# 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 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 through 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, decomposition and subsequent recomposition, within the Problem Frames framework. In particular, we introduce* Architectural Frames – *combinations of architectural styles and Problem Frames – and illustrate their use by applying them to two problems from the literature.*

## 1   Introduction

Problem Frames [23, 24] 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 [6] or within existing frameworks [11] and architectures [32, 1]. Finally, the expertise of the developer in specific domains is also the subject of reuse [13].

Whereas Problem Frames are used only in the problem space, we observe that each of these competing views uses 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, 32], located in the solution space, to guide the analysis of the problem space. 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. Their use is to guide problem decomposition, a fundamental part of the analysis process, and solution recomposition, a fundamental part of the synthesis process. The approach is validated through the application of AFrames to two problems from the literature. The work here builds on that reported in [16], where we focused on component-based development.

The paper is organised as follows. Section 2 provides background information and motivation for the work. Section 3 introduces the notion of Architectural Frame (AFrame) and the associated decomposition techniques. Sections 4 and 5 introduce two AFrames and their application to examples from the literature. Section 6 includes some reflections on the work, and Section 7 concludes the paper.

1

## 2 Background and motivation

Problem Frames [23, 24, 27] are an increasingly popular framework for requirements engineering [9]. They identify and characterise basic classes of problems that recur throughout software development, for reuse in requirements analysis. Each class is captured by a Problem Frame, an initial catalogue being provided in [24]. Problem characterisation is through the identification and description of relevant problem domains and shared phenomena. A diagramatic notation is used for problem representation, cataloguing, documentation and communication. Their semantic foundation characterises their use in requirements engineering [15], and allows for the construction and discharge of correctness arguments [17, 19].

Essential elements of a problem analysis process based on Problem Frames are the building of descriptions for domains, phenomena and requirements, followed by problem decomposition and recomposition. Problem decomposition is through templated applications of *problem frame diagrams*, a process reminiscent of analysis patterns [12], leading to grounded instances. The result of the instantiation is a *(sub-)problem diagram*, which can either be further analysed and decomposed, or is in a form which admits solution. Problem recomposition combines sub-problem solutions by ensuring that destructive interference is not created.

While recognising the conceptual clarity and semantic richness of the Problem Frames framework, we acknowledge certain criticisms raised of the approach.

Firstly, a 'green-field' development process is assumed in which problem analysis is made without reference to any solution space structures, such as existing components, architectural styles, *etc*; this is at odds with current software development practice. In [24], the machine is assumed to be a general purpose computer – essentially, a Turing machine – and all development is *ab initio*.

Secondly, Problem Frames rely on the expertise of practitioners to understand which decompositions (and subsequent recomposition) of a problem into its constituent sub-problems are appropriate and will lead to a 'good solution'. This expertise is not assumed to have any particular form, nor that it has, or indeed can be encoded; in [24] there is only a small set of generic heuristics for decomposition guidance.

Thirdly, Problem Frames work from fixed descriptions of problem domain artefacts and phenomena. In real life, problems do not stay still in this way, and a necessary development skill is the tracking of changing problems.

Problem Frames are unlikely to become a part of mainstream software development if such concerns cannot be addressed satisfactorily.

In this paper, we address the first two of these criticisms, leaving the third to other strands of our work (see [4]). In particular, we show how to use solution structures – in this case, architectural styles – to aid problem analysis, problem decomposition and solution synthesis through recomposition. For this to happen, work is required, as described in this paper: a) to recognise the role of solution structures in the problem space; b) to notate them suitably therein; and c) to develop problem frame analysis, decomposition and recomposition techniques which exploit that annotation. This reflects into the problem domain via Problem Frames the important work of others, including [32, 1, 25]. In previous work, the authors have shown how such techniques can work when components are available: [16] looked at integrating Problem Frames and component-based development.

We relate the first two criticisms above to limitations of the original Problem Frame framework. These are that:

- the machine, which is the object of design, is considered an unstructured general purpose computing device;

- there is little specific problem decomposition guidance;

- there is little specific solution recomposition guidance.

We address these limitations in this paper.

The structuring of problem problem domains and associated requirements has been the subject of considerable work in recent years [22, 21, 34, 27, 37, 7, 36]. Similarly, The structuring of the solution domain has also been investigated thoroughly [35, 1, 40]. However, the relationship between requirements and architectures has only recently become the object of attention [5, 2], and none of the approaches reported in the literature has explicitly focused on problem decomposition as their goal. Brandozzi and Perry [3] 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) [38]. Wile [39] 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.

# 3 Introducing Architectural Frames

We introduce the notion of an *Architectural Frame* (or *AFrame* for short) as a new element of the Problem Frames framework. The intention of AFrames is to provide a practical tool for sub-problem decomposition and recomposition that allows the Problem Frames practitioner to separate and address, in a systematic fashion, the concerns arising from the intertwining of problems and solutions. The rationale behind AFrames is the recognition that solution structures can be usefully employed to inform problem analysis.

An AFrame captures the combination of a class of problems, represented by a Problem Frame, and a class of solution structures, represented by architectural styles [32, 1, 25]. Notationally, an AFrame shares the Problem Frame diagram of the corresponding problem class, augmented by a further annotation of the machine domain, which records the intention of producing a problem solution which is based on a particular architectural style (we will illustrate AFrame diagrams in the following sections).

Architectural styles characterise software architecture classes in terms of the architecture's element types and their topology, and the patterns of data and control among elements. The characterisation of architectural styles for use within the Problem Frames framework entails ways of representing those architectural elements that impact the problem description. In this paper, this means representing generalised topologies, and binding domains and phenomena. This is achieved through *decomposition templates*, an integral part of every AFrame definition. Decomposition templates capture a standard and systematic way of decomposing the problem into sub-problems; they complement classical Problem Frames decomposition by providing further guidance and decomposition rules. Notationally, they are frame diagrams, augmented, in some cases, with an indication of domain multiplicity (again, we will illustrate the notation in the following sections).

An essential step in problem analysis is addressing recomposition concerns once sub-problem decomposition and analysis is completed. AFrames simplify the recomposition step by providing an indication of how sub-problems fit together and how the frame concern of the problem class distributes across those sub-problems. That the frame concern should also be addressed at recomposition is due to the fact that the argument 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 frame concern from the properties of the solved sub-problems.

# 4 A Pipe-and-Filter AFrame

The machinery of AFrames, reflecting architectural styles in the problem domain, must cope with their complexity there. The Pipes-and-Filters style is familiar enough to provide a convenient vehicle for their introduction. Later in the paper, we develop a more realistic (and useful) AFrame for the Model-View-Controller (MVC) style.

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. A common topology of architectures based on this style, and one that we consider in this section, is the linear pipeline, in which each filter has precisely one input pipe (its source) and one output pipe (its sink). See Figure 1 for an illustration.
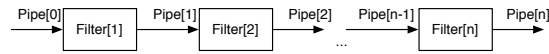


**Figure 1. The Linear Pipe-and-Filter**

The benefits and uses of Pipes and Filters are well known.

To relate Pipes and Filters to Problem Frames: Pipes and Filters 'solve' transformation problems, i.e. transforming some input into some output of a particular format, applying certain rules in the process. Transformation problems are captured in Problem Frames by the Transformation Frame [24]. Their frame diagram is shown in Figure 2. Its correctness argument is given in Figure 3. (Please refer to [24] for definitive details on the form and use of correctness arguments.)
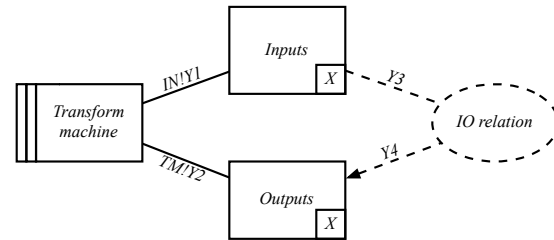


**Figure 2. The Transformation Frame Diagram**

The *(Linear) Pipe-and-Filter Transformation AFrame*[1] represents *the class of transformation problems whose solution is to be provided through the Pipe-and-Filter architectural style*. There are three components to the AFrame: the *AFrame diagram* – an annotated Problem Frame diagram,

---

[1]For simplicity in this paper, we restrict to a linear Pipe-and-Filter.
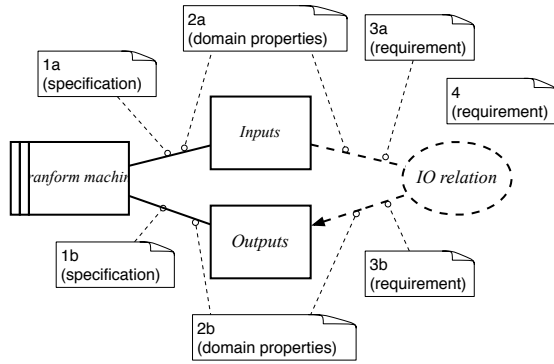
1a By traversing the input domain in this sequence...

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 3. The Transformation Frame Concern**

shown in Figure 4; a collection of *decomposition templates* – shown in Figure 5; and a *correctness argument* to determine correct recomposition – shown in Figure 6.
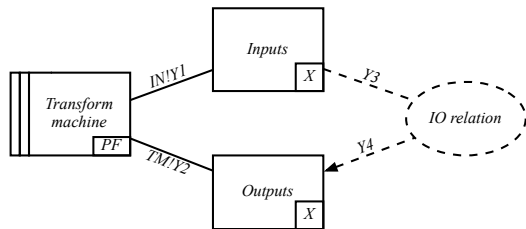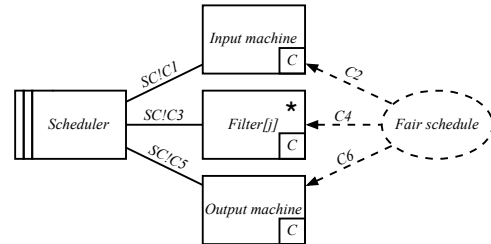


**Figure 4. The Pipe-and-Filter Transformation AFrame**

The AFrame diagram records the decision to use the Pipe-and-Filter style. For the Pipe and Filter Transformation Aframe, the AFrame diagram is an annotated Transformation Frame. The annotation is used in the same way as other problem domain annotations in that it indicates the domain must satisfy some constraints – in this case, be an instance of the Pipe-and-Filter architectural style.
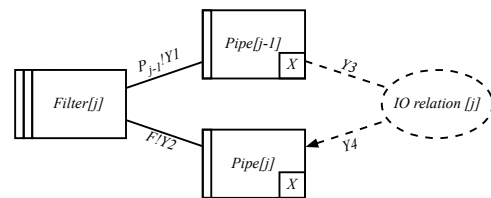
The decomposition templates for the Pipe and Filter Transformation AFrame, shown in Figure 5, identify the following sub-problems as needing consideration *if* the Pipe-and-Filter Architectural style is to be the chosen so-
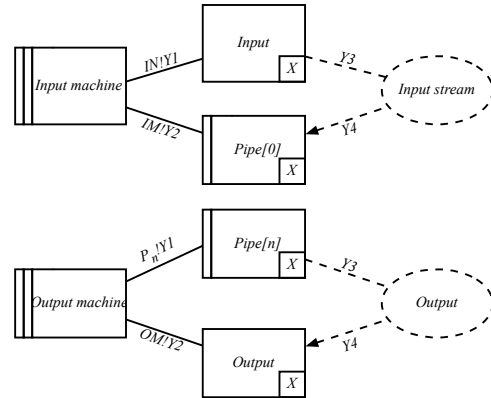


**(a) Input/Output sub-problems**



**(b) Scheduling sub-problem**



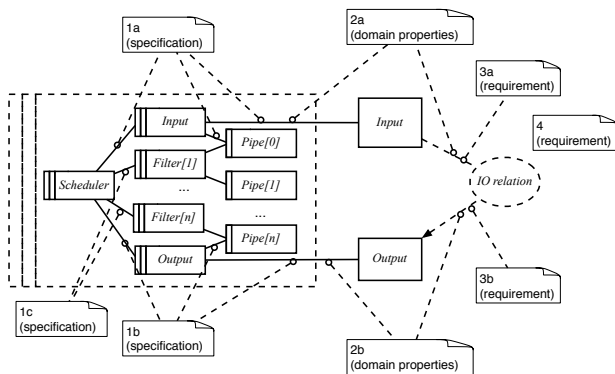**(c) Transformation sub-problem(s)**

**Figure 5. Decomposition templates for the Pipe-and-Filter Transformation AFrame**

lution architecture:

- the *input* and *output* sub-problems: 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 the linear topology. As *designed domains* [24], (indicated by their single vertical bar) the pipe domains reside within the solution machine, hence may have their data structures explicitly specified to solve the input/output problem.

- the *transformation* sub-problem: this is where the essence of the transformation problem, stripped of other considerations, is found.

- the *scheduling* sub-problem: as we work in the problem space, we can make no assumptions as to the hard-

ware architecture. This means that part of the problem is the scheduling of filter transformations[2]. In this sub-problem, therefore, we specify the requirements for fair scheduling of machine components.

In use, the sub-problems provide the decomposition guidance. Instantiation of an AFrame may require more than one instance of the same template to be applied for a given problem. For instance, in a typical Pipe-and-Filter solution, many applications of the filter template will be needed. 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.



**Figure 6. Pipe-and-Filter recomposition**

After appropriate filters have been found, the synthesis step of recomposing the solutions to sub-problems addresses the overall transformation problem, with the relationship between the sub-problem machines considered in terms of their shared phenomena. The topology of the recomposition is informed by the architectural style, which also influences the form of the resulting overall frame concern. The topology for the recomposed AFrame is illustrated in Figures 6. The correctness argument is derived from that of the transformation frame of Figures 3 by the addition of the following step:

1c knowing that the transformations occur on this way...

The new frame concern requires that an appropriate traversal of the input and output domains exists, that it can be completed, and that the composition of the individual filters, which solve the sub-problems, compose to produce the original transformation. 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 machine from being starved, and so failing to traverse the domains. The third part is discharged by appropriate design (or choice, where they exist already) of filter components.

In addition, in a general Pipe-and-Filter Architecture deadlock between filters 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. Of course, by choosing a linear pipeline, as we have done, we avoid deadlocking concerns. In the general case, this would be done by a correctness argument for recomposing the sub-problem solutions, based on any appropriate architectural constraints.

## 4.1 Using AFrames

Consider a simple but well known transformation problem from the literature: the 'keywords in context' (KWIC) problem ([31]). The problem is to produce the keywords for a sequence of lines, indexed by context. We show how the application of the (linear) Pipe-and-Filter Transformation AFrames could lead the developer to the classical solution to the KWIC problem (for instance, [32]) highlighting the assumptions that are needed to reach this solution.
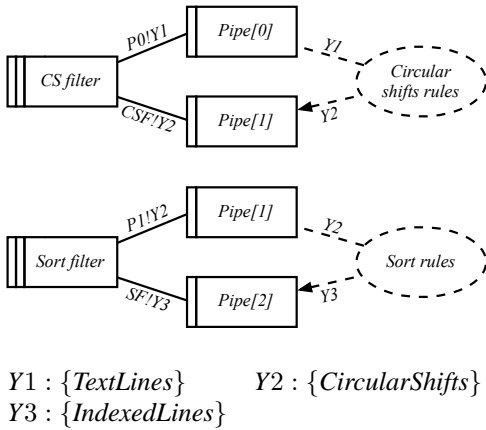
The first non-trivial application (i.e. with $n > 0$) of the (linear) Pipe and Filter solution to this problem, produces two (filter) sub-problems, leading to the sub-problem decomposition shown in Figures 7 (the filters sub-problems) and 8 (the input and output subproblems). Together with this would be an instantiated scheduling sub-problem; one benefit of the AFrames approach is that sub-problems that have generic solutions – such as scheduling – are highlighted, but do not require further analysis. From the filter sub-problems and the correctness argument, we know we require two filters whose composition is the KWIC transformation. These must either be designed or, more likely, reused; of course, AFrames, as representatives of architectural styles, cannot remove the need for creativity in the application of the style.

For completeness of the example, we have assumed that the developer has available a box of filters for reuse, including a circular shift filter (*CS*) and a sort filter (*Sort*), and can reason that the *KWIC* transformation is equal a *CS* transformation followed by a *Sort* transformation. This removes the need for further problem analysis in this paper.

Finally, in the figures, we assume that phenomena have been chosen so that the correctness argument can be discharged. In particular:

- input and output are text files, made up of text lines and characters;

- all pipes have enough capacity to contain the result of

---

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

intermediate transformations.



$Y1 : \{TextLines\}$      $Y2 : \{CircularShifts\}$
$Y3 : \{IndexedLines\}$

**Figure 7. KWIC filters sub-problems through the Pipe-and-Filter Transformation AFrame**



$Y1 : \{File, Line, Char\}$    $Y2 : \{TextLines\}$
$Y3 : \{CircularShifts\}$     $Y4 : \{File, Line, Char\}$
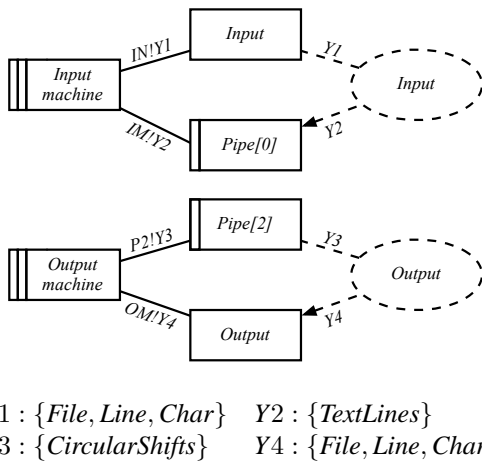
**Figure 8. KWIC input and output sub-problems through the Pipe-and-Filter Transformation AFrame**

## 5 An MVC AFrame

In this section we consider a second AFrame, resulting from the combination of control problems and the MVC solution architecture.

The MVC (short for Model-View-Controller) (see e.g., [1]) is a way of structuring a software solution into three parts – a model, a view, and a controller – to separate and handle concerns related, respectively, to the modelling of a

domain of interest, the visual feedback to the user, and the user input. The controller interprets user inputs and maps them into commands to the model to effect the appropriate change. The model manages one or more data elements, responds to queries about its state, and responds to instructions to change state. The view is responsible for feedback on the model's state to the user. Standard communication patterns (e.g., the Observer pattern [13]) apply between the MVC parts.

We consider the class of control problems with feedback to the operator, which are captured by the User Commanded Behaviour Frame [18]. More precisely, the problem is that of building a machine that will accept the user's commands, impose control on some part of the physical world accordingly, and provide suitable feedback to the user[3]. The User Commanded Behaviour Frame is illustrated in Figure 9.
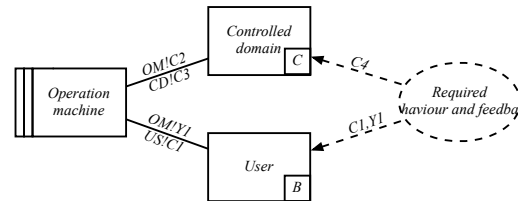


**Figure 9. User Commanded Behaviour Frame**

The frame concern for the User Commanded Behaviour Frame is given in Figure 10. From the figure you will notice that the argument has two parts: satisfying the required behaviour of the domain (from 1 to 4); and providing suitable feedback to the user (5 and 6).

By combining MVC and User Commanded Behaviour Frame we obtain an AFrame which represents *the class of user commanded behaviour problems with feedback for which an MVC solution is to be provided*. Again, the intention of using the MVC in the solution space is recorded through an annotation of the machine as illustrated in Figure 11. Guidance on decomposition is again in the form of decomposition templates, which are applied to obtain sub-problem diagrams. The decomposition templates for the MVC AFrame are given in Figure 12.

It can be seen from the figure that the original problem is decomposable into two sub-problems, whose machine domains are the View and Controller machines (in the MVC sense). Also, a Model domain is introduced which represents a faithful abstraction of the domain to be controlled. This is a designed domain, i.e. a domain designed

---

[3][24] introduces a subclass of this frame, the Commanded Behaviour Frame, which does not require the user to receive any feedback.
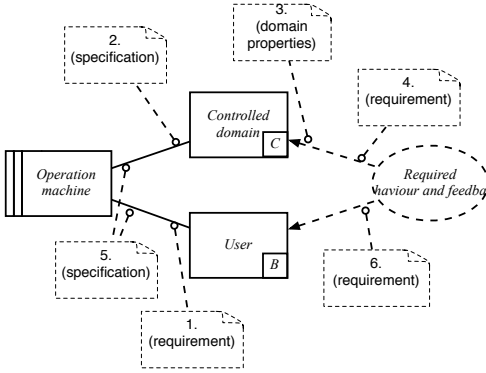
1. Given a choice of commands in the current state, when the user issues this command (it may or may not be sensible)..

2. if sensible or viable the machine will cause these events...

3. resulting in this state or behaviour...

4. which satisfies the requirement...

5. and which the machine will relate to the user...

6. thus satisfying the requirement in every case.

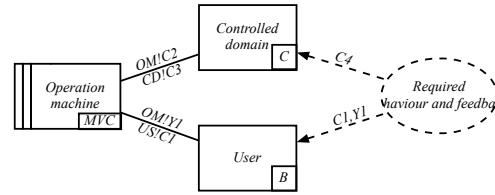**Figure 10. The frame concern for the User Commanded Behaviour Frame**



**Figure 11. MVC annotation of the User Commanded Behaviour Frame**
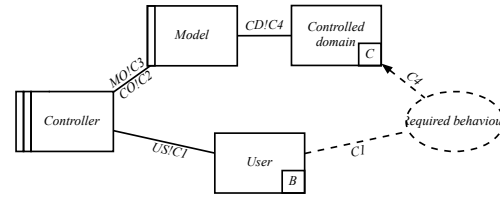


**(a) Controller sub-problem**



**(b) View sub-problem**

**Figure 12. Decomposition templates for the MVC AFrame**

as a faithful representation of the domain of interest, which will reside inside the solution machine[4] The resulting sub-problems are then: that of building a View machine to display the Model's representation of the state of the controlled domain; and that of building a Controller machine that acts on the Model, which will pass on the commands to the controlled domain. In Problem Frames terms, the Model acts as a *connection domain* [24] between the real-world domain and presentation and control subsystems.

The recomposition diagram for the AFrame, together with the correctness argument, is given in Figure 13.

## 5.1 An example

We consider the following example of a user commanded behaviour problem (derived from that reported in [29, 10, 20]). A computer system is required to control the safe and efficient operation of the catalyst unit of a chemical reactor. The system should allow an operator to issue commands for activating or deactivating the catalyst unit, and to monitor outputs. The application of the MVC AFrame to
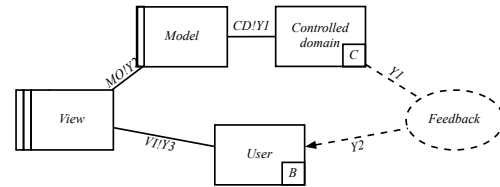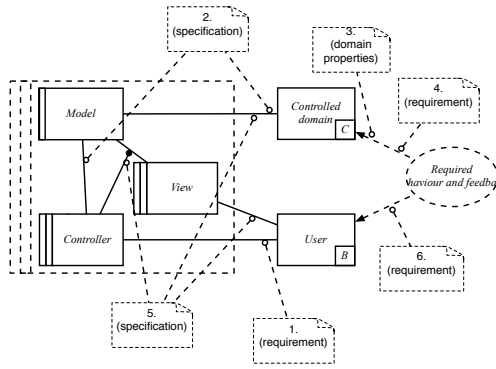
this problem results in the decomposition of Figure 14.

A possible description of the interaction rules could be as follows. The machine must allow the user to control the catalyst under the following constraints:

1. *catalyst_status* is a faithful representation of the state of the catalyst

2. the initial state of the catalyst is *catalyst_closed*

3. possible user commands are *open_catalyst* or *close_catalyst*

4. allowed state transitions are represented in Figure 15.

## 6 Discussion

We see at least two strengths of AFrames. The first is that they suggest how a problem would need to be restructured to fit a particular solution form; for instance, in the MVC case, that an abstract model of the catalyst needs to be
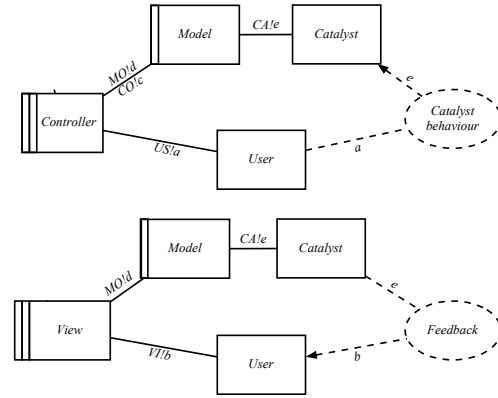
---

[4]Another MVC AFrame for the User Commanded Behviour frame is defined in [**?**]. This includes a third sub-problem template for establishing the equivalence of a model and a complex controlled domain.

1 Given a choice of commands in the current state, when the user issues this command (it may or may not be sensible)..

2 if sensible or viable the machine, through the model, will cause these events...

3 which, given a faithful model, results in this state or behaviour...

4 which satisfies the requirement...

5 and which, given a faithful model, the machine will relate to the user...

6 thus satisfying the requirement in every case.

**Figure 13. Correctness argument in MVC recomposition**



$b : \{catalyst\_status\}$  $\qquad a : \{open\_catalyst, close\_catalyst\}$
$d : \{is\_open, is\_closed\}$  $\qquad c : \{open\_catalyst, close\_catalyst\}$
$e : \{open, closed\}$

**Figure 14. MVC decomposition of the user commanded behaviour sub-problem**



**Figure 15. State machine model for the catalyst**

produced. This begins to address the 'green site' criticism that we mentioned previously. The second is that they help in solution synthesis by guiding the recomposition of sub-problem solutions into a solution for the original problem. Recomposition is facilitated through knowledge encoded in the links among architectural elements that are exploited during AFrame decomposition. This begins to address the previously mentioned 'need for expertise' criticism.
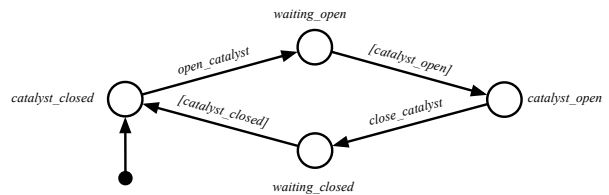
In deriving decomposition and recomposition guidance from architectural styles (and, we assume, architectures, and components too) AFrames offer to extend greatly the usefulness of the Problem Frame framework. We expect that the return on the investment of time to encode architectural artefacts as AFrames will be good, given the promise of Problem Frames, and the richness of Architectures.

AFrames, like Problem Frames, make no unreasonable assumptions of machines or humans: they are not, for instance, a substitute for creativity. Nor do they constrain the design process unreasonably; AFrames provide very general decomposition guidance so as not to second-guess choices that are properly part of design and implementation. In addition, an AFrame is an annotated machine domain to-

gether with a fixed number of sub-problem templates. The complexity of any AFrame is therefore constant. And yet, over the appropriate problem class, the decomposition guidance that an AFrame offers scales to complex problems of that class.

In addition, as we have seen, AFrames support reuse of artefacts and designs: although templates can identify many sub-problem classes, the generated sub-problems will, in being motivated by existing solution space structures, correspond to existing solution space artefacts; even in the worst case, they will have produced sub-problems that are closer to solution (with recomposition, through the architecture, being the basis of the discharge of the correctness argument).

# 7 Conclusions and Future Work

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

This work specifically addresses three limitations of Problem Frames by providing a way of interpreting existing solution structures within the Problem Frames framework. We note that this is much more systematic and general than, say, providing ad-hoc guidance for decomposition and recomposition. We believe that the translation of existing architectural styles to the problem domain will bring with it much of the related development expertise. These will then be available to strengthen the promise of the Problem Frames framework.

We have defined two AFrames corresponding to Pipe-and-Filter and MVC architectural styles as applied to transformation and control problems, respectively, and applied them to two problems from the literature, to produce detailed sub-problem decompositions. We have also discussed how properties of the sub-problem solutions contribute to recomposition and the discharge of the frame concern.

We have argued that the introduction of AFrames into the Problem Frames framework addresses some of the criticisms made of Problem Frames. In doing so, we have brought the benefits of a problem focus closer to the mainstream of software development.

Future work will extend the approach presented in this paper in many directions. Firstly, the scope of the work must be widened to include generic software architectures and realistic applications. Also, to become practically relevant, the approach must be demonstrated to be both useful and usable. A measure of its usefulness and usability must be developed and the approach validated empirically, above and beyond the limited validation offered in this paper.

Secondly, the relation between the provided architectural analysis techniques in the problem space, and more traditional trade-off analysis techniques in the solution space [8] will need exploring, in particular with a focus on the non-functional requirements and emergent non-functional characteristics of candidate designs.

Finally, the implications of the approach in terms of the analysis and synthesis processes will need considering. As already noted in the literature (see, e.g., [1, 30]), 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.

# 8 Acknowledgements

# References

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

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

[3] M. Brandozzi, D.E. Perry, Transforming Goal Oriented Requirement Specifications into Architectural Prescriptions. In [5].

[4] J. Brier, L. Rapanotti, J.G. Hall, "Problem Frames for Socio-technical Systems: predictability and change, in [9].

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

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

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

[8] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures: Methods and Case Studies, SEI Series in Software Engineering, Addison Wesley, 2001.

[9] K. Cox, J.G. Hall, L. Rapanotti (eds.), Proceedings of the 1st International Workshop on Advances and Applications of Problem Frames, ICSE Workshop, Edimburgh, 2004.

[10] O. Dieste and A. Silva, Requirements: Closing the Gap between Domain and Computing Knowledge, Proceedings of SCI2000, Vol. II (Information Systems Development), 2000.

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

[12] M. Fowler, Analysis Patterns, Addison Wesley, 1997.

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

[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.A. Gunter, E.L. Gunter, M. Jackson, P. Zave "A reference model for requirements and specifications", IEEE Software, 17(3):37-43, 2000.

[16] 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.

[17] J.G. Hall, L. Rapanotti, A Reference Model for Requirements Engineering, IEEE Proceedings of RE 2003, 2003.

[18] J.G. Hall, L. Rapanotti, Problem Frames for Socio-Technical Systems, Requirements Engineering for Socio-technical systems, J.L Mateè and A. Silva (eds.), Idea Group Inc., 2004.

[19] J.G. Hall, L.Rapanotti, Towards a semantics of Problem Frames, The Open University, Faculty of Mathematics and Computing, Research Report No. 2003/01, 2003.

[20] J.G. Hall and A. Silva. A requirements-based framework for the analysis of socio-technical system behaviour. In Proceedings of 9th International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ03), pp. 117-120, 2003.

[21] 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.

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

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

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

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

[26] M. Klein, R.Kazman, L. Bass, S.J. Carriere, M. Barbacci, H. Lipson, Attribute-based architectural styles, Proceedings of the First Working IFIP Conference on Software Architectures, San Antonio, TX, February 1999.

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

[28] 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.

[29] N. Leveson and C. Turner, An investigation of the Therac-25 accidents, Computer, Vol. 26, No. 7, pp. 18-41, 1993.

[30] Nuseibeh, B.A., "Weaving Together Requirements and Architecture", IEEE Computer, 34(3):115-117, March 2001.

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

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

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

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

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

[36] A. Sutcliffe, The Domain Theory: Patterns for Knowledge and Software Reuse, Lawrence Erlbaum Associates, 2002.

[37] 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.

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

[39] 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.

[40] P. Zave, From Architecture to Requirements: A Success Story, In [5].