Technical Report No: 2003/08

Architecture-driven Problem Decomposition

Lucia Rapanotti
Jon G Hall
Michael Jackson
Bashar Nuseibeh

October 2003

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

http://computing.open.ac.uk

The Open University

Architecture-driven Problem Decomposition

Lucia Rapanotti, Jon G. Hall, Michael Jackson, Bashar Nuseibeh
Computing Department, The Open University
Walton Hall, Milton Keynes, MK7 6AA, UK
{L.Rapanotti, J.G.Hall, M.Jackson, B.A.Nuseibeh}@open.ac.uk

Abstract

Jackson's Problem Frames provide a means of analysing and decomposing problems. They emphasise the world outside of the computer helping the developer to focus on the problem domain instead of drifting into inventing solutions. The intention is to delay consideration of the solution space until a good understanding of the problem is gained.

In contrast, early consideration of a solution architecture is common practice in software development. Software is usually developed by including existing components and/or reusing existing frameworks and architectures. This has the advantage of shortening development time though reuse, and increasing the robustness of a system through the application of tried and tested solutions.

In this paper, we show how these two views can be reconciled and demonstrate how a choice of architecture can facilitate problem analysis and decomposition within the Problem Frames framework. In particular, we introduce Architectural Frames *– combinations of architectural styles and Problem Frames – and illustrate their use in problem decomposition by applying them to a well-known problem from the literature.*

1 Introduction

Problem Frames [5, 6] classify software development problems. They structure the analysis of the problem and the world in which it is located — the problem space — describing what is there and what effects one would like a system located there to achieve. With its emphasis on problems rather than solutions, the Problem Frame approach uses an understanding of a problem class to allow the problem owner with their specific domain knowledge to drive the Requirements Engineering process.

Three characteristics of modern software development are in competition with this approach.

Firstly, even modestly complex problems can force problem owner and solution engineer into negotiation over trade-offs and consideration of details of the solution [1]. Secondly, in practice, the development of new systems is very rarely green-field: new software is usually developed from existing components [2] or within existing frameworks [3] and architectures [9, 1]. Finally, expert workers in well-known application domains express their expertise through development, even for bespoke software.

We observe that each of these competing views embody knowledge of the solution space: the first through the software engineer's domain knowledge; the second through choice of domain specific architectures, architectural styles, development patterns, *etc*, the third through the reuse of past development experience. All solution space knowledge can and should be used to inform the problem analysis for new software developments within that domain. Time to market of quality systems is shortened through the reuse of such solution space structures and experience.

The main contribution of this paper is to use architectural styles [1, 9] to guide problem decomposition, a fundamental part of the analysis process. To do this, we define a new tool within the Problem Frame framework, that of an *architectural frame* (or *AFrame*). AFrames characterise the combination of a problem class and an architecture class. An AFrame can be regarded as a Problem Frame for which a standard sub-problem decomposition exists. AFrames are a practical tool for sub-problem decomposition that allow the Problem Frame practitioner to separate and address in a systematic fashion the concerns arising from the intertwining of problems and solutions. We describe in detail the application of the techniques to a case study from the literature [8, 1, 9]. The work here builds on that reported in [4], where we focused on component-based development.

The paper is organised as follows. Section 2 provides a short introduction to the Problem Frames framework and to the case study and discusses the motivation behind the work. Section 3 contains the main contribution of the paper. It introduces the notion of architectural frame, their

decomposition techniques and their application to the case study. It also addresses the issue of correctness within the resulting analysis process. Section 4 discusses related work. Finally, Section 5 concludes the paper.

2 Background and Motivation

In this section we review some of the basic elements of the Problem Frames (PF) framework. A comprehensive presentation is beyond the scope of this paper and can be found in [6]. We also discuss issues related to problem decomposition, and the motivation for our approach. Finally, we introduce the case study.

2.1 The Problem Frames framework

One of the aims of the PF framework is to identify basic classes of problems that recur throughout software development. Each such class should be captured by a *Problem Frame* that provides a characterisation for the problem class. [6] identifies five *basic* Problem Frames. In this section, we will illustrate one of them, the Transformation Frame, in order to introduce some basic concepts.

The Transformation Frame is used when we need to transform some input into some output of a particular format, and by applying certain rules. The problem then is that of building a machine that can produce the required outputs from the inputs [6]. A Problem Frame has a topology, which is represented by a *frame diagram*. That of the Transformation Frame is shown in Figure 1.

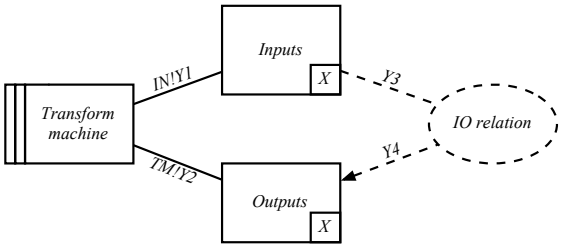


Figure 1. The Transformation Frame Diagram

The diagram deserves some explanation:

- The box *Transform machine*: the *machine domain*, i.e., the software system to be built, and its underlying hardware.
- The other boxes, *Inputs* and *Outputs*: *given domains* representing parts of the world that are relevant to the problem.
- The dotted oval, *IO relation*: the *requirement*, i.e., what has to be true of the world for the machine to be a solution to the problem.
- The connections between the various elements (indicating *shared phenomena*), including events, operations and state information. In Figure 1, for example, the connection between the *Transform machine* and *Inputs* is annotated by a set $Y1$ of phenomena controlled by *Inputs*: control is indicated by an abbreviation of the domain name followed by *!*, in this case *IN!*.

The frame diagram also includes an indication of the characteristics of the domains and phenomena involved. For the Transformation Frame:

- The *Inputs* and *Outputs* domains are *lexical* domains (the X annotation in the figure); their phenomena are symbolic (indicated by Y on the arcs).
- The *Transform machine* has access to symbolic phenomena of the *Inputs* domain (in $Y1$) and controls the output phenomena which are shared with the *Outputs* domain (in $Y2$).
- The requirement, *IO relation*, states what is required of the output phenomena given the inputs. Phenomena in $Y3$ and $Y4$ may or may not be the same as those of, respectively, $Y1$ and $Y2$. The difference could reflect, for instance, a different granularity: say, the machine operates at character level, while the requirement is expressed in terms of paragraphs.
- In the links between the requirement and the domains, a dotted line indicates that the phenomena are *referenced* by (i.e., an object of) the requirement, while a dotted arrow indicates that the phenomena are *required* (i.e., a subject for the requirement). In this case, the inputs are reference, while the outputs are required.
- The lexical phenomena at the requirement interface (i.e., those of sets $Y3$ and $Y4$) are distinct from those at the machine domain interface (i.e., those of sets $Y1$ and $Y2$). The former are called *requirement phenomena*; the latter, *specification phenomena*. The intuition behind this distinction is that the requirement is expressed in terms of elements of the problem, while the specification (that is what describes a machine domain) is expressed in terms of elements of the solution.

When a problem of a particular class is identified, it can be analysed though the instantiation of the corresponding frame diagram. The instantiation is a process of matching problem's and frame's domains and their types, as well as problem's and frame's phenomena types. The result of the instantiation is a *problem diagram*, which has the same topology of the frame diagram, but with domain and phenomena grounded in the particular problem.

For a problem to be fully analysed, this instantiation is only a first step of the process. Other necessary steps are the

provision of domain descriptions, and addressing a number of concerns.

Descriptions need be provided for all given domains (addressing relevant characteristics and behaviours), the machine domain (the specification) and the requirement. An important distinction in the PF framework is that of separating two types of descriptions: *indicative* and *optative*. Indicative descriptions are those which describe how things are; optative descriptions describe how things should be. In this sense, in a problem diagram, given domain descriptions are indicative, while requirement and machine descriptions are optative. In other words, things that have to do with the problem domain are given, while things that have to do with the solution domain can be chosen.

Each Problem Frame has some associated concerns, which have to be addressed in the analysis of the problem. For the Transformation Frame these are:

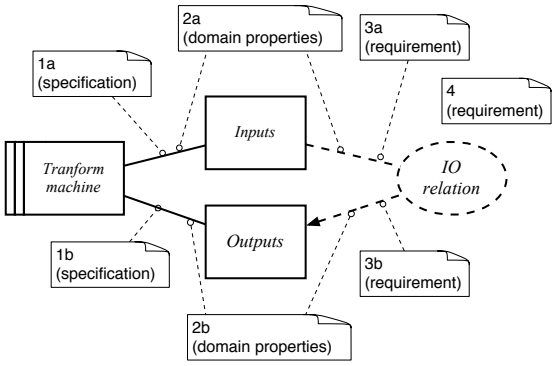
- the relation between specification and requirement phenomena;
- how the transform machine must traverse the lexical domains, that is how it will access and produce data in the inputs and outputs domains; and
- the *frame concern*, an overall correctness argument, which is common to all the problems of the class. Each Problem Frame comes with a particular concern, whose structure depends on the nature of the class problem. For the Transformation Frame, one has to be satisfied that as the machine, while traversing inputs and outputs, generates the outputs from the inputs correctly. The corresponding argument is outlined in Figure 2.

2.2 Problem Decomposition

Most real problems are too complex to fit basic Problem Frames. They will require, rather, the restructuring of the problem as a collection of (interacting) sub-problems, each of which is smaller and simpler than the original. Problem decomposition is the PF process through which this is achieved.

There is some ambiguity about how problem decomposition might be done in practice. The PF approach [6] relies heavily on the expertise of practitioners to understand which decompositions of a problem into its constituent subproblems are appropriate and will lead to a 'good solution'. This expertise is not assumed to have any particular form, nor that it can or has been encoded. [6] provides only a small set of generic heuristics for guidance.

In this paper, we show how the expertise encoded as architectural styles ([9, 1, 7]) can guide problem decomposition. We present this new approach through a case study in



1a By traversing the input domain in this sequence...

1b and simultaneously traversing the output domain in this sequence...

2a finding these values in the input domain structured like this...

2b and creating these values in the output domain structured like this...

3a the machine ensures that these input domain values...

3b produce these output domain values...

4 which satisfy the IO relation.

Figure 2. The Transformation Frame Concern

which the Pipe-and-Filter and Active Store styles are used to decompose the well known *Key Word In Context (KWIC)* problem ([8], elaborated in [9]).

2.3 KWIC

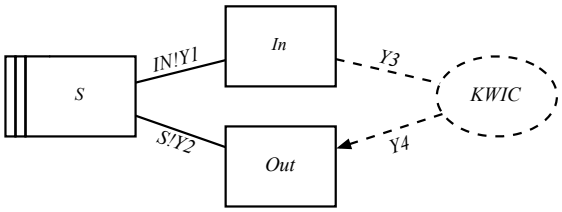
The KWIC problem is to produce the keywords for a sequence of lines, indexed by context.

By analysis of the problem's context, the Problem Frame representation of the KWIC problem, i.e., its problem diagram, is shown in Figure 3. As such, unsurprisingly, KWIC is an instance of the Transformation Frame (the reader interested at arriving at this conclusion themselves should consult [6]). (The Transformation Frame was shown in Figure 1.)

From the matching process, we know that *In* and *Out* are lexical domains (we assume text files, each line terminated by a carriage return); moreover, we have made the (arbitrary) assumption that the lexical phenomena ($Y1$ through $Y4$) are represented by text files, text lines and characters. The *KWIC* requirement: *Out* to contain the keywords for the text file *In*, indexed by context.

3 Architectural frames

To be able to exploit solution structures within the PF framework, work is required: a) to recognise their role in



$Y1 : \{File, Line, Char\}$ $Y3 : \{TextLines\}$
$Y2 : \{File, Line, Char\}$ $Y4 : \{IndexedLines\}$

Figure 3. The KWIC Problem Diagram

the problem space, and b) to notate them suitably therein. To this end, in this section we introduce the notion of an *architectural frame* (or AFrame) as a new element of the PF framework characterising the combination of a class of problems and a class of solution structures. An AFrame can be regarded as a Problem Frame (characterising a problem class) for which a standard sub-problem decomposition exists (informed by the solution class).

Problem classes under consideration are those captured by (basic) Problem Frames of the PF framework. Solution structures under consideration are architecture classes as captured by architectural styles [9, 1, 7]. (Each style characterises an architecture class through the description of: the architecture's element types and their topology; patterns of data and control among elements; and the benefits and drawbacks of the architecture.)

The characterisation of architectural styles for use within the PF framework entails ways of representing those elements of the architectural style that impact the problem description. For this paper, this means a) representing generalised topologies, and b) binding domains and phenomena. This is achieved through the notion of decomposition templates, an important part of the architecture frame definition.

In the reminder of this section we introduce two AFrames resulting from the combination of the Transformation frame with the Pipe-and-Filter and Active Store architectural styles, and we illustrate their usefulness through their application to the KWIC problem.

3.1 The Pipe-and-Filter Transformation Frame

The Pipe-and-Filter architectural style [1] sees a system as a series of filters (or transformations) on input data. Data enter the system and then flow through the components one at a time until they reach some final destination. Filters are connected by pipes that transfer data. In our favourite incarnation of this style, the linear pipeline, each filter has precisely one input pipe (its source) and one output pipe (its sink). See Figure 4 for an illustration.

Two of the major advantages of the Pipe-and-Filter style

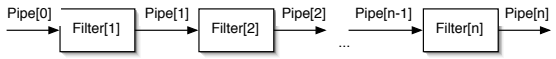


Figure 4. Linear Pipe-and-Filter with indexing

are that: 1) it is modular, each filter can be modified or replaced without affecting the other filters; and 2) many useful filters already exist, and can be reused as off-the-shelf. (Indeed, perhaps Filters have some claim to be *the* original reusable software component.)

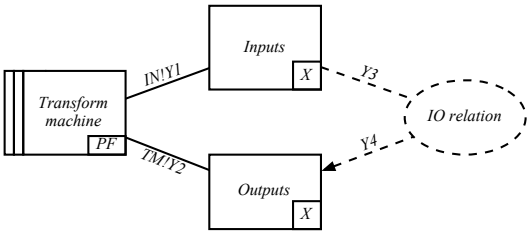
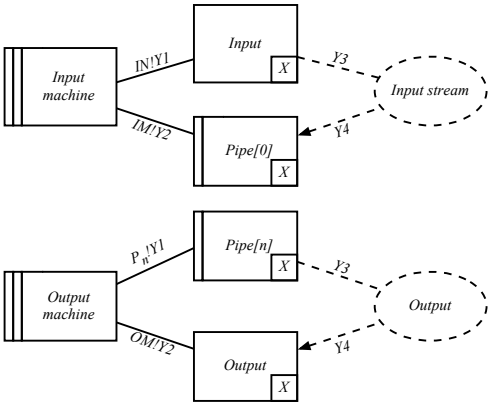


Figure 5. The Pipe-and-Filter Transformation Frame

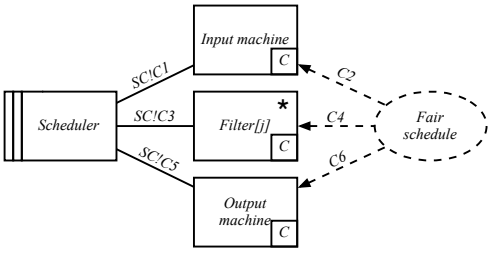
The *Pipe-and-Filter Transformation Frame* represents the class of transformation problems whose solution is to be provided through the Pipe-and-Filter architectural style. A decision to use the Pipe-and-Filter style is indicated by the annotation of the machine domain in the Transformation Frame, as illustrated in Figure 5. (The annotation is similar to other domain annotations used in Problem Frames indicating that the domain must satisfy some constraints – in this case, be structured as per the Pipe-and-Filter architectural style.)

An AFrame comes with a collection of *template diagrams* for sub-problem decomposition. For the Pipe-and-Filter Transformation Frame, there are four such template diagrams, those shown in Figure 6. They are:

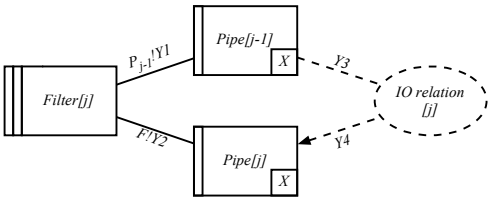
- the *input* and *output* sub-problem class: in a Pipe-and-Filter solution data are streamed between filters through pipes. The input/output sub-problem addresses the problem of converting data into suitable formats. The indexing of the 'pipe domains' provides our favourite linear topology. As designed domains (indicated by their single vertical bar) the pipe domains may have their data structures explicitly specified to solve the input/output problem.
- the *transformation* sub-problem class: this is where the essence of the transformation problem, stripped of other considerations, is found.
- the *scheduling* sub-problem class: as we work in the problem space, we can make no assumptions as to



(a) Input/Output sub-problem class

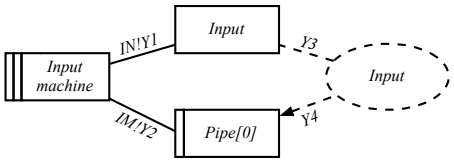


(b) Scheduling sub-problem class

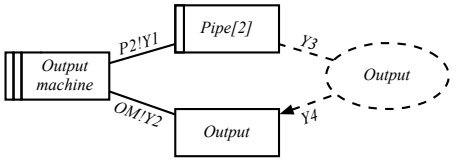


(c) Transformation sub-problem class

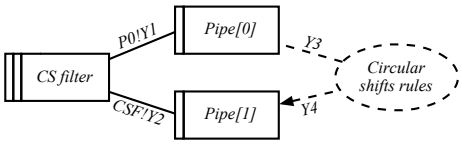
Figure 6. Decomposition templates for the Pipe-and-Filter Transformation Frame



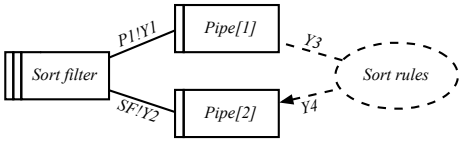
(a) Input sub-problem



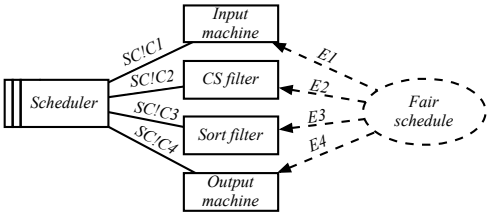
(b) Output sub-problem



(c) Circular shifts sub-problem



(d) Sort sub-problem



(e) Scheduling sub-problem

Figure 7. KWIC sub-problems decomposition through the Pipe-and-Filter Transformation Frame

hardware architecture. This means that part of the problem is the sequencing of filter transformations. In this sub-problem, therefore, we specify the requirements for fair scheduling of machine components.

Note that we annotate the Filter given domain with a star (*) to indicate that there may be many filters that share phenomena with the Scheduler.

Some specifics of this new notation: In the instantiation of an AFrame, more than one instance of the same template may need to be applied for a given problem. For instance, in a typical Pipe-and-Filter solution, many filters need to be designed. The consideration of each such filter may require a separate instance of the transformation sub-problem. The linear Pipe-and-Filter architecture is enforced by the indexing of the filters and pipes.

KWIC and Pipe-and-Filter Given that the KWIC is a transformation problem, we may apply the Pipe-and-Filter Transformation Frame to it. We know that a quick keyword index can be created by first producing circular shifts, then sorting them alphabetically, and so know that we need two filters in sequence. This leads to the sub-problem decomposition shown in Figure 7. For brevity we omit the detail of all phenomena.

In the figure, parts (a) and (b) give the problem diagrams for the input and output sub-problems; parts (c) and (d) those for the two filters; and part (e) for the scheduling. We comment on the general principles of the decomposition later in this section.

3.2 Active Store Transformation Frame

The Active Store architectural style [1] sees a system as made up of a number of independent components, which are producers and consumers of shared data.

The *Active Store Transformation Frame* represents the class of transformation problems whose solution is to be provided through an active store architectural style. In a way similar to the Pipe-and-Filter, the intention to use the Active Store is shown by an annotation on the machine domain, as shown in Figure 8. The decomposition templates for this AFrame are shown in Figure 9.

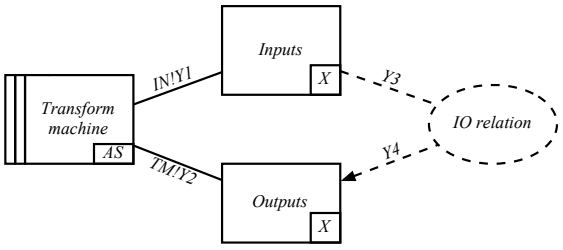


Figure 8. The Active Store Transformation Frame

There are four classes of sub-problems:

- the *input* and *output* sub-problem class: as for the Pipe-and-Filter architectural style above;
- the *subscribe* and *notify* sub-problems class: these give the mechanism for components to register their interest in the data store content and be notified when changes occur. Note that, in the figure, *Subscribers list* as a designed domain, meaning that its data structures are subject to design;
- the *transformation* sub-problem class: as for the Pipe-and-Filter architectural style, this is where the substantial design problem is to be found.
- the *data integrity* sub-problem class: by encapsulating shared data, the active store must provide mechanisms, such as mutual exclusion or locking, to guarantee the integrity of its data.

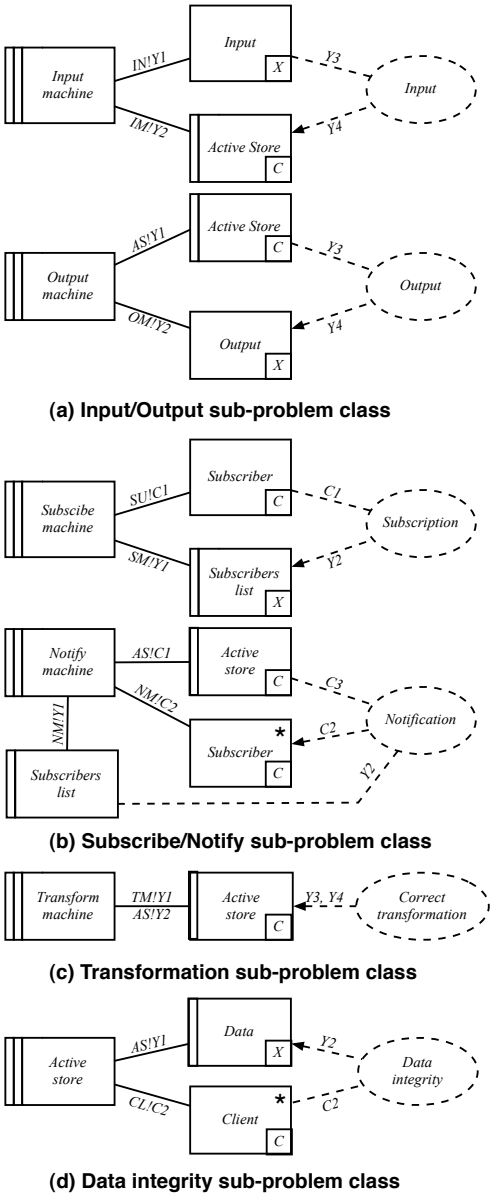
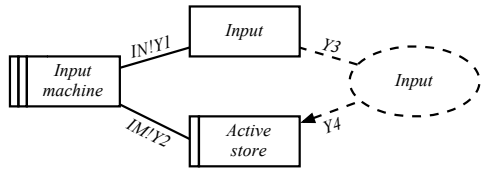


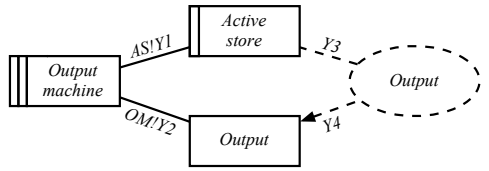
Figure 9. Decomposition templates for the Active Store Transformation Frame

KWIC and Active Stores We apply the Active Store Transformation Frame to the KWIC example, assuming that there are two transformation sub-problems (indeed these are circular shifts and sort, as for the Pipe-and-Filter case). This leads to the sub-problem decomposition shown in Fig-

ure 10. Once again, we omit detail of the phenomena.

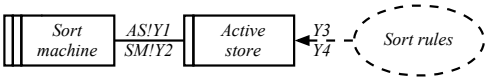


(a) Input sub-problem

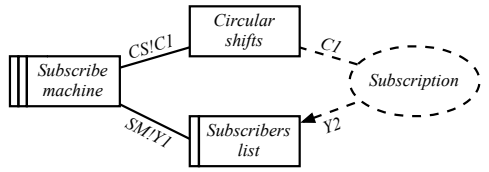


(b) Output sub-problem

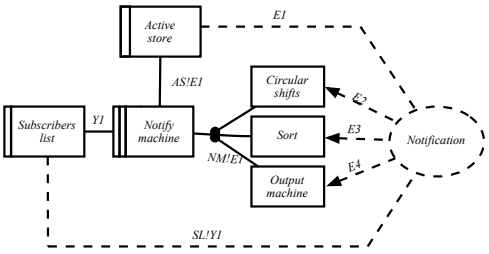
(c) Circular shifts sub-problem



(d) Sort sub-problem



(e) Subscription sub-problem



(f) Notification sub-problem

Figure 10. KWIC sub-problems decomposition through the Active Store Transformation Frame

In the figure, parts (a) and (b) give the problem diagrams for the input and output sub-problems; parts (c) and (d) those for the two transformation sub-problems; part (e) for the subscription sub-problem; and part (f) for the notification sub-problem.

3.3 Correctness

The frame concern is common to all the problems of a particular class and its structure depends on the nature of those problems. In the original PF framework, instantiation of a Problem Frame leads to a sub-problem that has the same topology, making it easy to establish the relationship between the elements of the frame concern and those of the problem diagram.

Using AFrames, we guide sub-problem decomposition through templated application of solution structures. There is an issue as to whether the original frame concern should still apply through the decomposition. The answer to this, we feel, is yes: the frame concern is motivated by the problem independently of the solution structure, and so should be dischargeable by any solution to that problem. We are then left with the task of discharging the original frame concern from the properties of the solved sub-problems.

For instance, the first component of the frame concern for the Transformation Frame states (recall Figure 2):

1a By traversing the input domain in this sequence...

This has two parts: a) that an appropriate traversal of the input domain exists, and b) that it can be completed. The first part is discharged from the detail of the specifications of the solutions to the input sub-problem (given appropriate description of the pipes). The second part is discharged by the requirement of fairness on the scheduling sub-problem, which prevents any of the other machines being starved, and so failing to traverse the input domain. (*Mutatis mutandis* for 1b, i.e., the traversal of the output domain.)

In addition, deadlock between filters (in a general Pipe-and-Filter Architecture) could prevent the chosen traversal from being completed. Care must therefore be taken to show that any sub-problem solutions when recomposed will be deadlock-free. For Pipe-and-Filter, one way of doing this is to choose a linear pipeline, as we have done, from which deadlocking concerns evaporate.

3.4 Discussion

In the AFrames we have intentionally kept decomposition guidance very general so as not to second-guess choices that are properly part of design and implementation. For instance, in the Pipe-and-Filter decomposition we have not assumed that the functionality of the Input machine is merged with that of the first filter, although, should the Input be a sequential stream, this would certainly have been possible. Moreover, for this decomposition the scheduling problem is explicitly given – even though for particular choices of pipe, it might be trivial – whereas it is subsumed by the subscribe and notify sub-problems in the Active Store decomposition.

In addition, although the templates identify many sub-problem classes, it is not always the case that each generated sub-problem will require the same in-depth consideration. For instance, for the Pipe-and-Filter decomposition, a Sorting filter may be available off-the-shelf so that its sub-problem will not require further analysis. Similarly, in the case of the Active Store decomposition, the active store itself is likely to be a component for which the data integrity concern has already been addressed.

In the case study we have seen how AFrames can be used to guide sub-problem decomposition. Also we have seen that the details of the decomposition depend on the solution structure at which we are aiming, as we might expect, and that different sub-problems emerge from the different architectures. Finally, we have discussed how properties of the sub-problems contribute to the discharge of the frame concern.

4 Related work

The structuring of problem domains and associated requirements has been the subject of considerable work in recent years. Early work by Parnas and colleagues proposed a four-variable model [16] upon which the SCR requirements specification approach and associated tools were developed [15]. SCR shares much in common with the Problem Frames approach, but its main focus is on event-based (control) systems, and its tabular notation has not, to our knowledge, been used to incorporate architectural considerations.

A number of goal-based requirements approaches, most notably KAOS [18] and the NFR framework [13], have proposed the explicit use of 'goals' to partition the problem domain and associated requirements. More recent work by Letier and van Lamsweerde addresses the identification of agents (e.g., components), and their assignment to goals [19, 18], but it is not clear how existing architectural choices can influence the problem structuring process. Similarly, the NFR framework uses high-level goals to initiate a process of identification of associated design components, but does not explicitly consider how such components can inform the goals descriptions identified.

A very common approach to structuring requirements specifications is still the use of pre-defined templates or standards that prescribe how requirements specification documents should be partitioned [21, 17]. This form of template-based structuring is common for requirements expressed in natural language. The technique allows both requirements and design information to be included in the descriptions, but it is difficult to enforce clear separation between descriptions of problem domain from those of the solution domain.

The structuring of the solution domain has also been investigated thoroughly. Most research on software architectures has focused on the use of components and connectors as structuring concepts [22], however, with a few notable exceptions [1] [25], very little consideration has been given to relating such software architectures back to requirements in the problem domain.

The relationship between requirements and architectures has received increased attention recently [11] [12], although these approaches have not explicitly focused on problem decomposition as their goal. Brandozzi and Perry [10] have suggested the use of intermediate descriptions between requirements and architecture that they call 'architectural prescriptions', which describe the mappings between aspects of requirements and those of an architectural description. Recent work on software product lines and system families has focused on identifying core requirements (identified perhaps through a process of requirements prioritisation) and linking them to core architectures (identified perhaps by examining the stability of various architectural attributes over time) [23]. Wile [24] has examined the relationship between certain classes of requirements and their corresponding dynamic architectures, to enable requirements engineers to monitor running systems and their compliance with these requirements. Finally, Grunbacher *et al* [14] explore the relationships between software requirements and architectures, and propose an approach to reconciling mismatches between requirements terminology and concepts with those of architectures.

5 Conclusions

In this paper, we have shown how to use solution structures to guide problem decomposition.

We have defined two AFrames corresponding to Pipe-and-Filter and Active Store architectural styles as applied to transformation problems. We have applied these AFrames to the KWIC problem to produce detailed sub-problem decompositions. We have noted that the two AFrames lead to the identification of different sub-problems. We have also discussed how properties of the sub-problem solutions contribute to the discharge of the frame concern.

Underpinning our approach is the observation that solution structures can be regarded as indicative properties of the machine domain. We mean by this that they indicate a recognised truth about the solution, which we need to take into account when specifying a machine's behaviour. As we have shown with AFrames, the result is a combination of problem analysis techniques and solution-oriented practices. In particular, the characterisation of solution structures in the domain leads to a specific way of binding domains and phenomena, which constrains the way sub-problems can be constructed and analysed. This has the advantage of separating problem-specific from architectural

issues, hence bounding sub-problems more precisely.

In this paper we have chosen to combine Problem Frames and architectural styles. In the future, we wish to extend the approach to other combinations of problem and solution structures. This task has at least two parts. The first is the identification of other AFrames corresponding to combinations of basic Problem Frames and other architectural styles. For instance, an obvious combination is the Information Display Frame and the MVC, and this may be the source of another AFrame.

The second is to generalise from architectural styles to other solution structures as the guidance for decomposition. For instance, we see no particular difficulty in capturing the combination of problem diagrams and components given in [4] using the ideas of this paper.

Other work will consider what the implications of the approach are in terms of the analysis process. As suggested, among others, by the Architectural Business Cycle [1], the design of software is a process that iterates between problem and solution spaces. In this paper, we have detailed a single iteration within the problem space via the solution space. Further iterations are then possible by considering the products of the problem decomposition. In this case, a major issue is the recomposition of solutions derived from different architectures to produce a solution for the original problem; there are a number of recomposition concerns that need investigation.

6 Acknowledgements

We acknowledge the kind support of our colleagues, especially Robin Laney, in the Department of Computing, the Open University.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, SEI Series in Software Engineering, Addison Wesley, 1998.

[2] J. Cheesman, J. Daniels, UML Components: A Simple Process for Specifying Component-Based Software, Addison-Wesley, 2000.

[3] D.F. D'Souza, A.C. Wills, Objects, Components, and Frameworks with UML : The Catalysis Approach, Addison-Wesley, 1998.

[4] J.G. Hall, M. Jackson, R.C. Laney, B. Nuseibeh, L. Rapanotti, Relating Software Requirements and Architectures using Problem Frames, IEEE Proceedings of RE 2002, 2002.

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

[6] M. Jackson, Problem Frames, ACM Press Books, Addison Wesley, 2001.

[7] M. Klein, R.Kazman, Attribute-based architectural styles, Technical Report CMU/SEI-99-TR-022 ESC-TR-99-022, October 1999.

[8] D.L. Parnas, "On the criteria to be used in decomposing systems into modules", Communications of ACM, 15(12):1053-1058, December 1972.

[9] M. Shaw, D. Garlan, Software Architecture, Prentice Hall, 1996.

[10] M. Brandozzi, D.E. Perry, Transforming Goal Oriented Requirement Specifications into Architectural Prescriptions. In [11].

[11] J. Castro J. Kramer (eds), Proceedings of First International Workshop From Software Requirements to Architectures (STRAW'01), Toronto, Canada, 2001.

[12] D. Berry, R. Kazman, R. Wieringa (eds), Proceedings of Second International Workshop From Software Requirements to Architectures (STRAW'03), Portland, USA, 2003.

[13] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos, "Non-functional Requirements in Software Engineering", Kluwer Academic Publishers, 2000.

[14] P. Grunbacher, A. Egyed, N. Medvidovic, "Reconciling Software Requirements and Architectures: The CBSP Approach", Proceedings of the 5th International Symposium on Requirements Engineering (RE'01), pp.202-211, IEEE CS Press, 2001.

[15] C.L. Heitmeyer, R.D. Jeffords, B.G. Labaw, "Automated Consistency Checking of Requirements Specifications", ACM Transactions on Software Engineering and Methodology, 5(3):231-261, 1996.

[16] K. Heninger, D.L. Parnas, J.E. Shore, J.W. Kallander, "Software Requirements for the A7E aircraft", TR3876, Naval Research lab, Washington, DC, 1978.

[17] B.L. Kovitz, Practical Software Requirements: A Manual of Content and Style, Manning Publications Company, 1998.

[18] A. van Lamsweerde, "Goal-Oriented requirements Engineering: A Guided Tour", Proceedings of the 5th International Symposium on Requirements Engineering (RE'01), pp.249-261, IEEE CS Press, 2001.

[19] E. Letier and A. van Lamsweerde, "Agent-based Tactics for Goal-Oriented Requirements Elaboration", Proceedings of 24th International Conference on Software Engineering, ACM Press, May 2001.

[20] D.L. Parnas, J. Madey, "Functional Documentation for Computer Systems", Science of Computer Programming, 25(1):41-6, Oct 1995.

[21] S. Robertson, J. Robertson, Mastering the Requirements Process, Addison Wesley, 1999.

[22] M. Shaw, G. Garlan, Software Architecture: Perspectives on an emerging discipline, Prentice Hall, 1996.

[23] D.M. Weiss, C.T.R. Lai, Software Product Line Engineering,: A Family-Based Software Development Process, Addison-Wesley, 1999.

[24] D. Wile, "Residual Requirements and Architectural Residues", Proceedings of the 5th International Symposium on Requirements Engineering (RE'01), Toronto, Canada, pp.194-201, IEEE CS Press, 2001.

[25] P. Zave, From Architecture to Requirements: A Success Story, In [11].