Relating Software Requirements and Architectures using Problem Frames

Jon G. Hall Michael Jackson Robin C. Laney Bashar Nuseibeh Lucia Rapanotti
The Open University, UK
{J.G.Hall, M.A.Jackson, R.C.Laney, B.A.Nuseibeh, L.Rapanotti}@open.ac.uk

Abstract

Problem frames provide a means of analyzing 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.

However, even modestly complex problems can force us into detailed consideration of the architecture of the solution. This is counter to the intention of the problem frames approach, which is to delay consideration of the solution space until a good understanding of the problem is gained.

We therefore extend problem frames, allowing architectural structures, services and artifacts to be considered as part of the problem domain. Through a case study, we show how this extension enhances the applicability of problem frames in permitting an architecture-based approach to software development. We conclude that, through our extension, the applicability of problem frames is extended to include domains with existing architectural support.

1. Introduction

Problem frames [14, 15] classify software development problems. They structure the analysis of the world in which the problem is located — the problem domain — and describe what is there and what effects one would like a system located therein to achieve. The problem frame approach provides an opportunity for practitioners to gain experience and for problem domain expertise to be built.

Partitioning knowledge in this way is also the realm of software architectures (by which we mean architectures themselves [24, 2, 1], as well as frameworks [7] and design patterns [8]). With software architectures firmly part of the solution domain, problem frames and software architectures could be regarded as complementary.

In previous work [21], we have argued that modern software development must exploit the synergy that exists between problem and solution domains, in order to allow software developers to explore requirements and design opportunities. To achieve this, we proposed the Twin Peaks model of software development — a variation of which is shown in Figure 1 — that illustrates the iterative nature of the development process. This is a process during which both problem structures and solutions structures are detailed and enriched. In this context, we would see the use of architectural support as aiding the focus on the essential design requirements of the problem by allowing design concerns to be treated more abstractly and to be combined with behavioural requirements. In this paper, we extend problem frames towards this end.

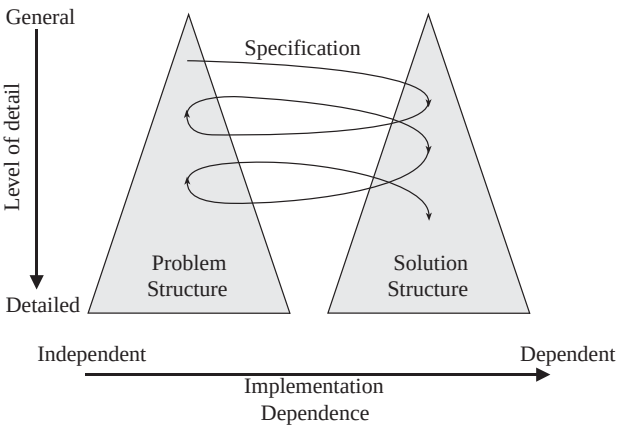


Figure 1. The Twin Peaks model: iterating between problem and solution structures

The paper is organised as follows. Section 2 introduces the relevant concepts and notation of problem frames and summarises our proposed extensions. Using a case study, Section 3 illustrates the difficulties of using problem frames to incorporate architectural design concerns during problem analysis. Our extensions are detailed and validated in Section 4. Section 5 provides a comparative account of related work. Section 6 concludes the paper with a discussion of the ideas, including outstanding issues for future work.

2. Problem Frames

Most real problems are too complex to fit within a problem identification/solution description model. They require, rather, a third level of description: that of structuring the problem as a collection of interacting subproblems, each of which is smaller and simpler than the original, with clear and understandable interactions.

Problem frames are a notation for this third level, and provide analysis techniques for progressing from problem identification to problem structuring. A problem frame defines the shape of a problem by capturing the characteristics and interconnections of the parts of the world it is concerned with, and the concerns and difficulties that are likely to arise in discovering its solution. An example of a problem frame is the *commanded behaviour* frame, illustrated in Figure 2.

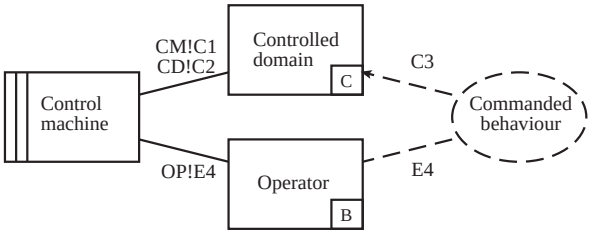


Figure 2. The commanded behaviour frame

The commanded behaviour frame is typically used when there is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by some operator. The problem is then to build a machine that will accept the operator's commands and impose control accordingly.

The components of the problem frame represent things in the real world:

- The *control machine* is the machine to be built.
- The *controlled domain* is the part of the world to be controlled. It is a *causal* domain, marked C in the diagram, its phenomena being physical and causally related. The controlled domain and the control machine share an interface consisting of two sets of phenomena: C1, controlled by the control machine (indicated by the prefix CM!), and C2, controlled by the controlled domain (prefix CD!). As we shall see, the interface acts as a conduit between the two domains. The controlled domain is an example of a *given domain*: a problem domain whose properties are given, not designed.
- The *operator* issues commands, which appear in the diagram as events E4, shared with the machine and controlled by the operator (prefix OP!). The operator is assumed to generate events in E4 spontaneously; such a domain is termed *biddable*, marked B in the diagram.
- The *commanded behaviour* is the required behaviour of the controlled domain in response to the operator's commands, E4. This required behaviour is expressed in terms of phenomena C3, which are, in general, distinct from both C1 and C2.

Problem frames are specialised to represent particular problems. In addition to more descriptive annotations on the various problem frame components, the specialised diagram also contains a legend detailing the various events and commands (jointly, phenomena). Later (Figure 3) we show the commanded behaviour frame specialised for a warehouse ordering system.

Each frame has an associated *frame concern* that must be addressed in any problem of the class. The frame concern identifies the correctness argument that must be made. For the commanded behaviour frame, this argument will exploit explicitly stated causal properties of the controlled domain to show that the machine behaviour in terms of the phenomena C1 and C2 will cause the required behaviour of the controlled domain in terms of the phenomena C3.

Using problem frames does not imply the choice of a particular development method. Even so, in this paper we structure the problem statement around the determination of particular problem frame diagrams, and identify the associated phenomena, domains and interfaces. We then describe the requirements of the frame in detail.

2.1. Extending problem frames

The problem frames approach does not address the concerns of those who work in well-known application domains; there, expertise — often expressed as software architectures — is used as the foundation of software development. The benefit is that expertise shortens development times for new systems. We observe that domain-specific solutions embody some knowledge of the problem domain, and so should be able to inform the problem analysis for new software developments within that domain. Also, these pre-packaged solutions manifest themselves as software artifacts, and so already provide part of the solution.

In [15], the aim is to design and build the machine domain by creating its software. The physical machine is assumed to be a 'general-purpose computer', to be specialised by the software; no particular structure is assumed of the machine. Being general purpose, then, classical problem frames can make no direct use of existing architectural support and expertise.

With the above observation, though, it is not unreasonable to assume that architectural support can be used to add structure to the machine domain of a problem frame. To

this end, we introduce an extension to problem frames that allows this to be achieved. We go on to show how the extension can be used to benefit from architectural support through a simple case study development of a small warehousing ordering system.

2.2. Notational conventions

To present our case study we must be able, succinctly, to represent various requirements statements. To do this we will make informal use of an event model similar to that of the causal logic of [20]. We make no claims of the notation other than convenience of expression.

Succinctly, an event instance is a point in time. Events are grouped (by event name) into event classes, from which event instances are taken. Two event instances of the same class e will be distinguished through parametrisation. The formula $e \wedge [p]$ means 'at the time that event instance e occurred, the predicate p held'. We also define, for event instances a and b in appropriate event classes:

- $a \rightsquigarrow b$ (read as a leads to b) the occurrence of a is sufficient and necessary to cause the occurrence of b;
- $a \# b$ (read as a before b): the occurrence of a is before the occurrence of b; and
- $a \hat{\#} b$ (read as a immediately before b): the occurrence of a is before the occurrence of b, and no other event instance of the class of a or b will occur in between.

For semantics, we need only observe that these operators constrain traces of event instances of a system: $a \rightsquigarrow b$ says that a will not occur in a trace without an associated b, and *vice versa*; $a \# b$ says that a occurs in a trace before b (the necessity is removed); $a \hat{\#} b$ says that if a occurs in a trace then b occurs as the next element of that trace (when classes other than those of a and b are ignored).

We also use an informal imperative pseudocode to describe behaviour where necessary.

3. Case Study — Warehouse Ordering System

A software house has developed an architecture to facilitate the development of warehouse ordering systems. The architecture includes a 'One At A Time (OAAT)' service, through which events arriving in quick succession can be managed, and a 'First Come First Served' service, which provides an event queue. For succinctness of reference within our case study, we specify the behaviours of the OAAT and FCFS services in our event model as follows.

For event classes A and B with event instances a and b, respectively:

$$\text{OAAT(A,B):} \quad a \rightsquigarrow b \Rightarrow a \hat{\#} b$$

Described in this way, OAAT prevents other events being interposed between a and b in any trace.

For event classes A and B with event instances a_1, a_2 and b_1, b_2, respectively:

$$\text{FCFS(A,B):} \quad (a_1 \rightsquigarrow b_1 \wedge a_2 \rightsquigarrow b_2 \wedge a_1 \# a_2) \Rightarrow b_1 \# b_2$$

Described in this way, FCFS preserves the ordering of related pairs of events with reference to the arrival of the first event of the pair.

OAAT and FCFS are defined as generic architectural services, i.e. they are parametrised with event classes. This feature permits architectural service instantiation within any application based on the services. In particular, we will make use of it in this paper in Section 4.

3.1. Requirements statement

The software house has received the following requirements statement for a warehouse ordering system:

The warehouse contains initial quantities of a range of products. (For simplicity, we assume that products are never replenished.)

Customers place orders by specifying an order number, a product, and a quantity. Orders are processed by passing requests to the warehouse. If the warehouse can satisfy an order from stock it reserves the specified quantity of stock and the allocation is notified. Otherwise no allocation or reservation is made.

No customer should be left waiting for an order indefinitely, nor should 'queue jumping' be allowed.

Warehouse staff behaviour Warehouse staff are responsible for reserving stock. Their behaviour, although not entirely appropriate (they are human!), can be described as follows:

```
while true {
  boolean satisfiable := false;
  receive Request(orderNo, prod, qty);
  satisfiable := (qty <=
    (prod.initialQty - prod.reservedQty));
  do the admin;
  if (satisfiable) then {
    prod.reservedQty :=
        (prod.reservedQty + qty);
    send Alloc(orderNo,prod,qty,true)
  }
  else
    send Alloc(orderNo,prod,qty,false);
}
```

where:

- `Request(orderNo, prod, qty)` is an order with number `orderNo` for `qty` amount of product `prod`.

- `Alloc(orderNo, prod, qty, success)` indicates the satisfaction (`success` = `true`) or otherwise (`success` = `false`) of the order. In the case of a satisfiable order, the reserved level of the product is increased accordingly. Otherwise no change is made.

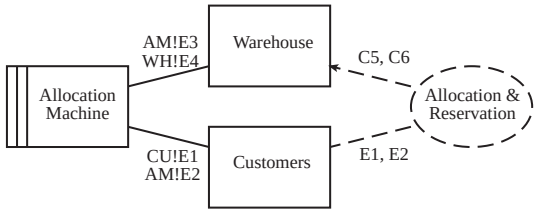
The presence of the unspecified administration activity reminds us that the test and possible assignment of `prod.reservedQty` should be executed atomically. Otherwise, interleaving of accesses by two staff members to the same product might give the wrong result. Unfortunately, the warehouse staff are not amenable to such disciplines as mutual exclusion.

3.2. A classical problem frame approach

The firm's developers wish to investigate the requirements for this problem using the Problem Frame approach. To do this, from the requirements statement the firm's developers decompose the problem into independent subproblems. These are:

- *satisfy the customers' orders if possible*: a commanded behaviour problem; the machine (i.e. the ordering system) conveys the order to the warehouse and notifies the customer whether the order can be satisfied.
- *prevent queue jumping*: a commanded behaviour problem; the machine ensures that orders are processed First Come First Served.
- *enforce mutual exclusion*: a commanded behaviour problem; the machine ensures that orders are dealt with by the warehouse One At A Time.

Satisfying customer orders The commanded behaviour frame for the first subproblem is shown in Figure 3.



CU!E1: Order(O#, Prod, Qty)
AM!E2:Reply(O#, Prod, Success)
AM!E3:Rqst(O#, Prod, Qty)
WH!E4: Alloc(O#, Prod, Qty, Success)
WH!C5: InitQ(Prod, Qty)
WH!C6: Resvd(Prod, Qty)

Figure 3. A commanded behaviour frame for Allocation & Reservation

The developers have identified three requirements associated with the Allocation & Reservation requirement. The first (R1a) states that each order will eventually receive a reply:

$$\text{R1a:}\quad \mathit{Order}(o,p,q) \rightsquigarrow \mathit{Reply}(o,p,\mathit{success})$$

where *success* is either *true* or *false*.

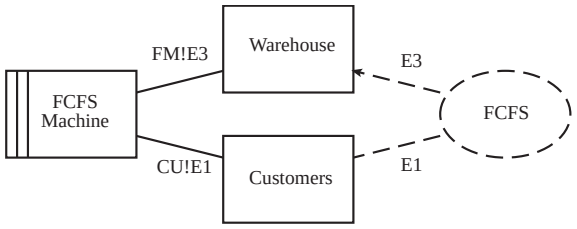
The second (R1b) states that a satisfiable order will result in a reply and a reservation:

$$\begin{aligned}\text{R1b:}\quad & \mathit{Order}(o,p,q) \wedge [\mathit{InitQ}(p,q_i) \wedge \mathit{Resvd}(p,q_j) \\ & \wedge (q_i - q_j) \geq q] \\ & \rightsquigarrow \mathit{Reply}(o,p,\mathit{true}) \wedge [\mathit{Resvd}(p,q_j+q)]\end{aligned}$$

The third (R1c) states that an unsatisfiable order will result in a reply, but no reservation:

$$\begin{aligned}\text{R1c:}\quad & \mathit{Order}(o,p,q) \wedge [\mathit{InitQ}(p,qi) \wedge \mathit{Resvd}(p,q_j) \\ & \wedge (q_i - q_j) < q] \\ & \rightsquigarrow \mathit{Reply}(o,p,\mathit{false}) \wedge [\mathit{Resvd}(p,q_j)]\end{aligned}$$

Preventing Queue Jumping The commanded behaviour frame for the second subproblem is shown in Figure 4.



CU!E1: Order(O#, Prod, Qty)
FM!E3: Rqst(O#, Prod, Qty)

Figure 4. A commanded behaviour frame for First Come First Served

The developers identify a single requirement (R2) associated with First Come First Served, that requests should be dealt with in the order that they are received by the machine:

$$\begin{aligned}\text{R2:}\quad & \mathit{Order}(o_1,p_1,q_1) \rightsquigarrow \mathit{Rqst}(o_1,p_1,q_1) \\ & \wedge \mathit{Order}(o_2,p_2,q_2) \rightsquigarrow \mathit{Rqst}(o_2,p_2,q_2) \\ & \wedge \mathit{Order}(o_1,p_1,q_1) \# \mathit{Order}(o_2,p_2,q_2) \\ & \Rightarrow \mathit{Rqst}(o_1,p_1,q_1) \# \mathit{Rqst}(o_2,p_2,q_2)\end{aligned}$$

Enforcing Mutual Exclusion The commanded behaviour frame for the third subproblem is shown in Figure 5.

The developers identify a single requirement (R3) associated with One At A Time, that only one request will be dealt with at a time:

$$\begin{aligned}\text{R3:}\quad & \mathit{Rqst}(o,p,q) \rightsquigarrow \mathit{Alloc}(o,p,q,\mathit{success}) \\ & \Rightarrow \mathit{Rqst}(o,p,q) \hat{\#} \mathit{Alloc}(o,p,q,\mathit{success})\end{aligned}$$

where *success* is either *true* or *false*.

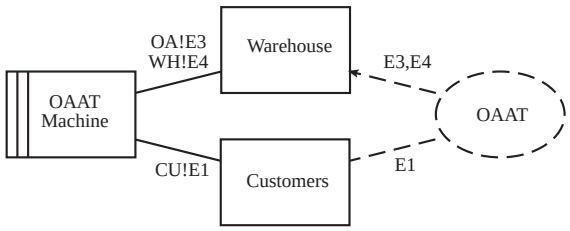


Figure 5. A commanded behaviour frame for One At A Time

At this point, the developers produce a machine specification for each subframe, build correctness arguments for the satisfaction of each subproblem's frame concern, and then recompose them in a way that satisfies the combined requirements to produce a correct system. For brevity, we do not reproduce these development steps here. It is clear that the development can be carried through to a machine specification for the entire problem.

The reader will note that, up to naming of events, R2 and R3 contain expressions that are similar to the specification of the architecturally provided services OAAT and FCFS given at the beginning of Section 3. In effect, this development process requires the bottom-up (re-)development of OAAT- and FCFS-like services, even though these are available within the firm's developed architecture. This redevelopment is necessitated by the fact that there is no mechanism in the classical problem frames approach to take advantage of available architectural support. To illustrate this further we consider the alternative analysis path based on given domains.

A classical approach based on given domains The alternative path available to the developers is to admit the architectural services as predetermined problem domain objects, i.e. given domains, and to construct a problem frame based thereon. The resulting problem frame is shown in Figure 6. Again the developers develop a machine specification, and build correctness arguments for the satisfaction of the frame concern, giving a correct system. Again, for brevity, we do not reproduce these development steps. But, again, the development can clearly be carried through successfully to a machine specification.

Although this development path would support our argument that architecturally provided services can be seen as problem domain objects, it has the following drawbacks:

- even though in this case the added complexity is small, in the general case there could be a combinatorial explosion of boxes and connectors between the added given domains;
- the problem frame moves from being within a recognised class: the domain objects introduced do not fit within the general commanded behaviour frame, but have been introduced *ad hoc*; whether the resulting frame remains within the expertise of a domain expert is moot; and
- because the services are outside the machine domain, their use therein is arguable.

We therefore discount this approach, and look instead to extend the problem frame approach in a more systematic way.

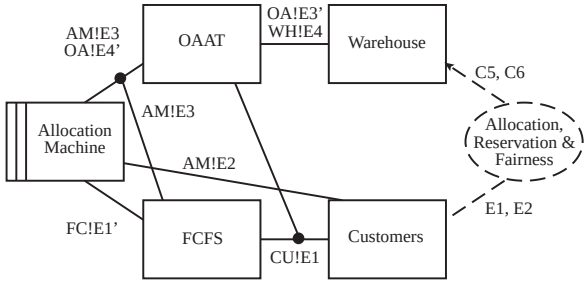
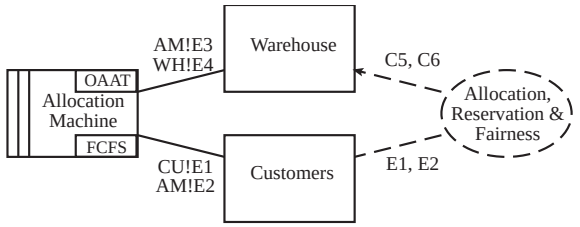


Figure 6. OAAT and FCFS services as given domains

4. Extending Problem Frames

Let us assume that we want to make use of the architecturally provided OAAT and FCFS services with properties defined by their service descriptions given in Section 3. As previously discussed, there is no gain in considering them for sub-problem frame analysis or as given domains; we therefore need to look elsewhere to employ them. The approach we suggest is to use them to annotate the machine domain. This simultaneously reminds us of their presence within the problem domain — they appear in the problem frame — and allows them to appear *within* the machine, and hence the solution, domain.

As an example of an annotated machine domain, Figure 7 shows the warehouse ordering system machine domain annotated with services: OAAT and FCFS, parametrised with the appropriate event classes.



CU!E1: Order(O#, Prod, Qty)
AM!E2:Reply(O#, Prod, Success)
AM!E3:Rqst(O#, Prod, Qty)
WH!E4: Alloc(O#, Prod, Qty, Success)
WH!C5: InitQ(Prod, Qty)
WH!C6: Resvd(Prod, Qty)

Services:
OAAT(E3,E4)
FCFS(E1,E3)

Figure 7. A frame with annotated machine domain

This frame shares the same domain descriptions, requirements and frame concern as that of Figure 3. However, through the annotation we are reminded of the fact that the machine domain can rely on the OAAT and FCFS services provided by the architecture. We can factor this extra information into the machine domain specification and into the correctness argument that discharges the frame concern.

4.1. Machine Specification

Accepting this annotated machine domain, we can specify the machine as having the following simple behaviour. The machine should generate request events from order events and reply events from allocation events:

S1: $Order(o, p, q) \rightsquigarrow Rqst(o, p, q)$

S2: $Alloc(o, p, q, success) \rightsquigarrow Reply(o, p, success)$

4.2. Discharging the Frame Concern

To follow the development through we need to build a correctness argument that brings together the requirements, domain descriptions and machine specification to discharge the frame concern. However, with the annotated machine, we may also rely on the descriptions of the architectural services. In the following we outline informally the correctness arguments required to discharge the frame concern; although a more formal treatment is possible, for brevity we do not give it here. The discharge of the frame concern is shown in Figure 8; the numbers there relate to the steps in the following subsections. As the frame concern is the same for both correctness arguments, we give the figure only once.

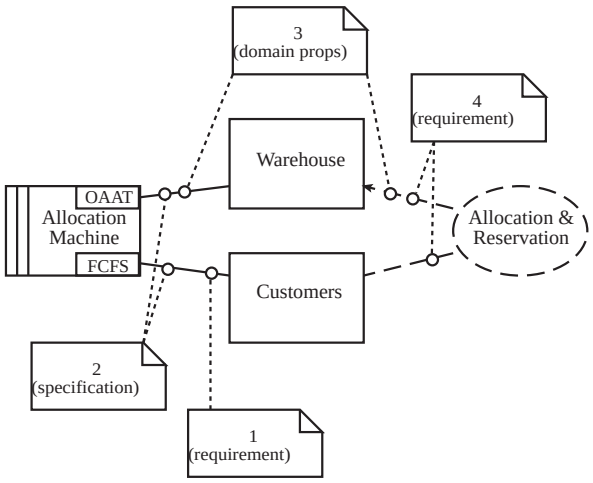


Figure 8. Discharging the frame concern

Service (R1a, R1b, and R1c)

1 When the customer issues an order (requirement)...

2a ... the machine generates a corresponding request to the warehouse (specification) ...

3 ... the warehouse processes the order with respect to current stock levels and issues a reply to the machine (domain properties) ...

2b ... the machine generates a corresponding response to the user (specification) ...

4 ... thus achieving the required results in every case.

Fairness (R2 and R3)

1 When customers issue two (or more) orders (requirement)...

2a ... the FCFS service sequences the orders by arrival time and sends them to the machine (specification) ...

2b ... the machine generates corresponding requests to the OAAT service (specification) ...

2c ... the OAAT service allows one request at a time through to the warehouse ...

3 ... the warehouse processes each order with respect to current stock levels and issues replies to the OAAT service (domain properties) ...

2d ... the machine generates corresponding responses to the customers (specification) ...

4 ... thus achieving the required results.

5. 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 [13] upon which the SCR requirements specification approach and associated tools were developed [12]. SCR partitions the world into monitored and controlled variables, and shares much in common with the problem frames approach. Its main focus, however, is on event-based (control) systems, and its tabular notation has not been used to incorporate architectural considerations.

A number of goal-based requirements approaches, most notably KAOS [17] and the NFR framework [5], have proposed the explicit use of the notion of 'goals' to structure requirements and, consequently, the problem domain. KAOS in particular, allows domain properties, assumptions and constraints to be represented as part of its goal-based specifications. More recent work by Letier and van Lamsweerde addresses the identification of agents (e.g., components), and their assignment to goals [18, 17]. The KAOS approach remains essentially goal-driven and it is not clear how existing architectural artifacts can influence the problem structuring process. Similarly, the NFR framework uses high-level goals to initiate a process of identification of associated design components.

A very common structuring approach to requirements specifications is still the use of pre-defined templates or standards that prescribe how requirements specification documents should be partitioned [23, 16]. 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 much more difficult to enforce clear separation between descriptions of the problem domain and those of the solution domain.

The structuring of the solution domain is much more commonly and thoroughly researched. It includes structuring at the implementation level through a variety of programming constructs; but, more relevant to the context of this paper, structuring at the design level is through design abstractions such as functions [22], objects [19], aspects [6], and so on. Much of the recent research on software architecture has focused on the use of components and connectors as structuring concepts [24]. But, with a few notable exceptions [2], very little consideration has been given to relating such software architectures to requirements in the problem domain.

The relationship between requirements and architectures has received increased attention recently [4]. 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. It is not clear however that the overhead of an additional description is practical.

Recent work on software product lines and system families has also examined the relationship between architectures and requirements. The work has focused on identifying core requirements (identified perhaps through a process of requirement prioritisation) and linking them to core architectures (identified perhaps by examining the stability of various architectural attributes over time) [25]. This work has not explicitly addressed the issue of iterative development in this context, focusing instead on domain abstractions and reuse.

Wile [26] 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. The focus of this work is runtime monitoring, not more traditional development activity.

Grunbacher et al [9] explore the relationships between software requirements and architectures, and propose an approach to reconciling mismatches between requirements terminology and concepts with those of architectures. This approach may provide useful capability for capturing and maintaining complex relationships between different artifacts of requirements and architecture; it has currently been used only in a very particular context.

Finally, the industrial strength method REVEAL [11], based on a clear separation between the world and the machine [10], provides a practical approach to developing systems in the manner that we proposed in this paper. REVEAL, however, assumes a more traditional development process that is less fine-grained than that suggested by our Twin Peaks model, and therefore potentially less amenable to iterative development in which design informs requirements as much as requirements inform design.

6. Discussion and Conclusions

In this paper we have described an extension to problem frames that is intended to facilitate their synergistic combination with software architectures. In essence, we have extended the current model of machine domain in problem frames to mean 'architectural engine', i.e. a computer in which architectural structures, services and artifacts can be relied on by software executing in the machine. We have applied the extension to a small case study to show how development with extended machine domains may proceed.

The extension has methodological implications that require investigation. By allowing the properties of a particular machine domain to become an explicit factor in discharging the frame concern, we enable 'non-green field' development using the problem frames approach. We con-

jecture that this will reduce development times by allowing the developer to describe the problem domain more abstractly, closer to the 'business logic' that operates in the domain. Moreover, the admission of the extension leads to multiple intertwined development paths; architectural support will be the subject of design activities separate from those of a product based on it. Whether our extension allows full use of the power of this idea remains to be seen. To ensure it does, we must at least factor in some notion of inter-developmental stability in which the architectural support appears a fixed point in product development; currently, this appears to be catered for by the notion of a fixed architecture 'API', with deprecation mechanisms providing flexibility.

Finally, notationally, the extension is slight. Whether the annotation of machine domains is sufficient to capture all useful architectural support again remains to be seen. We intend to apply the ideas of this paper to larger case studies, with various flavours of architectural support, to explore the demands that each places on the notation.

7. Acknowledgments

We acknowledge the support of our colleagues in the Department of Computing, the Open University.

References

[1] L. Barroca, J.G. Hall, P.A.V. Hall (eds), Software Architectures — Advances and Applications, Springer, 2000.

[2] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison Wesley, 1998.

[3] M. Brandozzi, D.E. Perry, Transforming Goal Oriented Requirement Specifications into Architectural Prescriptions. In [4].

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

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

[6] T. Elrad, R.E. Filman, A. Bader (eds), Special issue on Aspect Oriented programming, Communications of the ACM, 44(10), 2001.

[7] M.E. Fayad, D.C. Schmidt, R.E. Johnson, Building Application Frameworks, Wiley Computer Publishing, 1999.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley, 1995.

[9] 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.

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

[11] J. Hammond, R. Rawlings, A. Hall, 'Will it Work", Proceedings of the 5th International Symposium on Requirements Engineering (RE'01), pp.102-109, IEEE CS Press, 2001.

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

[19] B. Meyer, Object Oriented Software Construction. (2^{nd} edition) Prentice-Hall, 1997.

[20] J. Moffett, J.G. Hall, A. Coombes, J.A. McDermid, "A Model for a Causal Logic for Requirements Engineering". Journal of Requirements Engineering, 1(1), 1996.

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

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

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

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

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

[26] 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.