Architectural Design Support for Composition & Superimposition

Jilles van Gurp, 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 build 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.

1 Introduction

The ever growing size and complexity of software systems is making it increasingly harder to build systems that both meet current and future requirements. In earlier work [10], 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) [18], Subject Oriented Programming (SOP)[12] and Multi Dimensional Separation of Concerns (MDSC)[29]. 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.

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. [18][12][29]). 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 in that order 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. 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 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 tot 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 Running example

As a running example, we will use a fire alarm system that we used in an earlier case study [5] and [21]. In the original version of this fire alarm system, a number of design decisions are taken to optimize behavior of the architecture for real time and performance requirements.

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.

1.4 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. In Section 5 related work is discussed. And we conclude our paper in Section 6.

2 Notation

In [11], 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.

In this paper we focus on the architecture design phase. While this phase can be revisited later in the development (which is common in iterative development methods), 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.

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 [6], design patterns [8] and the developer's personal experience. Using this information, software architects construct the architecture by taking design decisions. 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.

2.1 Formal Notation

Table 1: Notation

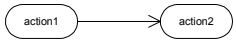
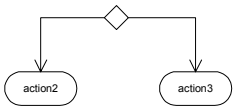
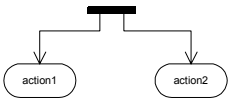
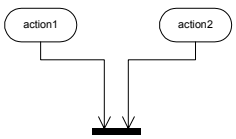
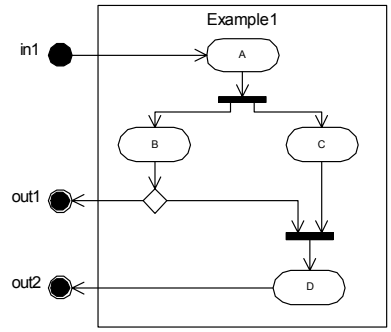
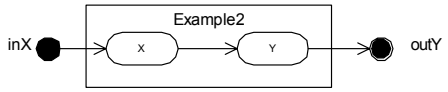
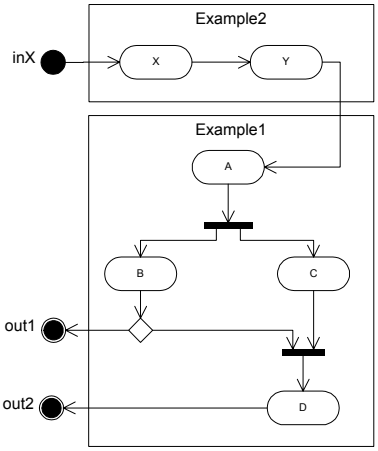
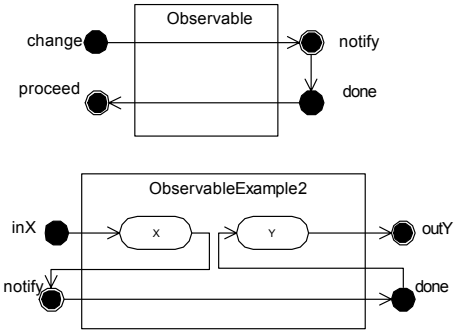
Graphical notation	Semantics	Pseudo code
	$action1 \bullet action2$	`action1 ; action2`
	$action1 \leftrightarrow action2$	**if** B **then** action1 **else** action2 **fi**
	$action1 \parallel action2$	**fork** action1 **||** action2 **end**
	$(action1 \bullet s) \parallel$ $(action2 \bullet s)$	**fork** action1 ; s **||** action2 ; s **end**

Table 1: Notation

	$Example1(in1;out1,out2)$ $=$ $in1 \bullet A \bullet$ $((B \bullet (out1 \leftrightarrow s))$ $\parallel$ $(C \bullet s)) \bullet$ $D \bullet out2$	**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**
	$Example2(inX;outY)$ $=$ $inX \bullet X \bullet Y \bullet outY$	**fragment** Example2 (**in** inX; **out** outY) **begin** inX ; X ; Y ; outY **end**
	$Compostion$ $=$ $Example2(inX, s)$ $\parallel$ $Example1(s, out1, out2)$ (where $inX = outY$), The result is: $inX \bullet X \bullet Y \bullet A \bullet$ $((B \bullet (out1 \leftrightarrow s)) \parallel$ $(C \bullet s)) \bullet$ $D \bullet out2$	**fragment** Composition (**in** inX; **out** out1, out2) **begin** **fork** example (in1, out1, out2) **||** example2 (inX, outY) **with** in1 = outY **end** **end**
	$Observable$ $=$ $change \bullet notify \bullet$ $done \bullet proceed$ The Composition is given by $inX \bullet X \bullet notify \bullet$ $done \bullet Y \bullet outY$	**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** = proceed **end** **end**

The notation we use is based on UML activity diagrams (also see [22]). Activity diagrams are used to model the dynamic behavior of a system as a series of activities that take place in a specified order. The activities can be loosely grouped into so-called swimlanes to indicate that they are related. Such swimlanes can, for example, be used to identify architectural components. By specifying compositions of these swimlane-fragments different architectures can be created.

In order to specify the composition of fragments, we use a formal notation that is equivalent to the graphical notation. The formal notation (also see [13] and [20]) first appeared in the *trace theory* approach of [27]. In this notation, a trace structure consists of an alphabet (a set of activities) and a trace set (all sequences of activities that are allowed in the structure; including their prefixes). We adopt the weaving and blending composition function of trace structures. In addition to this algebra, we also pro-

vide a pseudo code notation for enhanced readability. We use the formal notation only to define the semantics of our notation.

In this paper we use a, b, c for atomic 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. Common activities in P and Q are used for synchronization. For example $(a \bullet b \bullet c) \parallel (b \bullet d)$ uses b for 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.

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 out-going triggers of the swimlane. Such a representation of a swimlane is called a *fragment* (see Table 1 for an example).

2.2 Composition

Composition of a number of fragments can be accomplished by using the ||-operator together with the synchronization mechanism. Common activities are used as internal activities for synchronization. In Van de Snepscheut [27] 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 composition of `Example1 and Example2` that is created by connecting `outY` with `in1` (we map an out-going trigger with an in-going trigger). `outY` (or `in1`) is used as the common internal activity and is left out of the resulting composition since we use the blending function. 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 $(a \parallel (b \parallel c)) = ((a \parallel b) \parallel c)$ Van de Snepscheut [27] proves that the corresponding blending-operation is also 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 || operation. Associativity 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.

2.3 Superimposition

A second form of composition that is supported in our notation is superimposition (also see [3]). Superimposition allows for composition of a fragment with activities inside a fragment (i.e. the fragments internal behaviour is enhanced, unfortunately this breaks associativity as defined in the previous section). In order to express superimposition 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. $a \bullet e_1 \bullet e_2 \bullet \ldots \bullet e_n \bullet b$, where each e_i is an anonymous activity. In our pseudo code 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. If the internal activity goes just before, or just after a decision-node, we 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 [17]) use a similar mechanism.

2.4 Interfaces

When composing fragments the internal description is not needed, except when using superimposition. Therefore we introduce *fragment interfaces* that allow us to abstract from a fragment's internals. 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 pseudo code 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, ...*

2.5 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.

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. Consequently, we use a subset of the constructs typically found in a class diagram. Rather than specifying classes, we specify interfaces. 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.

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 out-going 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 (the in and out activities are lost in the blending process).

UML uses several types of relations, which can all be used to model delegation. The weakest 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, navigability can be used on the associations (this makes the association uni-directional).

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.

3 Examples: The Fire Alarm system

In the introduction we already mentioned the fire alarm case briefly. To illustrate our technique, we applied it to this case. The subject of the case is the creation of an architecture for a fire alarm system. In the earlier case studies [5][21] we described an architecture for this domain. In this paper we will use the requirements that were associated with this architecture and use them to create various architectures for the domain. 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 [4] 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 ad-

just the architecture to the quality requirements.

3.1 Functional design

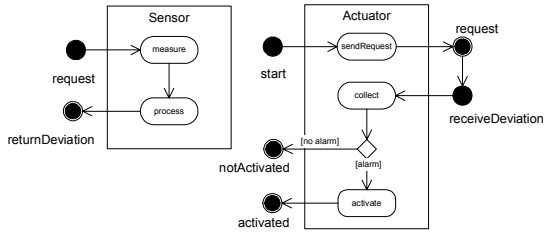


Figure 1 The Sensor and Actuator

The first version of the architecture 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. When a deviation occurs, an actuator (e.g. an alarm bell) needs to be activated (see Figure 1 for both fragments).

An actuator can be associated with multiple sensors. To establish whether 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).

By composing the actuator and the sensor as in Figure

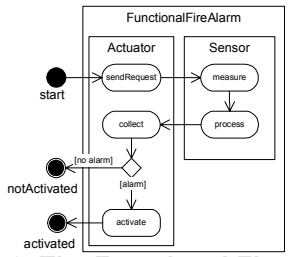


Figure 2 The Functional Fire alarm

2, 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.

As an example we also provide the composition in pseudo code. In the remainder of the paper we will omit pseudo code examples.

```
fragmentinterface ISensor
  (in request; out returnDeviation)
fragment Sensor implements ISensor
begin
   request ; measure ; process ; returnDeviation
end

fragmentinterface IActuator
( in start, receiveDeviation;
  out request, notActivited, 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
```

```
fragmentinterface IFireAlarm(in start; out
notActivated, activated)

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

3.2 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 (Figure 3) may be introduced to reduce the redundant communication between sensors and actuators.

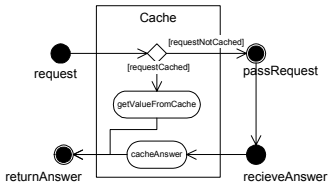


Figure 3 Cache

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 demonstrate them both (Figure 4 and Figure 5).

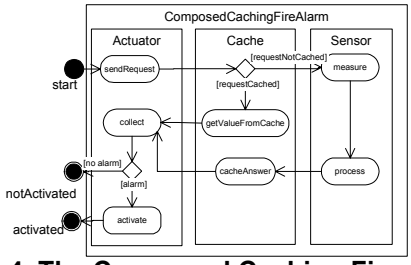


Figure 4 The Composed Caching Fire Alarm

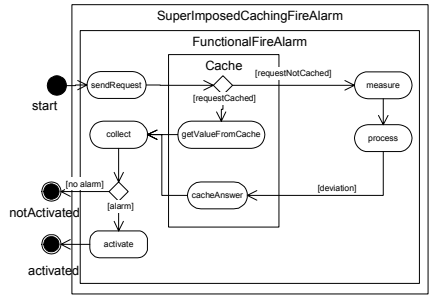


Figure 5 The SuperImposed Caching Fire alarm

The first approach uses ordinary composition however, the second composition (that uses super imposition) has the advantage that it reuses the FunctionalFireAlarm com-

position (at the cost of exposing its internal activities because of the use of superimposition).

3.3 Scheduling

An additional requirement from the domain of fire alarm systems is that the system has to do application level scheduling. The scheduler (Figure 6) can be composed with either of the compositions outlined above. As an example we will compose the scheduler with the fragment in Figure 5. The ScheduledCachingFireAlarm meets with all

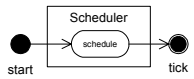


Figure 6 The Scheduler

the requirements outlined before. 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.

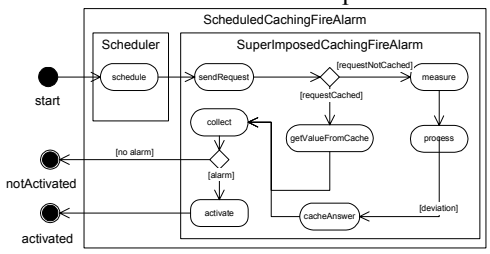


Figure 7 The Scheduled Caching Fire Alarm

3.4 Blackboard solution

The ScheduledCachingFireAlarm 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 (Figure 8), sensors update their deviations on a central blackboard at regular intervals. The actuators poll the blackboard and receive the latest value.

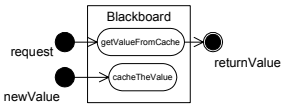


Figure 8 the Blackboard fragment

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 just present the composition with the Blackboard in Figure 9.

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 FunctionalFireAlarm since the control flow is reversed (i.e. the sensor updates the blackboard rather than that the blackboard polls the

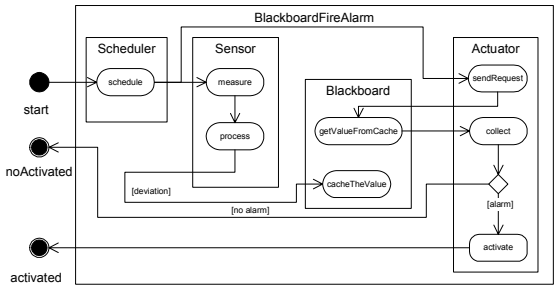


Figure 9 The Blackboard Fire alarm

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.

4 Analysis

Using our architecture modeling 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 3.2. 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 with similar 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. 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) [28]. 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 3.2. 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, is not an architectural concern and should therefore not be specified or constrained in the architecture design. What matters at the architectural level is that there is an entity called sensor (i.e. the sensor fragment) which performs the archetypical behaviour of sensors and fits in with the other architectural entities in a certain way.
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 [6]). 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.
Graphical 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 architecture 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, however.
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 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 [26] and [2] 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."* [14].

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."* [16]. 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 [6][8]. 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 [6].

Architecture Erosion. A motivation for writing this paper was the idea that due to requirement changes, architectures tend to erode over time. In [10], 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 [16], any changes in these decisions will affect the architecture. The notion of architecture erosion was first identified in [24]. In [15], 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 AOP [18], SOP [12] and Multi Dimensional Separation of Concerns [29]. 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 concerns.

Composition. Our composition technique bears some resemblance to the notion of super-imposition discussed by one of the co-authors [3]. 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 by Bosch [3] suggests an implementation/detailed design technique, our notation is intended for use on the architectural level.

Scripting. In [23], 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 scripting 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 [22]. The reason we use this notation instead of, for instance, ACME [9], Rapide [19] or WRIGHT [1], is twofold. 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 [7].

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 [25] we can describe superimposition [3]. WRIGHT is close to our approach because it is based on CSP [13]. In our approach a more subjective notation is used and it is based on trace theory [27] that has less basic principles but is sufficiently expressive, nevertheless.

6 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.

6.1 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 2.5. 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

[1] Robert J. Allen, *"A Formal Approach to Software Architecture"*, Ph.D. Thesis, CMU-CS-97-144 School of Computer Science, Carnegie Mellon University, 1997.

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

[3] J. Bosch, "Superimposition: A Component Adaptation Technique", *Information and Software Technology*, 1999.

[4] J. Bosch, *"Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach"*, Addison-Wesley, ISBN 020167494-7, 2000.

[5] J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", *Proceedings of the 1999 IEEE Engineering of Computer Based Systems Symposium (ECBS99)*, December 1999.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *"Pattern-Oriented Software Architecture: A System of Patterns"*, John Wiley & Sons, 1996.

[7] M. Fowler, K. Scott, "UML Distilled - applying the standard object modeling language", *Addison-Wesley*, 1998.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Publishing Co., Reading MA, 1995.

[9] D. Garlan, R. T. Monroe, D. Wile, "Acme: An Architecture Description Interchange Language", *Proceedings of CASCON '97*, November 1997.

[10] J. van Gurp, J. Bosch, "Design Erision: Problems and Causes" , submitted May 2001.

[11] J. Van Gurp, J. Bosch, M. Svahnberg, "On the Notion of Variability in Software Product Lines", accepted for WICSA 2001.

[12] W. Harrison, H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *Proceedings of OOPSLA '93*, pp 411-428.

[13] C.A.R. Hoare, *"Communication sequential processes"*, Englewood Cliffs, NJ: Prentice Hall, 1985.

[14] IEEE, *"Recommended Practice for Architectural Description of Software-Intensive Systems"*, Std 1471-2000.

[15] 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.

[16] M. Jazayeri, A. Ran, F. Van Der Linden., *"Software Architecture for Product Families: Principles and Practice"*, Addison Wesley Longman, 2000.

[17] G. Kiczales, J des Rivieres, D. G. Bobrow. *"The Art of the Metaobject Protocol"*. MIT Press, 1991. ISBN 0-262-61074-4

[18] 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.

[19] 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.

[20] R. Milner, *"Communication and concurrency",* Englewood Cliffs, NJ: Prentice Hall, 1993.

[21] P. Molin, L. Ohlsson, "Points & Deviations - A pattern language for fire alarm systems", in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

[22] OMG, UML Specification, http://www.omg.org/technology/uml/.

[23] J.K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", In *IEEE Computer Magazine*, March 1998.

[24] D. E. Perry, A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, vol 17 no 4.

[25] D. Roberts, R. Johnson, "Patterns for Evolving Frameworks", in *Pattern Languages of Program Design 3* p471-p486, Addison-Wesley, 1998.

[26] M. Shaw, D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, April 1996.

[27] J.L.A. van de Snepscheut, "Trace Theory and VLSI design", *Lecture Notes in Computer Science*, vol. 200, Springer Verlag, 1985.

[28] C. Szyperski, *Component Software - Beyond Object Oriented Programming*. Addison- Wesley 1997.

[29] P. Tarr, H. Ossher, W. Harrison, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *Proceedings of ICSE'99*, pp. 107-119.