

Architectural Design Support for Composition & Superimposition

Jilles van Gorp, Rein Smedinga, Jan Bosch
Department of Mathematics and Computing Science
University of Groningen
PO Box 800, 9700 AV The Netherlands
[jilles|rein|jan.bosch]@cs.rug.nl
<http://www.cs.rug.nl/Research/SE>

***Abstract.** The ever growing size and complexity of software systems is making it increasingly harder to built systems that both meet current and future requirements. During architecture design, a lot of important design decisions are taken. In this paper, we present an architecture design notation based on UML's activity diagrams. The notation allows for the specification of architecture fragments and supports composition of these fragments as well as superimposition of the fragments on each other. This notation allows us to make various compositions of architecture fragments (reflecting design decision alternatives) to adapt the architecture to new requirements. We have found that our notation is very suitable for modelling separate concerns at the architectural level and for creating.*

1 Introduction

The ever-growing size and complexity of software systems is making it increasingly harder to built systems that both meet current and future requirements. In [Van Gorp & Bosch 2001], we identified that development of systems consists, to a large extent, of taking design decisions. Typically these design decisions accumulate and consequently it is often hard to discard decisions taken early in the development due to the consequences such an action would have on the subsequent design decisions. Eventually, new requirements will invalidate some of these decisions. The process of incorporating new requirements properly can be expensive. Consequently, a less than optimal solution is often preferred to preserve the architecture that resulted from the earlier design decisions. The use of such quick-fixes erodes the architecture and adds to the problem rather than solving it.

Currently there is ongoing research that focuses on separation of concerns. E.g. Aspect Oriented Programming (AOP)[Kiczalez et al. 1997], Subject Oriented Programming (SOP)[Harrison & Ossher 1993] and Multi Dimensional Separation of Concerns (MDSC)[Tarr, Ossher & Harrison 1999]. However, considering that the most important design decisions are those taken early in the development, these approaches share a flaw: they all operate on the implementation level and detailed design level only. In this paper we propose an architecture level design notation that is specifically designed for modelling concerns on an architectural level while preserving information about the design decisions taken during the architecture design phase.

1.1 Problems

Lack of architectural separation of concerns. Many important design decisions are typically taken early in the development of a system. Especially during architecture design, many important decisions are taken. However, despite this, few architecture design techniques take separation of concerns into account. Such techniques do exist for the detailed design and implementation phases (e.g. [Kiczalez et al. 1997][Harrison & Ossher 1993][Tarr, Ossher & Harrison 1999]). Methods and techniques for achieving separation of concerns at the architecture level are lacking, though.

Poor support for withdrawing design decisions. A second problem is that many architecture design methods work in an iterative fashion and accumulate design solutions as the architecture evolves. Because of this, each new design solution added to the architecture becomes dependent on all of the previous decisions. However, some decisions do not really affect all of the system and could be imposed on an early version without affecting later versions.

If, for instance, we have a set of design decisions, D1, D2 and D3, that are applied to an architecture A, the normal course of development would be to first change the architecture to incorporate D1 then D2, and then D3. However it would be difficult to first do D2 and then D3 and then apply D1 to the original architecture (i.e. without D2 and D3 applied). With stepwise refinement, D1 has to be applied to the full architecture because the only architecture available is that with D2 and D3 already applied. The original architecture is lost in the process. This causes problems when there exists a variant of D1: D1' that needs to be inserted instead of D1.

Imposing new design decisions on an existing architecture. Often, design decisions need to be taken that have an effect on design decisions already taken. A good example of this is imposing a caching algorithm on an architecture to improve efficiency of the communication. After a building a first version of the architecture without caching, testing might show that communication needs to be improved. Typically adding caching can be added to a system in a transparent fashion. However expressing this on an architectural level may be cumbersome since the component structure is changed. Ideally, we would like to model the architecture without caching and then specify how caching can be added to this architecture rather than re-specifying the architecture to include caching. In addition, when taking future design decisions, we do not want to add dependencies to the caching design decision unless this is required or cannot be avoided (i.e. further design decisions are dependent on the architecture without caching).

1.2 Example

As an example, consider the fire alarm system we use as the domain for our example in Section 3. This example is based on an earlier case study by [Bosch & Molin] and [Molin & Ohlsson]. In the original version of the fire alarm system, a number of design decisions are taken to optimise behaviour the architecture for real time and performance requirements. As a result an application level scheduler is introduced and a blackboard architectural style is used to streamline communication between the software representations of sensors and actuators.

In the final architecture of this system, several concerns are mixed:

- **Domain Behaviour.** The domain behaviour in a fire alarm system consists of actuators that are triggered by measured deviations in a set of associated sensors.
- **Concurrency.** For each component, a design choice has been made as to whether it is actively scheduled or passively scheduled (i.e. only active when called by another component). This results in the introduction of a tick method in the components that is called by the scheduler. Components that provide a tick method can be actively scheduled by the scheduler. Components that do not provide such a method are only executed when they are called by other components. This is called passive scheduling.
- **Communication.** The architecture uses a blackboard architecture. This means that the communication between sensors and actuators is intercepted by a blackboard component, which minimizes redundant communication between the components.

In this paper we explore an alternative to the blackboard architecture: caching. The decision whether to use a blackboard or a cache also has consequences for scheduling. In a blackboard architecture, sensors can be actively scheduled (i.e. they post their values on the blackboard at regular intervals). In a caching architecture, however, the cache is responsible for requesting sensor values. This means that in the latter situation, sensors are passively scheduled.

This small example shows the essence of the problems we are dealing with. The concerns concurrency and communication are mixed. Consequently, design decisions affecting one of these concerns, also have an effect on the other concerns. In addition, these design decisions are not associative and consequently, it is hard to replace the blackboard with a cache. The reason for this is that doing so requires a different scheduling strategy (passive scheduling for sensors instead of active scheduling).

When embedding a fire alarm in a larger system (e.g. a building management application that integrates other systems such as climate control systems, for instance), a relevant design choice is which version of the fire alarm is to be used. This choice is influenced by the requirements of the larger system. In our example, the choice would be whether to use the cache version or the blackboard version. Ideally, it would not matter which version is used since both implement the same functionality. If this is the case, we would say the fire alarm architectures are substitutable (i.e. one can transparently replace the other). However, in practice, the mixing of concerns in the implementation makes this difficult.

1.3 Solutions

We address the identified issues by introducing a UML based notation for defining and composing architecture fragments. Since the composition of fragments is made explicit, to a large extent, it does not suffer from the problems outlined above. Of course some mixing of concerns is necessary to express the functionality of the system. However, this mixing of concerns is limited to constraints on the composition of fragments. The remainder of this paper consists of a discussion of the notation and the discussion of an example case where this notation is used as well as an analysis.

1.4 Related Work

Architecture. The notion of software architecture was already identified in the late sixties. However, it wasn't until the nineties before architecture design gained the status it has today. Publications such as, for instance, [Shaw & Garlan 1996] and [Bass et al. 1998] that discuss definitions, methods and best practices have contributed to a growing awareness of the importance of an explicit software architecture. The IEEE currently provides the following definition: "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution." [IEEE1471 2000].

More in line with our view on architecture is the following definition: "Software architecture is a set of concepts and design decisions about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains." [Jazayeri, Ran & Van Der Linden 2000]. This definition supports our notion that it is possible to compose an architecture from such basic components as domain components and architecture fragments.

Patterns. At the same time the notion of an architecture was developed, the notion of a design pattern also became important [Buschmann et al. 1996][Gamma et al. 1995]. Design patterns and architectural patterns isolate particular design solutions that can be applied during detailed or architectural design. The resulting pattern is a generic solution to a recurring problem. The notation discussed in our paper could be used to model architecture patterns. The example we discuss in Section 3, for instance, uses the blackboard architectural style discussed in [Buschmann et al. 1996].

Architecture Erosion. A motivation for writing this paper was the idea that due to requirement changes, architectures tend to erode over time. In [Van Gorp & Bosch 2001], we presented a case study that demonstrates how architecture erosion works. One of the conclusions in this paper is that due to requirement changes, particular design decisions may need to be reconsidered. Since the architecture is the composition of all design decisions [Jazayeri, Ran & Van Der Linden 2000], any changes in these decisions will affect the architecture. PSEUDO CODEThe notion of architecture erosion was first identified in [Perry & Wolf]. In [Jaktman, Leaney & Liu 1999], a set of characteristics of architecture erosion is presented.

Separation of Concerns. An approach to prevent architecture erosion is to pursue separation of concerns. By separating concerns, the effect of changes can be isolated. E.g. by separating the concern synchronization from the rest of the system implementation, changes in the synchronization code will not affect the rest of the system. Examples of approaches that try to improve separation of concerns are Aspect Oriented Programming [Kiczalez et al. 1997], Subject Oriented Programming [Harrison & Ossher 1993] and Multi Dimensional Separation of Concerns [Tarr, Ossher & Harrison 1999]. A problem with these approaches is that they focus on the implementation level whereas important design decisions are taken prior to the implementation. Our approach addresses this issue by providing an architectural level notation that allows for separation of concern.

Composition. Our composition technique bears some resemblance to the notion of super-imposition discussed by one of the co-authors [Bosch 1999]. In this approach, program fragments are imposed on an existing program structure. The main advantage of superimposition compared to existing techniques such as inheritance or wrapping is that the change is transparent to users of the original program structure. However, whereas the approach [Bosch 1999] suggests is an implementation/detailed design technique, our notation is intended for use on the architectural level.

Scripting. In [Ousterhout 1998], scripting languages are characterized as a simple means to glue together objects and components. Our notation could be viewed as an architectural scripting language. Our notation, and especially the associated pseudo code notation, is not concerned with such details as Classes, Types and Properties. It describes components purely in terms of the functionality they provide. This simplifies the composition and the graphical notation makes it very readable. An explicit goal of our notation is to facilitate describing architectures while reusing existing architecture fragments. So in a way it is very similar to a script-

ing language. It also shares the same benefits. Since distracting details like types and data format are omitted, the notation is very flexible.

Notations. Our notation is based on UML's Activity Diagrams [OMG UML]. The reason we use this notation instead of, for instance, ACME [Garlan et al. 1997], Rapide [Luckman 1996] or WRIGHT [Allen 1997], is two-fold. The first reason is that we need a more fine-grained notation in order to do compositions of architecture fragments. Notations like ACME apply a boxes and arrows approach to modelling architectures. However, the semantics of individual components are determined by how the box works internally rather than how it cooperates with other components. A second reason is that UML's activity diagrams can be seen as a means of identifying domain components and complementary to Use Case diagrams typically used in the early phases of development [Fowler & Scott 1998].

Rapide is an ADL that allows one to specify systems in terms of partially ordered sets of events and can simulate architecture designs; ACME is a common interface format for architecture design tools. Unlike most ADLs, our notation also describes the control flow inside the components (rather than just the externally visible behaviour) and allows for composition of different components, or fragments as we prefer to call them. Therefore, our notation uses a white box approach (we describe internal functionality of components as well as communication between components) while the ADL's uses a blackbox approach (only the communications between components are taken into consideration). With our white box approach [Roberts & Johnson 1998] we can describe superimposition [Bosch 1999]. WRIGHT is close to our approach because it is based on CSP [Hoare 1985]. In our approach a more subjective notation is used and it is based on trace theory [Snepscheut 1985] that has less basic principles but is sufficiently expressive, nevertheless.

1.5 Remainder of the paper

In Section 2 we introduce our approach. Section 3 discusses an extensive example where this approach is used. In Section 4 we provide an analysis of the use of our approach on the case presented in Section 3. And, finally, we conclude our paper in Section 5.

2 Notation

In [Van Gorp, Bosch & Svahnberg 2001], we outline the development process as a process of constraining variability. The process starts with collecting and interpreting requirements, creating an architecture design, a detailed design, an implementation, a compiled system, a linked system and a running system. At each phase decisions are taken about the design of the system. For instance, during requirements analysis, decisions are taken about which features to include and which features to exclude from the system. Each decision removes variability because a design decision picks one alternative out of several others, therefore limiting the amount of variability.

In this paper we focus on the architecture design phase. While this phase can be revisited later in the development (which is not uncommon in iterative development methods such as extreme programming [Beck 1999]), most of the architecture design is created very early in the development process. The reason for this is that as the development process progresses, the legacy of the later phases (e.g. detailed design and implementation) starts to become an obstacle for radical architectural changes. Radical architectural changes have a strong effect on this legacy and are therefore not very cost effective. Consequently, the architecture design is something that is typically fixed early in the development process even though many requirements are still unknown then.

Based on this assumption, it is safe to say that the architecture design process gets most of its input from the requirements analysis and previous experience with building similar systems. The latter knowledge is available as architectural styles [Buschmann et al. 1996], design patterns [Gamma et al. 1995] and the developer's personal experience. This knowledge is used to take suitable architecture design decisions to create an architecture that can be used to build a system that can both meet the available requirements and that is flexible enough to meet expected/likely future requirements.

An architecture design decision may have one or more of the following effects on an architecture:

- It can introduce new design rules.
- It can impose constraints on the existing architecture.
- It can introduce new structural elements to the architecture.
- It can remove structural elements from the architecture.
- It can superimpose new behaviour on some or all elements of the existing architectural structure.

The notation we introduce in this paper primarily supports the latter three types of design decisions and can easily be extended to provide support for first two types. Our approach revolves around so-called activities. Activities represent behaviour the system exposes. Activities take place in a certain order and may also execute asynchronously. The reason activities take a central place in our approach is that when the architecture of a system is conceived, the only knowledge available about the future system are the requirements that mostly define what functionality the system is going to expose (e.g. use case diagrams). In addition, quality requirements can be expressed as additional activities that are superimposed on the functional architecture. Consequently, specifying the initial architecture is a matter of breaking down the functionality and organizing it into a suitable structure. This strategy of starting with a functional design is also a part of the architecture design outlined in our book on Software Product Lines [Bosch 2000]. Consequently, our notation can be used in conjunction with this method and is in fact a very natural way of using it. In the example we present later on, we start with a functional design. From there on we explore several design alternatives to address the quality requirements. By means of composition and superimposition these design steps are kept transparent with respect to the original design as much as possible.

The notation we use is based on UML's activity diagrams. A UML Activity diagram can be used to model the functionality of the system at a very high level of abstraction (which is required during architecture). By grouping activities in so-called swimlanes, architectural or domain components can be identified and specified in terms of their functionality. An additional property of activity diagrams we discovered and exploit in this paper is that it is easy to specify compositions of fragments of activity diagrams. In addition it is possible to superimpose such fragments on an existing diagram.

2.1 Formal Notation

In order to specify the composition or fragments, we use a formal notation that is equivalent to the graphical notation. Because one of the aspects in UML's activity diagrams is concurrency and synchronization, we use basic formal theories to describe the behaviour of communicating components. They date back to [Hoare 1985] and [Milner 1993]. The formal notation, used in this paper, first appeared in the *trace theory* approach of [Snepscheut 1985]. In this notation, a trace structure consists of an alphabet (set of activities) and a trace set (all sequences of activities that are allowed in the structure; including their prefixes). We adopt the weaving composition function of trace structures. In addition to this algebra, we also provide a pseudo code notation for enhanced readability. We use the formal notation only to precisely define the semantics of our graphic and pseudo code notations.

In this paper we use a, b, c for single activities and P, Q, R for sequences of activities. The operator $a \bullet b$ denotes concatenation: activity b follows after a . The operator $P \leftrightarrow Q$ denotes choice: either P or Q will be the next sequence of activities. Concurrency is denoted by $P \parallel Q$ and means that P and Q can run in parallel. Com-

Table 1: Notations


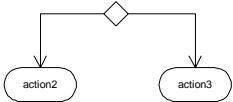
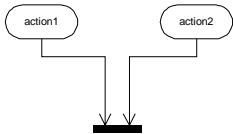
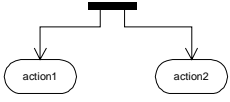
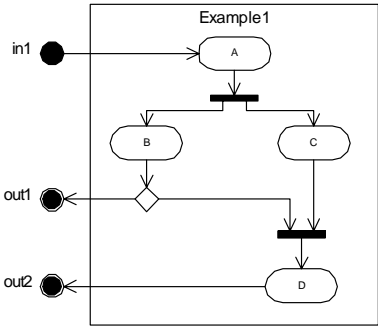
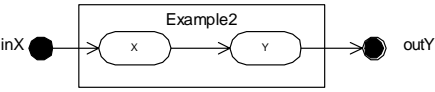
Activity diagram	Algebra notation	Programming notation
	$action1 \bullet action2$	<code>action1 ; action2</code>
	$action1 \leftrightarrow action2$	<code>if B then action1 else action2 fi</code>
	$(action1 \bullet s) \parallel (action2 \bullet s)$	<code>fork action1 ; s action2 ; s end</code>

Table 1: Notations

Activity diagram	Algebra notation	Programming notation
	$action1 \parallel action2$	<pre> fork action1 action2 end </pre>
	<p>The swimlane Example can be described by:</p> $ \begin{aligned} &Example1(in1;out1,out2) \\ &= \\ &in1 \bullet A \bullet \\ &((B \bullet (out1 \leftrightarrow s)) \parallel (C \bullet s)) \bullet \\ &D \bullet out2 \end{aligned} $	<pre> fragment Example1 (in in1; out out1, out2) begin in1 ; A ; fork B ; if condition then out1 else s fi C ; s end ; D ; out2 end </pre>
	$ \begin{aligned} &Example2(inX;outY) \\ &= \\ &inX \bullet X \bullet Y \bullet outY \end{aligned} $	<pre> fragment Example2 (in inX; out outY) begin inX ; X ; Y ; outY end </pre>

mon activities in P and Q are used for synchronization. For example $(a \bullet b \bullet c) \parallel (b \bullet d)$ uses b as synchronization. P and Q can only proceed with such a common activity if both P and Q are ready to do so at the same time. The resulting order of activities in our small example is $a \bullet b \bullet (c \parallel d)$. This composition function is called weaving in trace theory. When the common b is an *internal activity* for synchronization purposes only, we use the composition function blending. With blending, the internal activity is left out the resulting behaviour. In the above example the blending results in $a \bullet (c \parallel d)$ (i.e. first a and then c and d in parallel). We use blending to formally describe the internal synchronization (see Example1 in Table 1) and for composition of fragments (see Section 2.2).

UML Activity Diagrams use so-called swimlanes to group related activities. In our notation, swimlanes can be formally described by using the above operators together with internal activities defining the in-going and outgoing triggers of the swimlane. Such a representation of a swimlane is called a *fragment* (see table 2 for an example). The interface of the fragment is given in the parameter list where **in** is used for the required interface and **out** for the provided interface. **in**-activities are the in-going triggers for the fragment (correspond to the initial states of the UML activity diagrams) while **out**-activities are the outgoing triggers (the final states). Composition of fragments means mapping required to provided interfaces, as is explained in the next section.

The programming notation uses boldface keywords like **fragment**, **begin**, **end**, etc. Curly brackets { and } are used for constraints. We use two kinds of constraints in our notation: *composite constraints* (for composition of fragments, see the next subsection) and *natural language constraints* (for all other constraints) The latter type of constraint may be useful in the later stages of development, e.g. in detailed design.

2.2 Composition & Superimposition

Composition of a number of fragments can be done using the \parallel -operator together with the synchronization mechanism. Common activities are used as internal activities for synchronization (in [Snepscheut 1985] this is called blending (weave both behaviours by synchronizing on the common events and omit the common events in the result)).

As an example consider the second system Example2 and the composition with Example1 by connecting

outY with in1 (we map an outgoing trigger with an in-going trigger). outY (or in1) is used as the common internal activity and the blending of both behaviours is represented by the corresponding figure in table 2. By connecting outY with in1 both these internal activities disappear in the composition, but all other internal activities remain and can be used for further compositions. The resulting composition is again a fragment in the sense that it can be used for further compositions as well.

The connection operator is both symmetric and associative i.e. $a \parallel b = b \parallel a$ and $(a \parallel (b \parallel c)) = ((a \parallel b) \parallel c)$ In

Table 2: Composition of fragments

Activity diagram	Algebra notation	Program notation
	<p><i>Composition</i> $=$ $Example2(inX, s)$ \parallel $Example1(s, out1, out2)$ (where $inX = outY = s$), The result is: $nX \bullet X \bullet Y \bullet A \bullet$ $((B \bullet (out1 \leftrightarrow s)) \parallel (C \bullet s)) \bullet$ $\bullet out2$</p>	<pre> fragment Composition (in inX; out out1, out2) begin fork example (in1, out1, out2) example2 (inX, outY) with in1 = outY end end </pre>

[Snepscheut 1985] it is proven that the corresponding blending-operation is both symmetric and associative. It should be noted, though, that blending is only associative as long as internal activities are common to at most two of the involved fragments. This rule applies to our notation because we explicitly declare internal activities as equal, pairwise for each \parallel operation, because we assume all possible internal activities to be unique.

The associative property makes it possible to compose fragments in any particular order. Only the activities denoted by **in** and **out** in the parameter list of the fragment are used for the composition. For each composition, a number of **in**-activities are coupled to a **out**-activity in the other fragment, declared equal and used as a common (or internal) activity in the composition. The other activities in the body of the fragment are used nor disrupted in any sense (except for superimposition). Also the description of the fragment is not changed. Thus, fragment descriptions remain unchanged under composition as long as the coupling of **in** and **out**-activities is done pairwise (in Section 3.5 we will show an example where we do not connect these activities pairwise).

A second form of composition that is supported in our notation is superimposition [Bosch 1999]. Superimposition allows for composition of a fragment with activities inside a fragment (i.e. the fragments internal behaviour is enhanced). In order to express this in our notation, all arrows in the UML-swimlanes are considered to be anonymous internal activities. Formally, we assume that instead of $a \bullet b$, the concatenation consists of a finite and suitable number of internal activities, e.g. $a \bullet e_1 \bullet e_2 \bullet \dots \bullet e_n \bullet b$, where each e_i is an anonymous activity. In our program notation these anonymous activities are present at each semicolon. We can indicate e_1 by writing $a.after$ and e_n by writing $b.before$. Both these internal activities can then be used as if they were listed in the parameter list with **in** or **out**. The keywords **before** and **after** are also used in the pseudo code notation. In case the internal activity goes just before, or just after a decision-node, we will use the condition X together with the **if** to denote the internal activity, for example $ifX.before$ denotes an anonymous activity just before the decision-node and $ifXtrue.after$ an anonymous activity just after the decision-node following the true-arrow. Many reflective OO languages (e.g. CLOS [Kiczalez et al. 1991]) use a similar mechanism.

In table 3 an example is given: Observable is superimposed on Example2, in order to apply the observer pattern [Gamma et al. 1995]. The resulting fragment gains an extra input and an extra output state. Note that we are still able to change Example2 independently of this composition, as long as in Example2 the provided and required interface is not changed.

Table 3: Superimposition of fragments

Activity diagram	Algebra notation	Program notation
	<p><i>Observable</i> $=$ $change \bullet notify \bullet$ $done \bullet proceed$</p> <p>The Composition is given by</p> <p>$inX \bullet X \bullet notify \bullet$ $proceed \bullet Y \bullet outY$</p>	<pre> fragment Observable (in change, done; out notify, proceed) begin change ; notify ; done ; proceed end fragment ObservableExample2 (in inX, done ; out outY, notify) begin fork Example2(inX, outY) Observable(change, done, notify, proceed) with X.after = change and Y.before = done end end </pre>

2.3 Interfaces

When composing fragments the internal description is not needed, except when using superimposition. Therefore we introduce *fragment interfaces*. A fragment interface is a fragment without internal activities. Fragment interfaces can be used in compositions instead of real fragments. The advantage of this is that different fragments ‘implementing’ the fragment interface can be substituted in that composition.

When associating a fragmentinterface with a concrete fragment, the fragment must have the same in- and out-parameters. The fragmentinterface only describes the outside of the corresponding fragment in the activity diagrams. The program notation for fragment interfaces is **fragmentinterface** *IName* (*parlist*). By convention, we add a prefix (I) to the name to distinguish it from ordinary fragments. To indicate that a fragment is a realization of one or more fragment interfaces, we use the following syntax: **fragment** *Name implements IName1, IName2, ...*

2.4 Deriving a detailed design

Our notation is intended for use on the architectural level. While our notation is UML based, we feel that it is necessary to elaborate on how to use the resulting composition as a starting point for detailed design. An important thing to realize is that there may be more than one possible detailed design for a given architecture design. When creating the detailed design additional design decisions are made.

The UML diagrams, typically used during detailed design, are class diagrams and collaboration diagrams. Because of this, we use these notations as detailed design notations. Since architecture level diagrams lack certain information present in a detailed design, we do not consider such things as implementation inheritance or class variables. Specifying such information really is part of the detailed design and since we are primarily concerned with the architectural level, such details are irrelevant for the time being.

Consequently, we use a subset of the constructs typically found in a class diagram. Rather than specifying classes, we specify interfaces. The detailed design phase then consists of specifying suitable implementation classes, defining an inheritance hierarchy for these classes and specifying additional information about the way these classes collaborate.

A straightforward method to derive a detailed design from a fragment composition is to interpret the fragments as UML-interfaces and the activities as method calls. The composition of the fragments then serves as information about collaboration and can be used to derive aggregation and containment relations between the fragment interfaces. Since we have an interface construct in our fragment definition language, we also have information about which methods should be public and which methods should be private (i.e. those activities without incoming arrows). It should be noted that the activities specified in a fragment interface are lost in the blend operation described earlier.

Furthermore, the information from the various compositions provides us with the information about how these UML interfaces relate to each other. Every time an outgoing activity is mapped to an incoming activity in another fragment, we are dealing with some form of delegation (either a method call or a return from a previous call). In the composition, the outgoing operation is mapped to an incoming operation, so, in a UML class diagram this results in a call to one of the public methods on an interface.

UML uses several types of relations, which can all be used to model this type of delegation. The simplest form is defining an association relation. An association relation says nothing more than that one end of the association is associated with the other end in some way. By specifying cardinalities, it can be expressed that, for instance, one end is associated with multiple entities on the other end. Information about these cardinalities may be present in the fragment definition in the form of constraints. Since the control flow is unidirectional in the fragment definition, it is probably also a good idea to use navigability on the associations (this makes the association uni-directional) to indicate this in a class diagram.

More advanced forms of delegation-like relations in UML include aggregation and composition relations. However, our fragment notation does not provide enough information to derive this type of relation. We consider making decisions regarding this type of relation to be important design decisions that are part of the detailed design. However, sometimes it is obvious that e.g. an aggregation relation is intended, so specifying such relations during derivation may be done if possible but in general the architecture design does not provide the necessary information to make such a decision.

Inevitably superimposition information is lost in the process since we do not have similar detailed design constructs available. It may be necessary to take additional design decisions such as splitting/merging interfaces and specifying additional methods. We have found that the distinction between an architecture design and a detailed design is a very grey area. In fact the derivation process outlined in this section could be considered to be part of either development phase.

Once a class diagram has been derived, additional object collaboration diagrams may be defined as well. Doing so is rather straightforward and boils down to following the arrows in the activity diagram notation we use. In the next section, we will discuss examples based on our notation. At the end of that section, we will also discuss a derived detailed design based on one of the examples.

3 Examples

In the introduction we already mentioned the fire alarm case briefly. To illustrate our technique, we applied it to this case. As mentioned in the introduction, the case is based on a case study we performed a few years ago [Bosch & Molin][Molin & Ohlsson].

3.1 The fire alarm system

The subject of the case is the creation of an architecture for a fire alarm system. In [Bosch & Molin][Molin & Ohlsson] we describe an architecture for this domain that was developed together with Telealarm AB, a manufacturer of such alarm systems. In this paper we will use the requirements that were associated with this architecture and use them to create various architectures for the domain. The original architecture will serve as a reference architecture only. We will interpret the requirements liberally to allow for different architectures and design decisions.

A fire alarm system consists of sensors, actuation devices, communication devices and so on. In an industrial setting there may be hundreds or even thousands of these devices. The purpose of the software system is to manage these devices and their software representations. In addition, the communication between these devices needs to be handled. Since it is vital that a fire alarm is activated within a predetermined time interval after the sensors detect that there is fire, there are a number of real-time and security requirements on the operation of the system. It would be dangerous, for instance, if there would be much delay in time between the detection of a fire and the activation of the alarm. Because of this, a fire alarm system must comply with government-enforced regulations for such delays. Another important element in this case is that the software has to be able to deal with large industrial setups, meaning that there may be thousands of sensors and actuators.

Functional Requirements.

- Read sensor values
- Evaluate sensor values and determine if they deviate from preset trigger values.
- Trigger actuators when appropriate.

Quality Requirements.

- **Real-time behaviour.** The performance of the system has to scale in such a way that the predetermined period of 3 seconds between detection and alarm is never exceeded.
- **Scheduling.** The software will run on a simple OS, meaning that we will have to implement our own scheduling.

In the remainder of this section we discuss a number of different approaches to modelling this architecture. We have used the architecture design method presented in [Bosch 2000] to design the various versions of the architecture. In this method, the design starts with a functional design. In subsequent design iterations, changes are incorporated to adjust the architecture to the quality requirements.

3.2 Functional design

The first version of the fire alarm does not take the quality requirements into account and is based on the functional requirements only. The functionality can be described as follows: A sensor can be requested to measure itself; It then compares its value to some trigger and establishes whether it deviates from the trigger. In our notation this is expressed as follows:

```

fragmentinterface ISensor(in request; out returnDeviation)

fragment Sensor implements ISensor
begin
  request ;
  measure ;
  process ;
  returnDeviation
end

```

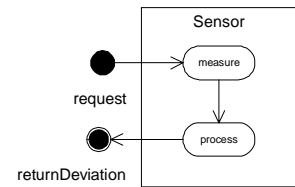


Figure 1 The Sensor

Note that we define both a fragmentinterface and a fragment. When composing fragments in the future we will use the fragmentinterface to allow for modified fragments (e.g. with extra behaviour superimposed on it). When a deviation occurs, an actuator (e.g. an alarm bell) needs to be activated. An actuator can be associated with multiple sensors. To establish whether an actuation is needed it has to check for deviations in all its sensors. The actual actuation strategy is left to the actuator (e.g. all sensors must have deviation or one deviating sensor can trigger the actuator). The actuator can be modeled as follows:

```

fragmentinterface IActuator(in start, receiveDeviation;
  out request, notActivated, activated)

fragment Actuator implements IActuator
begin
  start ;
  sendRequest {do this for all sensors} ;
  request ;
  receiveDeviation ;
  collect ;
  if alarm
  then activate ; activated
  else notActivated
  fi
end

```

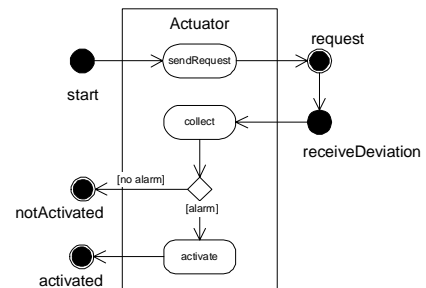


Figure 2 The Actuator

By composing the actuator and the sensor as in Figure 3, a simple version of the fire alarm can be made. In this version of the fire alarm, an actuator requests all its sensors for deviations and then decides whether to trigger the alarm. Also note that we compose the FunctionalFireAlarm from fragment interfaces rather than concrete fragments. This allows us to replace the components from which the fire alarm is constructed (i.e. sensors and actuators) with other fragments implementing the same interface.

3.3 Fire alarm with cached sensor deviations

The simple approach outlined above works for small systems. However, when multiple sensors and actuators are used, the communication grows exponentially. Especially, when one sensor is used by more than one actuator. A consequence of this may be that the system no longer complies with the regulations. To address this issue a caching mechanism may be introduced to reduce the redundant communication between sensors and actuators. A caching component can be expressed as follows:

```

fragmentinterface IFireAlarm(in start; out notActivated, activated)

fragment FunctionalFireAlarm implements IFireAlarm
begin
  fork
    IActuator(in start, receiveDeviation;
      out request, notActivated, activated)
  ||
    ISensor(in request; out returnDeviation)
  with IActuator.request = ISensor.request and
    IActuator.receiveDeviation = ISensor.returnDeviation
  end
end

```

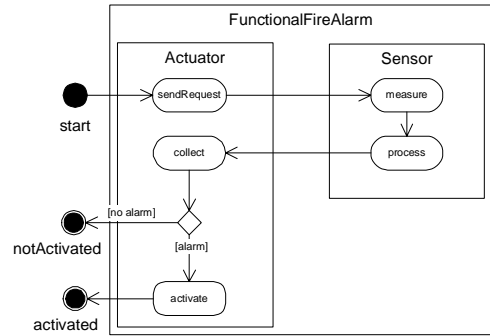


Figure 3 The Functional Fire alarm

```

fragmentinterface ICache(
  in request, receiveAnswer;
  out passRequest, returnAnswer)

fragment Cache implements ICache
begin
  request ;
  if requestCached
  then getValueFromCache
  else passRequest ; receiveAnswer ; cacheAnswer
  fi ;
  returnAnswer
end

```

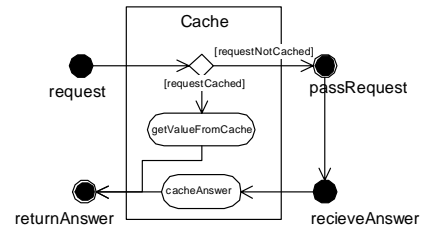


Figure 4 The Cache

We are then faced with the choice whether to compose it with the sensor and actuator or whether to superimpose this on our previous simple fire alarm composition. Our notation allows for both approaches so we will discuss them both:

```

fragment ComposedCachingFireAlarm
  implements IFireAlarm
begin
  fork
    IActuator (in start, receiveDeviation;
      out request, notActivated, activated)
  ||
    ICache (in request, receiveAnswer;
      out passRequest, returnAnswer)
  ||
    ISensor (in request; out returnDeviation)
  with Actuator.request = Cache.request and
    Actuator.receiveDeviation = Cache.returnAnswer and
    Cache.passRequest = Sensor.request and
    Cache.receiveAnswer = Sensor.returnDeviation
  end
end

```

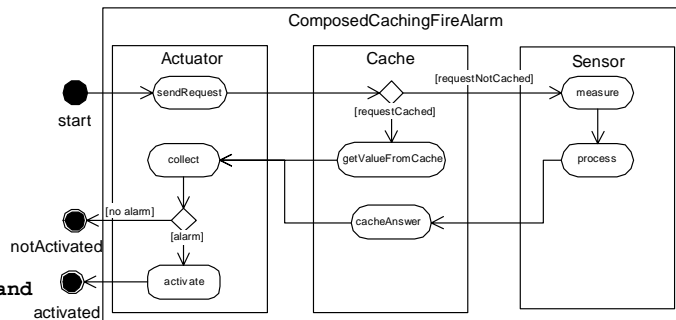


Figure 5 The Scheduled Caching Fire alarm

While the above compositions look different, they result in the same system. Arguably the first composition is more readable, however, the second composition has the advantage that it reuses the FunctionalFireAlarm composition (at the cost of exposing its internal activities because of the use of superimposition).

3.4 Scheduling

An additional requirement from the domain of fire alarm systems is that the system has to do application level scheduling. An application level scheduler can be expressed as follows: The scheduler can be composed with either of the compositions outlined above. As an example we will compose the scheduler with the last one.

The ScheduledCachingFireAlarm meets with all the requirements outlined before. However, it is not the same

```

fragment SuperImposedCachingFireAlarm
implements IFireAlarm
begin
  fork
    ICache (in request, receiveAnswer;
            out passRequest, returnAnswer)
    ||
    FunctionalFireAlarm (in start; out notActivated, activated)
  with Cache.receiveAnswer = FunctionalFireAlarm.sendRequest.after
  and Cache.receiveAnswer = FunctionalFireAlarm.process.after
  and Cache.passRequest = FunctionalFireAlarm.measure.before
  and Cache.returnAnswer = FunctionalFireAlarm.collect.before
  end
end

```

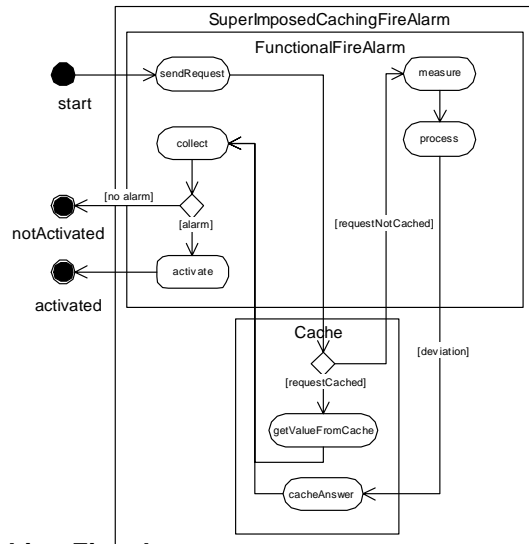


Figure 6 The SuperImposed Caching Fire alarm

```

fragment interface IScheduler (in start; out tick)
fragment Scheduler implements IScheduler
begin
  start ;
  schedule ;
  tick {for all scheduled objects}
end

```

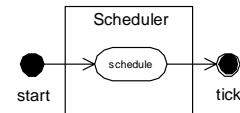


Figure 7 The Scheduler

solution as the one chosen in the original case study. In the remainder of this section we will discuss alternative solutions and demonstrate the flexibility of our notation by reusing as much as possible from what we have defined up till now.

```

fragment ScheduledCachingFireAlarm implements IFireAlarm
begin
  fork
    IScheduler (in start; out tick)
    ||
    SuperImposedCachingFireAlarm (
      in start; out notActivated, activated)
  with Scheduler.tick =
    SuperImposedCachingFireAlarm.start
  end
end

```

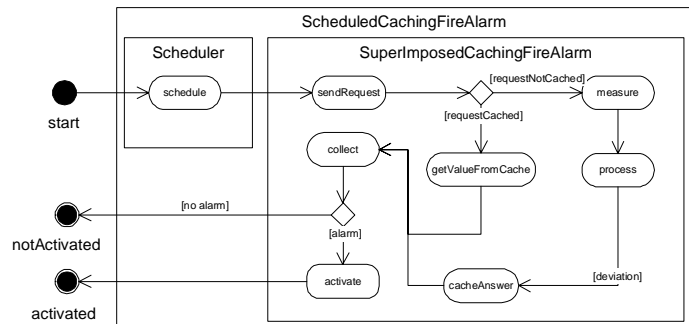


Figure 8 The Scheduled Caching Fire Alarm

3.5 Blackboard solution

The ScheduledCachingFireAlarm still has one disadvantage: it may potentially poll a lot of Sensors (if they have not been polled before). Also, there is no way for the cache to determine whether the cached value is still correct. To solve this a blackboard architecture can be used. In a blackboard architecture, sensors update their deviations on a central blackboard at regular intervals. The actuators poll the blackboard and receive the latest value. A blackboard can be expressed like this:

In combination with the scheduler, a replacement for ScheduledCachingFireAlarm can be made. This is done by first composing Scheduler with Sensor and Actuator to create ScheduledSensor and ScheduledActuator. Since this is a trivial composition, we leave it as an exercise to the reader and proceed with the composition with the Blackboard:

```

fragmentinterface IBlackboard(in: request, newValue ; out: returnValue)

fragment Blackboard implements IBlackboard
begin
  fork
    request ;
    getValueFromCache ;
    returnValue
  ||
    newValue ;
    cacheTheValue
  end

```

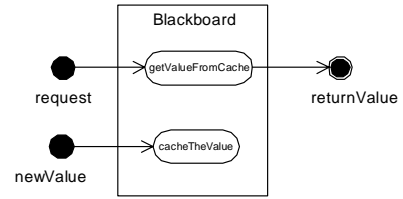
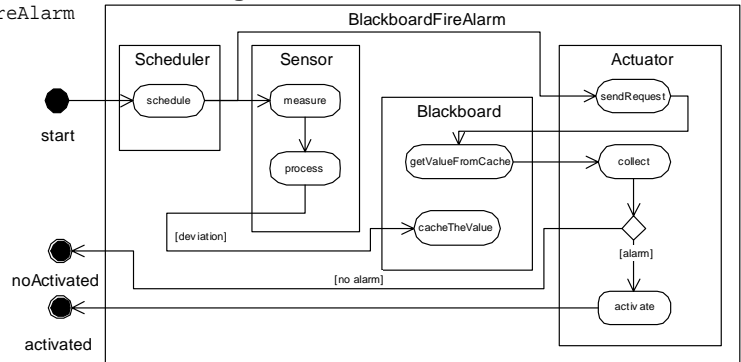


Figure 9 the Blackboard fragment

```

fragment BlackboardFireAlarm implements IFireAlarm
begin
  fork
    IScheduler(in start; out tick)
    ||
    ISensor(in request; out returnDeviation)
    ||
    IBlackboard(in: request, newValue
      out: valueStored, returnValue)
    ||
    IActuator(in start, receiveDeviation;
      out request, notActivated, activated)
  with ISensor.returnDeviation =
    IBlackboard.newValue and
    IBlackboard.request = IActuator.request and
    IBlackboard.returnValue = IActuator.receiveDeviation and
    IScheduler.tick = ISensor.request = IActuator.start
  end

```



```

IBlackboard.request = IActuator.request and
IBlackboard.returnValue = IActuator.receiveDeviation and
IScheduler.tick = ISensor.request = IActuator.start

```

end

Figure 10 The Blackboard Fire alarm

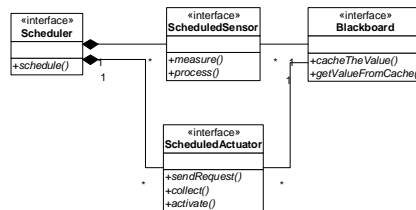


Figure 11 Blackboard Fire Alarm Derived Class Diagram

Once again, the fragment has the same parameters as the previous compositions. This means that it can be used in any place the previous compositions are used. Unfortunately, it is not possible to reuse the Functional-FireAlarm since the control flow is reversed (i.e. the sensor updates the blackboard rather than that the blackboard polls the sensor). However, the BlackboardFireAlarm implements the same interface as the previous alarm fragments so they can be used interchangeably.

It should be noted that in the above composition we have coupled the Scheduler's tick-activity to both the Sensor's request activity and the Actuator's start activity. We have declared an activity in three fragments to be equal and this conflicts with the restriction for the formal blending operator from trace theory to be associative. Since the Scheduler is put in front of the Sensor and the Actuator, we still have the associative property, however (proof is left to the reader). In general, however, this may not be the case.

3.6 Detailed Design of the Blackboard-based Fire Alarm

As an example, we will provide a derivation of a detailed design based on the BlackboardFireAlarm from Section 3.5. The main reason we use that example is because it is the one that resembles the original fire alarm architecture from [Bosch & Molin] the most.

The BlackboardFireAlarm is a composition of three other fragments (ScheduledSensor, ScheduledActuator and BlackBoard), two of these fragments are themselves compositions (Sensor, Actuator and Scheduler). Based on this decomposition four interfaces can be defined (Scheduler, ScheduledSensor, ScheduledActuator and Blackboard). A class diagram laying out the relations between the interfaces can be found in Figure 11.

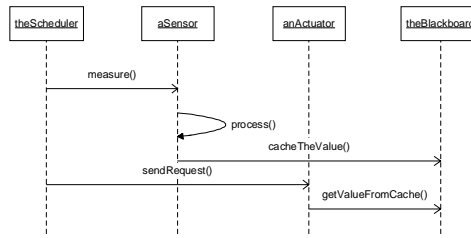


Figure 12 Blackboard Fire Alarm Derived Sequence Diagram

While this diagram may appear to be simple, one should realize that this is only the starting point for the detailed design, not the end result. During detailed design this diagram may be refined in various ways.

An example of such a refinement may be the application of the composite design pattern [Gamma et al. 1995] to deal with scheduling. In the diagram above Sensor and Actuator have no parent interface defining a tick method. Consequently, there are two separate relations to the scheduler rather than one to this parent interface. Another refinement may be to have multiple class implementations of the Sensor and Actuator interface. A whole class hierarchy of such implementations could be defined without affecting the architecture level representation or breaking its design rules. In Figure 12, a sequence diagram for the architecture is given. In this sequence diagram, archetypical sensors and actuator interactions are illustrated using the methods defined in the class diagram of Figure 11.

The diagrams we provide in Figure 11 and Figure 12 can be used as a starting point for the detailed design. In order to obtain an implementable design (which is the purpose of creating a detailed design), design decisions about such things as data formats, properties of objects, utility classes, inheritance and containment hierarchies, etc. need to be taken. Since such design decisions may have an architectural impact, there is no way back and the original architecture design will need to be revised to keep it synchronized with the evolving detailed design.

4 Analysis

Using our architecture modelling notation approach, we have created a number of different compositions of fragments. In this section, we will provide an analysis of the application of the notation on the case in Section 3. Also we will reflect on the issues outlined in the introduction.

4.1 Problems and Solutions

In the introduction we identified a number of problems. In this section we will argue how the notation addresses the issues outlined in the introduction.

Separation of Concerns. Our notation provides support for superimposition. This means that we can alter a fragment by imposing another fragment on it. The superimposition mechanism can be used to separately define concerns and impose them where necessary. An example of this is the way we impose caching on the functional firealarm in Section 3.3. The caching fragment is fitted between the actuator and sensor fragment, transparently changing the way these two fragments interact. The resulting caching firealarm has the same externally visible fragmentinterface so any composition it is involved in will be unaffected by the change.

Withdrawing design decisions. Compositions of fragments can be altered easily by replacing parts of this expression with similar (i.e. providing at least the same ins and outs) other parts. An example of an application of this feature would be to design a system with a fire alarm embedded. Initially the FunctionalFireAlarm could be used. Later on, it could be replaced by one of the other fire alarm fragments easily (see also substitutability).

Substitutability (i.e. a is-a relation) is one of the three properties Szyperski identifies as essential of inheritance (the other two are inheritance of interfaces, inheritance of implementation) [Szyperski 1997]. Since our notation is an architecture level notation, it does not provide implementation inheritance. However, by providing an interface construct we can support the other two. An example of this is the IFireAlarm interface we provide. In our example, several fragments are defined that implement this interface. However, when using the fire alarm in a composition it doesn't really matter which one is used (i.e. the different variants are substitutable).

Superimposing new decisions. We have used superimposition to add caching to the functionalfirealarm in

Section 3.3. Superimposition is transparent to the fragment that is subjected to it. Consequently, no unnecessary dependencies are created between design decisions. This allows us to use the functional fire alarm architecture in some composition and then later we are still able to add caching to this larger composition in exactly the same way.

4.2 Lessons learned

Abstracting from data. Our notation deliberately has a strong focus on functionality. We have found that abstracting from such details as data format and types allows us to capture the essence of an architecture. A Sensor is thus reduced to an entity that returns something when asked for it. What exactly is returned (and how) is an implementation detail. The fact that there will probably be different kinds of sensors with varying properties like what is measured, what kind of information is returned and how accurate the measurement is not an architectural concern. What matters at the architectural level is that there is an entity called sensor (i.e. the sensor fragment) which performs some archetypical behaviour characteristic of sensors (i.e. the activities) and fits in with the other architectural entities in a certain way (i.e. the fragment interface).

No clear boundary between architecture and detailed design. Our intention was to create a representation that is simple yet expressive enough to capture common architecture idioms and patterns (e.g. the architectural styles from [Buschmann et al. 1996]). We believe that our notation meets these criteria, however, in trying to keep things simple we have had to ask ourselves the question whether modelling a particular aspect of a design was an architecture design issue or a detailed design issue (in which case our notation would not need to support it). We have found that this is a rather grey area and we are aware that architecture and detailed design are not independent activities. Rather the architecture design evolves with the detailed design and often new requirements, requiring architectural changes, become apparent when working on the detailed design. This notion is also a motivation for our future work plans.

Graphic support is essential. In this paper, three notations ranging from very formal to a UML diagram have been discussed. We have found that it is generally much harder to understand one dimensional text representations than two dimensional graphics. Traditionally, things like separation of concerns and composition have been expressed using source code primarily. An important contribution of our paper is that we have shown how to do it by manipulating diagrams.

4.3 Remaining Issues

Traceability of design decisions. Considering that software development is generally an iterative process (as opposed to the waterfall model of software development), architecture notations, such as ours, share a common problem: important information is lost when progressing from one phase to another. Our notation is not different in that respect. For instance, a feature of our notation is the ability to define superimposition of fragments onto existing fragments. When a detailed design is derived however, this information is lost (the full composition is used to derive the detailed design). When later changes in the evolving detailed design need to be propagated to the architecture design, the original architecture design may no longer be accurate and it will have to be recovered from the detailed design. Since the detailed design notation has no means to express such things as superimposition, this information is lost. Note that this is not just an issue with our notation. To the best of our knowledge, any ADL available today suffers from this problem. This problem used to also apply to the detailed design phase vs. the implementation phase. However, the emergence of sophisticated CASE tools that integrate source code and UML notations has addressed this to a large extent. We believe that the solution to the issue lies in extending the support of such tools to architectural level notations, such as ours. The UML based nature of our notation may be helpful in achieving this.

Non-deterministic derivation. An issue that also needs to be considered in order to do so is that the detailed design derivation process is not deterministic. A consequence of specifying architecture fragments in a generic way is that there are multiple detailed designs that conform to such an architecture. Consequently, the derivation process has to allow for multiple derivations. Which derivation process is chosen, largely depends on design decisions that we consider to be part of the detailed design phase.

Separation of concerns in the Detailed Design. Our notation can be used to express separated concerns at the architectural level. Existing approaches towards separation of concerns mostly work on the implementation level. This leaves the detailed design as an area where support for separation of concerns has yet to be added. Once this is accomplished, it is possible to trace concerns throughout the whole development process. Currently this information is simply not included during detailed design due to a lack of suitable notations. Consequently, concerns are not designed/implemented until work on the implementation has started.

5 Conclusion

In this paper we have provided a notation for defining architecture fragments and defined its semantics using a formal notation. To illustrate how the notation works, we have used a pseudo code notation. However, we expect that in practice the graphic notation may be preferred as a more efficient means of communicating design decisions whereas the pseudo code notation may be used to provide additional details and prototyping. Also we have found the graphical way of doing composition and superimposition is quite intuitive.

The main advantages of our notation are:

- It abstracts from distracting details that really belong to the detailed design.
- It provides support for both composition and superimposition.
- It allows for some flexibility in the order in which design decisions are applied.

Because of this, it is easy to define different variants of the same architecture, apply an architectural style and compose existing architecture fragments.

5.1 Future Work

Our approach is an architectural level approach. We chose to operate on this level first because decisions made during this phase have a large impact on the subsequent development of a system. Now that we have this approach in place we can start thinking about extending it to the detailed design level. We feel that such a step is necessary as information is lost in the derivation process outlined in Section 2.4. This makes it hard to evolve a system in an iterative fashion since this requires a continuous effort to keep the architecture design in line with the detailed design.

In addition, we would like to do a more extensive case study to learn more about the effectiveness and applicability of the notation. In addition we would like to learn more about what concerns drive the architecture design using conventional techniques. At the moment of writing, we are preparing a case study at a local company that will provide us some feedback.

References

- Allen 1997.** Robert J. Allen, “*A Formal Approach to Software Architecture*”, Ph.D. Thesis, CMU-CS-97-144 School of Computer Science, Carnegie Mellon University, 1997.
- Bass et al. 1998.** L. Bass, P. Clements, R. Kazman, “*Software Architecture in Practice*“, Addison-Wesley, 1998.
- Beck 1999.** K. Beck, “*Extreme Programming Explained*“, Addison Wesley 1999.
- Bosch 1999.** J. Bosch, “Superimposition: A Component Adaptation Technique“, *Information and Software Technology*, 1999.
- Bosch 2000.** J. Bosch, “*Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*“, Addison-Wesley, ISBN 020167494-7, 2000.
- Bosch & Molin.** J. Bosch, P. Molin, “Software Architecture Design: Evaluation and Transformation“, *Proceedings of the 1999 IEEE Engineering of Computer Based Systems Symposium (ECBS99)*, December 1999.
- Buschmann et al. 1996.** F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, “*Pattern-Oriented Software Architecture: A System of Patterns*“, John Wiley & Sons, 1996.
- Fowler & Scott 1998.** M. Fowler, K. Scott, “UML Distilled - applying the standard object modeling language“, *Addison-Wesley*, 1998.
- Gamma et al. 1995.** E. Gamma, R. Helm, R. Johnson, J. Vlissides, “*Design Patterns: Elements of Reusable Object-Oriented Software*“, Addison-Wesley Publishing Co., Reading MA, 1995.
- Garlan et al. 1997.** D. Garlan, R. T. Monroe, D. Wile, “Acme: An Architecture Description Interchange Language“, *Proceedings of CASCON '97*, November 1997.
- Van Gorp & Bosch 2001.** J. Van Gorp, J. Bosch, “Design Erosion: Problems & Causes“, submitted March 2001.
- Van Gorp, Bosch & Svahnberg 2001.** J. Van Gorp, J. Bosch, M. Svahnberg, “On the Notion of Variability in Software Product Lines“, accepted for WICSA 2001.
- Harrison & Ossher 1993.** W. Harrison, H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)“, *Proceedings of OOPSLA '93*, pp 411-428.

- Hoare 1985.** C.A.R. Hoare, “*Communication sequential processes*”, Englewood Cliffs, NJ: Prentice Hall, 1985.
- IEEE1471 2000.** IEEE, “*Recommended Practice for Architectural Description of Software-Intensive Systems*”, Std 1471-2000.
- Jaktman, Leaney & Liu 1999.** C.B. Jaktman, J. Leaney, M. Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", *The First Working IFIP Conference on Software Architecture* (WICSA1), Kluwer Academic Publisher, 22-24 February 1999, San Antonio, TX, USA.
- Jazayeri, Ran & Van Der Linden 2000.** M. Jazayeri, A. Ran, F. Van Der Linden., “*Software Architecture for Product Families: Principles and Practice*”, Addison Wesley Longman, 2000.
- Kiczalez et al. 1991.** G. Kiczales, J des Rivieres, D. G. Bobrow. “*The Art of the Metaobject Protocol*”. MIT Press, 1991. ISBN 0-262-61074-4
- Kiczalez et al. 1997.** G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, “Aspect Oriented Programming“, *Proceedings of ECOOP 1997*, pp. 220-242.
- Luckman 1996.** D. C. Luckham, “Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events”, *DIMACS Partial Order Methods Workshop IV*, Princeton University, July 1996.
- Milner 1993.** R. Milner, “*Communication and concurrency*”, Englewood Cliffs, NJ: Prentice Hall, 1993.
- Molin & Ohlsson.** P. Molin, L. Ohlsson, “Points & Deviations - A pattern language for fire alarm systems“, in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.
- OMG UML.** OMG, UML Specification, <http://www.omg.org/technology/uml/>.
- Ousterhout 1998.** J.K. Ousterhout, “Scripting: Higher Level Programming for the 21st Century”, In *IEEE Computer Magazine*, March 1998.
- Perry & Wolf.** D. E. Perry, A.L. Wolf, “Foundations for the Study of Software Architecture“, *ACM SIGSOFT Software Engineering Notes*, vol 17 no 4.
- Roberts & Johnson 1998.** D. Roberts, R. Johnson, "Patterns for Evolving Frameworks", in *Pattern Languages of Program Design 3* p471-p486, Addison-Wesley, 1998.
- Shaw & Garlan 1996.** M. Shaw, D. Garlan, “*Software Architecture: Perspectives on an Emerging Discipline*”, Prentice Hall, April 1996.
- Snepscheut 1985.** J.L.A. van de Snepscheut, “Trace Theory and VLSI design”, *Lecture Notes in Computer Science*, vol. 200, Springer Verlag, 1985.
- Szyperski 1997.** C. Szyperski, *Component Software - Beyond Object Oriented Programming*. Addison- Wesley 1997.
- Tarr, Ossher & Harrison 1999.** P. Tarr, H. Ossher, W. Harrison, “N Degrees of Separation: Multi-Dimensional Separation of Concerns“, *Proceedings of ICSE’99*, pp. 107-119.