supports the definition of two new coordination laws: one for each new requirement. The advantage of doing so is that they can be independently superposed, in run-time, to the gate, regardless of any modes of operation that may be or become in place. For instance, it should not matter if the operator is acting sensibly or not in the sense discussed in the previous section.

Each of these new properties requires a new interface between the *Gate&Motor* and the new machine. For instance, for the first requirement, the new machine has to detect changes of the direction of the gate. This can be achieved by considering that *onClockw* and *onAnti* are now events of the interface of the law that enforces the new requirement; this means that the new Machine must be able to detect when the *onClockw* and *onAnti* of the motor are invoked, regardless of who invokes them.

```
coordination interface switch
events       onAnti, onClockw, off
end interface

coordination law Switch_Concern
partners     mc: switch; tm:timer
parameters   switchtime:nat
attributes   direction: {↑,↓}; changeable,idle:bool
rules
    1 when mc.onClockw ∧ direction=↓
      with changeable
        do direction:=↑;
    2 when mc.onAnti ∧ direction=↑
      with changeable
        do direction:=↓;
    3 when mc.onAnti ∨ mc.onClockw
        do changeable:=false ‖ idle:=false;
    4 when mc.tick(switchtime) ∧ idle
        do changeable:=true
    5 when mc.off ∧ ¬idle
        do tm.reset ‖ idle:=true
end law
```

In order to enforce the new requirement on switching direction, we need to distinguish the situation when the motor moves, stops and then moves again in the same direction from the situation when the motor moves, stops and then moves in the opposite direction. In the first case there is no requirement that the motor has to be stationary for the amount of time *switchtime*, while in the second, such requirement is imposed. For this purpose, we introduce attributes that are local to the law and record information about the state of the gate. The attribute *direction* records the direction in which the motor is working; *idle* records the fact that the motor is stationary. As already discussed, these attributes are just a prosthesis; they are not features that are required of the code that lies in the Machine

The coordination rules enforce the new requirement as follows:

(a) *switchtime* is calculated from each *mc.off* occurring in the non-idle state (rule 5).

(b) *changeable* becomes true only when the idle state has held for *switchtime* (4), subject to correct initialisation (see below).

(c) occurrences of *mc.onClockw* and *mc.onAnti* that change direction are accepted only when *changeable* is true (1, 2).

(d) when accepted, calls for *onClockw* and *onAnti* on the machine cause *changeable* and *idle* to become false (3).

Other properties that emerge are:

(e) from any state in which *idle* is true and *changeable* is false, *changeable* will be set to true after one stretch of *switchtime*.

(f) calls to *onAnti* with *direction*=↓ or *onClockw* with *direction*=↑ can be accepted.

(g) from any state in which *idle* is true, calls to *off* are ignored until an occurrence of *mc.Anti* or *mc.onClockw*.

(h) from any state in which *direction*=↓ any call for *onAnti* can be accepted (similarly for *direction*=↑ and *onClockw*).

As indicated in (b), some of the required behaviour can only be ensured subject to proper initialisation. As discussed in section 2, we consider initialisation to be a concern not of the description of the machine but of the configuration process, i.e. it must be addressed at configuration time, not at design time. This is because, to ensure reusability and true modular decomposition of problems, we cannot design solutions from specific configurations of decomposition structures. Moreover, in order to support incremental and run-time composition, initialisation of a machine will depend on the configuration under which the system is executing. For instance, the correct instalation of *Switch_Concern* will depend on the state of *Gate&Motor* and the component that instantiates *timer* (see below). One possibility is to wait for the engine to stop, initialise *direction* according to the movement of the gate, *idle* to true, *changeable* to false, and reset the timer. Another possibility is to wait for the motor to be idle for at least *switchtime* before installing *Switch_Concern* and initialise *changeable* to true. These aspects are left to the configuration process that we mentioned in section 2 and that will be discussed in a subsequent paper.

Notice that we have had to rely on a third component of the Problem Domain – a timer. The timer must provide a service for being *reset* and events that report elapsed time – *tick(n:nat)*.

```
coordination interface timer
services     reset
events       tick(n:nat)
properties
    tick(n) ⊃ (¬tick(m) before (tick(n+1) ∨ reset))
    reset ⊃ tick(0)
end interface
```

In this case, the properties are specifying that the ticking should be incremental and sequential until the timer is reset, starting with 0.

This example also illustrates how coordination interfaces can be used for identifying not only the features of the Problem Domain through which required forms of coordination are going to be superposed, but also components that need to be provided in addition to the Machine. Such components, like the timer, are not necessarily part of the original Problem Domain – the *Gate&Motor* does not necessarily come with an integrated timer or clock – nor of the solution domain – the timer is not to be realised by software. They need to be provided at configuration time, and they may be evolved independently of the Machine and changes operated in the application domain, provided that the fit to the coordination interface is maintained.

This law does not require the *Operator* as a partner. Indeed, it just superposes safety mechanisms on the *Gate&Motor*, thus illustrating in another way how our approach supports separation of concerns. However, it is important to analyse what happens when this law is bound to *Gate&Motor* while connected to *Operator* through a *Sensibly_Operated* contract, or an assembly of *Uncontrolled_Operator* and *Sensibly*, which is behaviourally equivalent. When the operator invokes *raise* with the gate shut and the motor stopped, rule (2)

```
when op.raise
with stopped ∧ shut
  do mc.onClockw ‖ stopped:=false ‖ open:=true
```

is activated. Because the guard is true, the execution of the reaction starts. When, as part of the execution of the synchronisation set, *onClockw* is invoked on the motor, rule (1)

```
when mc.onClockw ∧ direction=↓
with changeable
  do direction:=↑
```

is activated and, if the guard is false, i.e. if the delay for switching direction has not been observed, the trigger is refused and the whole transaction fails. That is, the invocation *raise* by the operator fails and the motor is not started. Again, this is because of the semantics of parallel composition as applied to the simultaneous execution of several laws.

This is an example of the kind of validation that we would like to see assisted by a tool supporting our approach. One of the costs of incremental and modular development through the assembly or superposition of new laws is, precisely, in controlling the interference between the new and existing laws. This is known in Telecommunications as the Feature Interaction Problem [23].

Consider now the second requirement, concerned with the observation of the limit for stationary operation. We need to be able to determine when the motor begins operating in *stationary* mode, and switch it off when the limit is reached. Again, a timer is required:

```
coordination interface station
services     off
```

```
events       up, down, off
end interface

coordination law station_concern
partners     mc: station; tm:timer
parameters   motorlimit:nat
attributes   stationary:bool
rules
    1  when  (mc.up ∨ mc.down)
         do  tm.reset ‖ stationary:=true
    2  when  mc.off
         do  stationary:=false
    3  when  tm.tick(motorlimit) ∧ stationary
         do  mc.off
end law
```

It is interesting to notice that *off* is required both as an event and a service, meaning that the new machine must be able to invoke this action and also detect when it is invoked (regardless of who invokes it).

This new law can be added to the system independently of the previous one, i.e. one can choose whether or when both requirements should come into force, which corresponds to putting in parallel one or more controllers. One of the advantages of our approach is that different requirements, leading to different problem frames, get represented through different coordination laws so that a typical and-composition of requirements corresponds to a typical parallel composition of the machines that enforce them, provided that the implementation platform in which these machines run supports the operational semantics of parallel composition that we outlined, the formalisation of which can be found in [10].

Indeed, the coordination primitives that we have been describing fit well into the problem frames approach to decomposition, which is substantially different from what is normally found in Software Engineering. As put in [15], traditional decomposition assumes a pre-established structure or architecture into which the parts identified in the decomposition are fitted as they are successively identified. This means that each part must conform to the modes of interaction that this structure subsumes, say remote procedural calls of given services, or reading and writing sequential data streams, which do not necessarily reflect requirements that derive from the problem domain and, therefore, introduce unnecessary bias.

In the problem frames approach, decomposition is carried out making few or no explicit assumptions about the mechanisms by which each machine may interact with others. Because the coordination approach is based on the externalisation of interactions and the dynamic superposition of the connectors that coordinate them, each machine can be described and developed by assuming no more than that is a solution to a sub-problem. Composition concerns can be addressed at configuration time, i.e. when the different machines need to be brought together as a global solution to the top-level problem. In fact, the coordination approach is dynamic in the sense that it can support the evolution of the initial configuration to reflect changes in the requirements.

Deferring composition concerns until the different sub-problems have been well identified and understood is a key feature of the problem frames approach. In our opinion, this is well justified given that we consider that composition is not a static, compile-time problem of linkage, but a dynamic process that must be subjected to its own rules. The coordination approach goes somewhat further by advocating an explicit separation between the two concerns and providing specific primitives to model configuration and evolution. and will be reported in future papers. For instance, for the purpose of managing configurations, other composition operators become relevant such as precedence between laws to resolve undesirable interactions. As already mentioned, this is part of the work that we are now pursuing.

6. CONCLUDING REMARKS

We have discussed primitives for representing explicitly, in the problem frames approach, the coordination aspects that concern the interaction between the Machine and the Problem Domain. The resulting models constitute an architectural layer that relates requirements and the software solutions that satisfy them when executed in the given Problem Domain. This architectural layer facilitates the incremental integration of new concerns resulting from changes in the requirements.

6.1 Related Work

As far as we know, ours is one of the first attempts at bringing together problem decomposition approaches to requirements specification and principles of separation of concerns that have been typically used for software design. This is an effort that, in our opinion, will allow us to contribute towards taming the complexity of evolving software applications according to the changes that occur in the problems that they are meant to solve.

Existing approaches to decomposing problems rather than solutions, like KAOS [16] and the NFR framework [7], do not address the separation of concerns that our coordination approach promotes, as they do not concentrate on domain properties in the same pervading manner as problem frames. Composition of software artefacts on the basis of separation of concerns has been addressed by a range of aspect-oriented techniques [9]. However, with the notable exception of [12] and [21], aspect-based approaches, whilst good at addressing design and implementation issues, are weak with regards to requirements, and in particular their decomposition. The approaches of [12] and [21] are mainly concerned with reconciling conflicts between a range of non-functional requirements and do not fully address decomposition of functional requirements.

There is also little work relating requirements and architecture, exceptions include [5,6]. However, those works do not fully address problem decomposition.

6.2 Further work

Composition is often a dynamic concern in the sense that conflicting requirements many times result from the need to distinguish contextual modes of operations, for instance exceptional circumstances that require normal behaviour to be overridden. In other words, conflicts are normally avoided when one takes into account the circumstances in which each requirement applies and with which priority. Hence the advantage of having mechanisms for addressing the process of (re)configuration explicitly.

Thanks to the ability of coordination contracts to be dynamically assembled and superposed over existing systems without intruding in the components (machines) already in place, we can, indeed, support a process of run-time evolution that is driven by customer policies or directives that put requirements in a dynamic context. For instance, consider again the operator-controlled sluice machine; the need for superposing some of the controls that we have mentioned may depend on the nature of the operator in the particular case; different kinds of operators may require different kinds of controls; hence, the contract that, in a given configuration of the system, is coordinating the way the operator is interacting with the sluice-gate, may change as one operator is replaced by another. We may even envision a system that self-adapts to this change of operator by reconfiguring itself through the replacement of one contract by another. The same applies, for instance, for certain dependability requirements: these are usually costly in terms of performance and often conflict with basic functional requirements that should prevail in normal circumstances. Hence, it makes sense that the corresponding contracts are only superposed to the system when necessary, taking precedence over those that control a stationary functioning.

We are currently investigating methods and techniques for supporting the evolution of configurations, for which we are borrowing previous work on dynamic reconfiguration of distributed systems and evolving architectures [3,17,22]. Our approach is to separate completely two concerns: the *what* – which are the different requirements and the machines that enforce them – from the *when* – in which circumstances does each requirement apply and, in case of conflict, which prevails. As a result, separate requirements can be handled independently at the static stage, leaving the control of their interference to a dynamic process of reconfiguration controlled by a specific set of primitives.

Acknowledgements

The authors would like to thank their colleagues for valuable discussions, in particular Bashar Nuseibeh in The Open Universityy, Luis Andrade at ATX Software SA, Antónia Lopes at the University of Lisbon, and Michel Wermelinger at the New University of Lisbon.

7. REFERENCES

1. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM,* 6(3), 1997, 213-249.
2. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in R.France and B.Rumpe (eds), *UML'99 – Beyond the Standard*, LNCS 1723, Springer-Verlag 1999, 566-583.
3. L.F.Andrade and J.L.Fiadeiro, "Architecture Based Evolution of Software Systems", in M.Bernardo & P.Inverardi (eds), *Formal Methods for Software Architectures*, LNCS 2804, Springer Verlag 2003, 148-181.
4. L.Barroca, J.L.Fiadeiro, M.Jackson, R.Laney, B.Nuseibeh, "Evolving Problem Frames: a Case for Coordination", in *Coordination Languages and Models*, R.deNicola and G.Meredith (eds), Springer-Verlag 2004
5. D.Berry, R.Kazman and R.Wieringa (eds), *Proceedings of Second International Workshop from Software Requirements to Architectures (STRAW'03)*, Portland, USA, 2003.
6. J.Castro and J.Kramer (eds), *Proceedings of First International Workshop from Software Requirements to Architectures (STRAW'01)*, Toronto, Canada, 2001.
7. L.Chung, B.A.Nixon, E.Yu and J.Mylopoulos. *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers, 2000.
8. G.Coulson, G.Blair, M.Clarke and N.Parlavantzas, "The design of a configurable and reconfigurable middleware platform", in *Distributed Computing* 15(2), 2002, 109-126.
9. T.Elrad, R.Filman and A.Bader (Guest editors). Special Issue on Aspect Oriented Programming. *Communications of the ACM* 44(10) 2001.
10. J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", in *Generic Programming*, R.Backhouse and J.Gibbons (eds), LNCS 2793, Springer-Verlag 2003, 190-234.
11. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35(2), 1992, 97-107.
12. J.Grundy, "Aspect-Oriented Requirements Engineering for Component-based software systems", in *Fourth IEEE International Symposium on Requirements Engineering (RE'99).* IEEE Computer Society Press 1999.
13. M.Jackson, *Problem Frames: Analysing and Structuring Software Development Problems,* Addison Wesley 2000.
14. M.Jackson, "Some Basic Tenets of Description", *Software System Modelling* 1, 2002, 5–9.
15. M.Jackson, "Why Software Writing is Difficult and Will Remain So", in J.Fiadeiro, J.Madey and A.Tarlecki (eds), *Information Processing Letters Special Issue in Honour of Wlad Turski* 88(1-2), 2003.
16. A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", in *Proceedings of the 5th International Symposium on Requirements Engineering (RE'01),* IEEE Computer Society Press 2001, 249-261.
17. J.Magee and J.Kramer, "Dynamic Structure in Software Architectures", in *4th Symp. on Foundations of Software Engineering,* ACM Press 1996, 3-14.
18. Z.Manna and A.Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1991.
19. B.A.Nuseibeh, "Weaving Together Requirements and Architecture", IEEE Computer 34 (3):115-117, March 2001.
20. D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, 40-52.
21. A.Rashid, A.Moreira, and J.Araujo, "Modularisation and Composition of Aspectual Requirements", *Aspect Oriented Software Development* 2003.
22. M.Wermelinger and J.L.Fiadeiro, "Algebraic Software Architecture Reconfiguration", in *Software Engineering – ESEC/FSE'99*, LNCS 1687, Springer-Verlag 1999, 393-409.
23. P.Zave, "Feature Interactions and Formal Specifications in Telecommunications", *IEEE Computer* XXVI(8), 1993, 20-30.
24. P.Zave and M.Jackson, "Conjunction as Composition", *ACM TOSEM* 2(4), 1993, 371-411.