Aspect Interactions: A Requirements Engineering Perspective

Thein Than Tun[1], Yijun Yu[1], Michael Jackson[1], Robin Laney[1], and Bashar Nuseibeh[1,2]

Abstract The principle of Separation of Concerns encourages developers to divide complex problems into simpler ones and solve them individually. Aspect-Oriented Programming (AOP) languages provide mechanisms to modularize concerns that affect several software components, by means of joinpoints, advice and aspect weaving. In a software system with multiple aspects, a joinpoint can often be matched with advice from several aspects, thus giving rise to emergent behaviours that may be unwanted. This issue is often known as the aspect interaction problem. AOP languages provide various composition operators: the *precedence* operator of AspectJ, for instance, instructs the aspect weaver about the ordering of aspects when advice from several of them match one joinpoint. This ordering of conflicting aspects is usually done at compile-time. This chapter discusses a type of problem where conflicting aspects need to be ordered according to runtime conditions. Extending previous work on Composition Frames, this chapter illustrates an AOP technique to compose aspects in a non-intrusive way so that precedence can be decided at runtime.

1 Introduction

Software systems are typically required to satisfy multiple concerns of several stakeholders. Users may want a software system to be responsive, and the computer interface to be intuitive. Sponsors of the software system may want the information to be handled securely. Programmers who maintain the software system may want to work with a program design that is easy to modify. The principle of Separation of Concerns encourages developers to address these concerns of performance, usability, security and maintainability individually. Yet, when composed together, these concerns make different and often conflicting demands on the system architecture,

[1]Department of Computing, The Open University, UK, e-mail: {t.t.tun, y.yu, m.jackson, r.c.laney, b.nuseibeh}@open.ac.uk .[2]Lero, Irish Software Engineering Research Centre, Limerick, Ireland

the program design, and other software artefacts. Aspect-Oriented Programming (AOP) languages provide mechanisms for implementing, in a modular fashion, concerns that cut across several components. Towards this end, AOP languages provide mechanisms for joinpoints, advice and aspect weaving, which have been explained and illustrated in [13].

The issue of feature interaction is well known in telecommunication and other software systems [1, 7, 4]. Generally, software features are thought to interact when features that individually satisfy the user requirements, when composed together, produce unwanted behaviour. The interactions are often due to conditions such as non-determinism, divergence and interference. When resolving such feature interactions, compile-time mechanisms are often over-restrictive in the sense that the composition has to be decided at compile-time and it cannot respond to runtime conditions.

For instance, in a smart home application [7], the security and climate control features may interact when the security feature shuts the window because the home owners are away but the climate control feature opens the window to allow fresh air in. This condition is known as divergence.

A similar issue can be observed in aspect composition. A program that has to satisfy multiple concerns may have a joinpoint that could be matched with advice from several aspects, corresponding with the concerns the component has to satisfy. When these aspects are composed, the weaver is free to choose the ordering of the aspects if the developer does not specify the desired ordering. Divergence here can be illustrated by the following main program and the two aspects in the syntax of AspectJ 6 (simply AspectJ henceforth).

```
// The main program Window.java
public class Window {
     public static void main(String[] args) {
          System.out.println("Window has now started.");
     }
}

// SecurityFeature.aj
public aspect SecurityFeature {
     after() returning: execution(* main(..)) {
          System.out.println("SecurityFeature: Window is now
                                  shut because it is night now.");
     }
}

// ClimateFeature.aj
public aspect ClimateFeature {
     after() returning: execution(* main(..)) {
          System.out.println("ClimateFeature: Window is now
                           opened because it is hot indoors.");
     }
}
```

Running the program could produce a seemingly random ordering of the two aspects. In one run of the program, the following output is produced, although an-

other valid ordering of aspects is also possible. Such uncontrolled behaviour may be unwanted, and therefore can be seen as a form of aspect interaction.

```
Window has now started.
SecurityFeature: Window is now shut because it is night now.
ClimateFeature: Window is now opened because it is hot indoors.
```

If a particular ordering of these aspects is desired, for instance, if the climate feature is always more important than the security feature, then the precedence of these aspects has to be declared. Since the advice of these aspects are applied after the execution of the main method, the so-called 'after' advice, they need to be listed in ascending order of priority.

```
// ComposeAspects.aj
public aspect ComposeAspects {
    declare precedence: SecurityFeature, ClimateFeature;
}
```

The program now resolves the aspect interaction and always produce the desired ordering of the aspects, namely that the climate control aspect is always executed before the security aspect:

```
Window has now started.
ClimateFeature: Window is now opened because it is hot indoors.
SecurityFeature: Window is now shut because it is night now.
```

This style of resolving aspect interactions is over-restrictive because once the precedence is defined at compile-time, it cannot be changed easily in order to respond to runtime conditions. The ordering of the security and climate control features in the example above cannot be changed at runtime, for instance.

In our previous work on feature composition, we have formalized the notion of Composition Frames which monitor the features being composed, and depending on the requirements and runtime conditions, determine the ordering of features [9]. This style of composition is more flexible and can be extended to aspect composition.

In this chapter, we show that features can be treated as aspects, and feature composition as aspect composition. We then discuss how Composition Frames can be used to compose aspects and resolve aspect interactions at runtime. We present a way to implement the aspect composition as a distinct crosscutting concern that can be treated as a separate aspect. We show that this approach to composing aspects at runtime is generic and non-intrusive.

2 Preliminaries

This section illustrates the notion of feature interaction using a simple problem from a smart home application [7] before discussing how Composition Frames can be used to resolve the feature interaction problem.

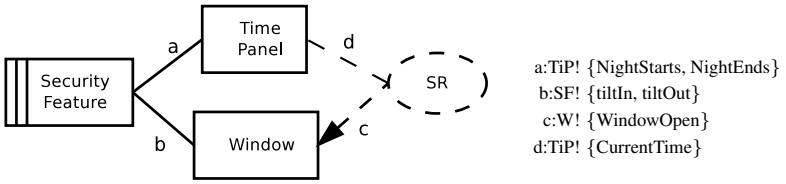


Fig. 1 Problem diagram for the security feature

2.1 Feature Interaction: An Example

Let us consider again a simple smart home application with two features, both of which control a motorized window that can be opened and shut. The security feature has a requirement for keeping the window shut at night. The requirement for the temperature feature is to keep the window opened when it is hot, meaning when the indoor temperature is higher than the required temperature and at the same time, the outdoor temperature is lower than the indoor temperature. An important characteristic of smart home applications is that their features may be developed independently by manufacturers. Therefore, conflicts between features may have to be detected and resolved at runtime.

When analyzing the requirements for these two features, we use problem diagrams [5] to show the relationship between three descriptions: (i) user requirements, (ii) problem world domains which make up the context of the software and (iii) specifications of the behaviour of the running software. The relationship between these descriptions is intended to indicate that the specifications, in the described context, will satisfy the requirements.

Fig. 1 shows the problem diagram for the security feature, where the requirement is denoted by a dotted oval, problem world domains are denoted by plain rectangles and the specification is denoted by a rectangle with two vertical stripes. The requirement SR says that the window should be kept shut at night.

The problem world domains are entities in the world that the program must interact with, such as Time Panel and Window, in satisfying the requirement SR. The solid lines (a and b) are domain interfaces representing shared variables and events between the domains and the machine involved. At the interface a, the variables NightStarts and NightEnds are controlled by Time Panel (as denoted by TiP!), and can be observed by the security feature. Descriptions of other interface labels can be read in the same way.

Assuming that NightStarts and NightEnds are variables for non-negative integers between 0 and 2400, when $NightStarts < CurrentTime$ and $CurrentTime < NightEnds$, it is night; otherwise, it is day. At the interface b, the security feature can fire two events tiltIn and tiltOut, and these events can be observed by the window. The property of Window is such that when tiltOut is observed, the window is open, meaning that WindowOpen is true. Likewise, when tiltIn is observed, the window is shut (WindowOpen is false). Dotted lines (c and d) denote requirement

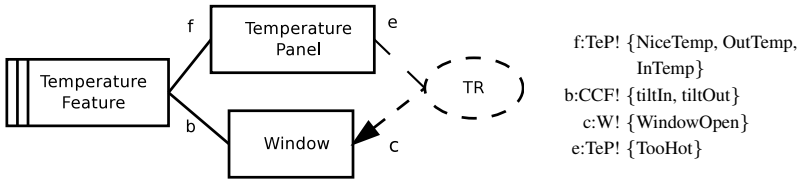


Fig. 2 Problem diagram for the temperature feature

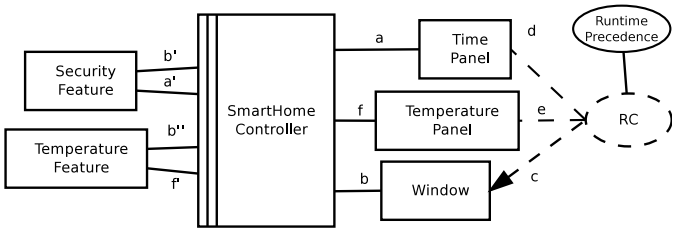
phenomena. The requirement is a desired relationship between the current time and the state variable of the window, namely that when $NightStarts < CurrentTime$ and $CurrentTime < NightEnds$ is true, WindowOpen should be false.

One description of the specification Security Feature is to fire the event tiltIn whenever it is night, and to ensure that tiltOut is not fired until the night ends. The relationship between the three descriptions is as follows: If the behaviour of the window and the time panel is as stated, the specification Security Feature satisfies the requirement SR. This simple specification, of course, ignores a number of issues: for instance, it does not check whether the window is already shut when night starts, or how long it takes for the window to fully open. Let us ignore such issues in our discussion.

The problem diagram for the temperature feature, shown in Fig. 2 is similar to the diagram in Fig. 1. The requirement here is that if it is too hot indoors, meaning that the desired temperature (NiceTemp) and the indoors and outdoors temperatures (OutTemp and InTemp) are in a certain relationship, the window should be kept open. The temperature readings are controlled by the temperature panel, and the temperature feature can observe them. One description of the specification Temperature Feature is to fire the tileOut event whenever the conditions $NiceTemp < InTemp$ and $OutTemp < InTemp$ hold and to ensure that the tiltIn is not fired as long as that relation remains true.

Notice that the two requirements above do not say anything about what to do during the daytime, and when it is not hot indoors. However, if the inside temperature is higher than the desired temperature, and the outside temperature is lower than the inside temperature, the window should be opened even if the outside temperature is higher than the desire temperature (thus not possible to possible achieve the required temperature just by opening the window).

Composing these two features can lead to a divergent behaviour under certain conditions. During a hot night, according to the temperature feature, the window should be open, but according to the security feature, the window should be shut. It is important to note that although the temperature feature will not close the window by firing the tiltIn event, it cannot stop the security feature from firing the same event during the hot night. Likewise, although the security feature will not open the window by firing the tiltOut event, it cannot stop the temperature feature from firing the same event during the hot night. In other words, an individual feature cannot have an exclusive control of the window over a length of time.



d:TiP! {CurrentTime}
a:TiP! {NightStarts, NightEnds}
e:TeP! {TooHot}
f:TeP! {NiceTemp, OutTemp, InTemp}
a':SC! {NightStarts, NightEnds}
b':SF! {tiltIn, tiltOut, ShutUntil}
f':SC! {NiceTemp, OutTemp, InTemp}
b'':TF! {tiltIn, tiltOut, OpenUntil}
b:SC! {tiltIn, tiltOut}
c:W! {WindowOpen}

Fig. 3 Composition of the security and temperature features

Furthermore, if a precedence operator is applied in the composition of these two features, one of the two features will always have priority over the other. This might be over-restrictive. It is sometimes desirable for the system to allow the user to indicate at runtime how the features should be ordered. Finally, in order to separate the concerns of individual features from the concern of composition, the two specifications should not be modified in order that they find out what the other feature is doing before carrying out their own actions. Our previous work on feature interaction shows that Composition Frames are suitable for such feature composition.

2.2 Resolving Feature Interaction using Composition Frames

As shown in Fig. 3, the two features can be composed by introducing the new software component SmartHome Controller, which is obtained by merging two wrappers that sit at the interfaces a and b of the security feature in Fig. 1 and the interfaces d and b of the temperature feature in Fig. 2 (see [15] for wrapper transformation rules). In effect, SmartHome Controller intercepts the information and events going in and coming out of the two features.

The variable ShutUntil is used by the security feature to indicate the time point until which it does not want other features to open the window. In principle, the value of ShutUntil is be determined by the value of NightEnds. The variable OpenUntil is used by the temperature feature to indicate the time point until which it does not want other features to shut the window. In principle, this is the first time point when InTemp is equal to NiceTemp. However, the temperature feature cannot know in advance how long the room will remain too hot. Therefore, it may have to set this time on a periodic basis.

Again, notice that each of the features does not prevent another feature from opening or shutting the window. Each feature only declares what it wants other features not to do within a certain duration. ShutUntil and OpenUntil will then be used by the SmartHome Controller to mediate when conflicts arise. Broadly speaking, a conflict occurs when two features attempt to maintain two contradictory properties. As discussed in [9], the values of ShutUntil and OpenUntil can be derived as part of the specification of the two features by means of the *Prohibit* predicate.

Runtime Precedence defines several ways in which conflicts can be resolved: we will call them the semantics of the composition operator. Although they can be defined more generally and precisely [9], we will focus on the specific example here.

- **No Control:** In this composition, the requirements for the security and temperature features should each be met at times when they are not in conflict; but when conflicts occur, any emergent behaviour is acceptable. It allows, for example, the window to oscillate in a partly open position. None of the requirements of the two features may be satisfied.
- **Exclusion:** In this composition, the requirements for the security and temperature features should each be met at times when they are not in conflict; but when conflicts occur, the requirement of the feature that started first should have priority. For example, if the security feature shuts the window before the temperature feature needs to open it, the temperature feature will not be able to shut the window until the security requirement has been satisfied. This exclusion is symmetrical.
- **Exclusion with Priority.** In this composition, the exclusion is asymmetrical, for instance, in favour of the security requirement. It means that the security feature can shut the window during the time in which the temperature feature wants the window open. The temperature feature, however, cannot open the window if the security feature wants it shut.

Other possible semantics include exclusion with event-level priority [9]. The requirement RC in Fig. 3 says that the window should be opened and shut according to the Runtime Precedence option the user of the smart home application has selected.

If the security and the temperature features are implemented as aspects, and if there is no definition of the ordering of these aspects using the precedence operator, the weaver will produce the "no control" behaviour defined above. The precedence operator of AspectJ can produce the composition similar to the behaviour defined by the "exclusion with priority" option. We now show how the "exclusion" option can be implemented using the aspect-oriented technique.

3 The Proposed Approach: Runtime Composition of Aspects

In this proposed approach, the problem world domains are implemented first as Java components, forming the base system. Features are then implemented as aspects

which weave into the base system of the problem world domains. This separation of aspects from the base system fits well with the separation of specifications from the problem world domains because like aspects, specifications can interface with multiple components, as highlighted in Fig. 3. Composition of the features is regarded as a separate concern that is implemented as an aspect in its own right.

3.1 Implementing the Problem World Domains

The problem world domains such as Window can be implemented in a straightforward way. We can simply define a singleton class for each of the domains, and their variables as class variables and events as methods. (Full listings of all programs in this section are provided in [14].)

```
class Window {
        static boolean WindowOpen;
        Window(){
                // code for initializing the window
                WindowOpen = false;
        }

        public void tiltOut() {
                // code for opening the window
                WindowOpen = true;
        }

        public void tiltIn(){
                // code for shutting the window
                WindowOpen = false;
        }
}
```

Other problem world domains are implemented in a similar fashion, but they are omitted here for space reasons. In the main method of the ProblemWorldDomains class, classes for window, time panel, temperature panel and runtime precedence are instantiated and initialized, as shown below.

```
public class ProblemWorldDomains {
       static Window win = new Window();
       static TemperaturePanel TeP = new TemperaturePanel();
       static TimePanel TiP = new TimePanel();
       static RuntimePrecedence rp = new RuntimePrecedence();

       public static void main(String args[]) {
              win.showStatus();
              TiP.NightStarts=2000;
              TiP.NightEnds=600;
              TeP.NiceTemp = 15;
              rp.Options = 1;
       }
}
```

3.2 Implementing the Features

At runtime, features such as the security and temperature features will be long-running and concurrent processes. Therefore, these features are implemented as threads. In principle, some features could be implemented without using aspects. However, to illustrate the problem of aspect interaction, both features our example are implemented using aspects.

```
public aspect SecurityFeature {
  static long shutUntil;
  class Spec implements Runnable {
    Thread runner;
    Window win;
    TimePanel tiPanel;

    public Spec(String threadName, Window w1, TimePanel tp1) {
      runner = new Thread(this, threadName);
      win = w1;
      tiPanel=tp1;
      // -1 indicates that no need to keep the window shut
      shutUntil=-1;
      runner.start();
    }

    public void run() {
      while (true){
        if (currentTime > tiPanel.NightStarts &&
            currentTime < tiPanel.NightEnds) {
          // It is night now, so the window should be shut
          // and should not be opened until tiPanel.NightEnds
          win.tiltIn();
          shutUntil=tiPanel.NightEnds;
          // do nothing until tiPanel.NightEnds
        }
        shutUntil =-1;
      }
    }
  }

  pointcut getWinRefs(): execution(* main(..));
  after(): getWinRefs() {
    Spec SF =new Spec("Security Feature",
                                  ProblemWorldDomains.win,
                                  ProblemWorldDomains.TiP);
  }
}
```

The aspect above implements the security feature by instantiating a new thread as soon as the main method has been executed. The program then gets hold of the references to the window object `win` and the time panel object `TiP` from the main method. The aspect also declares the variable `shutUntil` to indicate the time point until which the program wants the window to be shut. When the thread starts run-

ning, it continuously checks whether the current time is between `NightStarts` and `NightEnds`. Notice that the variable `currentTime` has to be declared and assigned appropriate values in a format compatible with `NightStarts` and `NightEnds`. If the current time is within the range, then the tiltIn method is called and the value of `shutUntil` is set to `NightEnds`.

It is worth emphasizing that the security feature does not stop the temperature feature calling the tiltOut method during the night. The temperature feature is implemented likewise: it opens the window when it is too hot and indicates the length of time it wishes to keep the window open by setting the value of `openUntil`. Since the two features are largely independent, they do not communicate with each other about what they do not want the other feature to do. This is in line with the principle of separation of concerns: individual features are not concerned with how they will be composed together.

Notice if the security and temperatures are implemented as singleton classes then the variables `shutUntil` and `openUntil` are to be treated as class variables.

This completes the implementation of the temperature and security features. If these programs are run, the ordering of the two features is entirely random, and will satisfy the "no control" option discussed above.

3.3 Implementing Composition Frames

The composition controller SmartHomeController is implemented by a separate aspect. This new aspect monitors the method calls made by the security and temperature features, and examines the `openUntil` and `shutUntil` to see whether calls to the `tiltOut` and `tiltIn` methods should proceed. As indicated in Fig. 3 the controller will rely on the runtime precedence option selected by the user. If the user wants the "exclusion" option, for instance, the `tiltOut` method call will be delayed until the time point `shutUntil` has passed, and the `tiltIn` method call will be delayed until the time point `openUntil` has passed. These delays could be achieved by putting the threads to sleep.

```
public aspect SmartHomeController {
  pointcut delayTiltIn() : call (void tiltIn(..));
  before(): delayTiltIn() {
    // the window is about to tiltIn
    if (TemperatureFeature.openUntil > 0) {
      // but the window should remain open
      if (ProblemWorldDomains.rp.Options == 1)  {
        // the user has selected the exclusion option
        // wait until TemperatureFeature.openUntil has passed
      }
    }
  }

  pointcut delayTiltOut() : call (void tiltOut(..));
  before(): delayTiltOut() {
```

```
      // the window is about to tiltOut
      if (SecurityFeature.shutUntil > 0) {
        // but the window should remain shut
        if (ProblemWorldDomains.rp.Options == 1) {
          // the user has selected the exclusion option
          // wait until SecurityFeature.shutUntil has passed
        }
      }
    }
  }
```

The above program also provides a template for the implementation of the exclusion with priority option. For instance, if we want to give priority to the security feature over the temperature feature, there is no need to delay calls to tiltIn, and only calls to tiltOut should be examined for a possible delay.

3.4 Comparing Precedence Operator with Composition Frames

The precedence operator of AspectJ is, in a sense, similar to Composition Frames, in particular to the exclusion with priority option. The similarity lies in the fact that they both provide mechanisms for ordering aspects. There are, however, notable differences.

First, the precedence operator has only one semantic, and the operator is applied at compile-time. Composition Frames provide a multitude of possible semantics, of which we have discussed three in this chapter but there are more [9]. Operators of Composition Frames are applied mostly at runtime, although they can also achieve the effect of compile-time composition. In this sense, Composition Frames can be seen as an extension of the precedence operator.

Second, the precedence operator is applicable only when there is a joinpoint matching with advice from multiple aspects. The pointcuts in our composition operator can be defined on multiple joinpoints. In the smart home example, pointcuts are defined on tiltOut and tiltIn, and they are matched with different aspects for delaying the events involved. In order for the precedence operator to work in the smart home example, tiltOut and tiltIn have to be covered by a single pointcut definition, while providing two aspects for dealing with different events. Such a design is feasible but introduces unnecessary complications. When used judiciously, the precedence operator works well as a simple compile-time operator, whilst Composition Frames provide a richer set of runtime composition operators.

3.5 Fairness in Exclusion

In our implementation of the temperature and security specifications, we have used the Java thread mechanism to design them as long-running and concurrent pro-

cesses. The thread mechanism offers an added advantage when delaying method calls: the thread can simply be put to sleep for a certain duration. For example, the temperature thread wanting to open the window can be put to sleep until the night ends. However, the sleep mechanism of Java cannot guarantee that once the night finishes, the temperature feature will definitely open the window: in fact, it is quite possible that the security feature requests to keep the window shut again before the temperature feature closes it and that the request is successful, thus effectively blocking the temperature feature. If fairness of access is important in the application, then the thread synchronisation facility of Java may have to be used.

4 Common Case Study: Discussion

Our approach is applicable when there are interacting aspects and features and the software system needs to resolve them at runtime in order to continue to satisfy the requirements as far as possible. In the Crisis Management System (CMS) [6], there are several requirements, which under certain runtime conditions will make conflicting demands on the system.

For instance, although it may be possible to satisfy the statistic logging requirement and the real-time requirement individually and their composition most of the time, there may be runtime conditions when it is not possible to log all data access, and provide information about an on-going crisis at intervals not exceeding 30 seconds. In such cases, the users may want to give priority to one requirement over another in order to maintain a satisfactory level of requirement satisfaction.

Likewise, one requirement for multi-access in CMS states that the system should support management of at least 100 crises at a time. Perhaps not all crises are equally important at all times: some could be more important than others in terms of the level of security they require. Again, in such cases, requirements for certain crises may have higher priority over others.

Our approach for resolving interactions between aspects at runtime could be helpful in such cases.

5 Related Work

The ideas presented in this chapter are related to several strands of research work. However, giving a systematic review of all related work is beyond the scope of the chapter. Instead, the following discussions provide a brief overview of some of the work.

Composition Frames: Jackson [5] introduces the conceptual framework of Problem Frames. Laney et al. [8] first use Composition Frames to compose requirements and resolve conflicts before formalizing the composition in [9]. We deploy Composition Frames as a kind of architectural wrapper in order to evolve a feature-

rich software system [15]. This chapter discusses the synergy between Composition Frames and aspect-oriented programming with respect to managing feature interaction and managing aspect interaction. Both approaches are based on the principle of separation of concerns, but offer different ways of composing features and aspects.

Feature Interaction: The problem of feature interactions is a long-standing problem in software engineering. Although they were first observed in telecommunication software systems [2], they are now considered to be a more general problem affecting many modern software systems [1, 7, 4]. Sanen et al. [12] highlight the issue of aspect interaction and contribute a scheme to classify and record aspect interactions. This chapter provides a general mechanism to resolve aspect interactions at runtime.

Aspect Interaction: Mussbacher et al. [10] propose an approach for detecting aspect interactions, in which aspects are first annotated with domain-specific markers. These markers are then mapped to a goal model showing how markers influence each other before conflicting markers and their associated aspects are detected. Other approaches to detecting feature interactions, as well as the difficulties faced by these approaches, are discussed by Velthuijsen [16]. The approach presented in this chapter focuses on resolving, rather than detecting, aspect interactions.

Requirement Interaction: Similar to our approach, Chitchyan et al. [3] consider the problems with using syntactic operators when composing requirements written in a natural language. They propose a new language for documenting textual requirements and their composition. Various formulations of the temporal composition operators in their work are similar to the three semantics of the composition operator given in this chapter. Weston et al. [17] present a formalisation of a similar semantics-based approach to resolving requirements conflicts. However, our approach is aimed at resolving conflicting aspects at runtime, rather than resolving conflicting requirements at design time.

Dynamic Aspect Weaving: There are a number of approaches to weaving aspects at runtime. Popovici et al. [11] suggest that they can be divided into compile-time, load time and runtime approaches, and provide a framework for runtime aspect weaving. Although the problems addressed by their approach and ours are similar, their work requires modification of the Java Virtual Machine in order to load and unload aspects at runtime, and there is a performance penalty every time an aspect is weaved or unweaved. Our implementation uses only the standard AspectJ constructs. While their approach offers a way to resolve aspect interactions by weaving and unweaving aspects at runtime, the variety of composition semantics in our approach is more flexible. For instance, exclusion can be achieved without unweaving and weaving aspects at runtime. However in our approach, aspects have to be known at compile time: their approach does not have this limitation.

Locking Access to Shared Variable: There is a long history of research on controlling access to shared variables by concurrent programs. Hoare-style monitors are a case in point. Typically in such cases, a lock has to be introduced in order to indicate when a given program can or cannot access the shared variable. The variables `openUntil` and `shutUntil` in our example are similar to locks, but these locks cannot be observed, let alone be enforced, by the window. Composition Frames

make use of these locks, together with runtime conditions and user preference to resolve the conflicts. This mechanism provides a neat way to separate concerns of the individual aspects from the concern of their composition.

6 Conclusion

In this chapter, we have described that aspect-oriented software systems that are designed to satisfy multiple requirements may have joinpoints, each of which can be matched with advice from several aspects. In such cases, aspect weavers, such as the one in AspectJ, are free to choose the ordering of aspects. If a particular ordering of aspects is needed, the developer can specify the ordering using the precedence operator, which is used by the weaver to determine the ordering at compile-time. Since the ordering specified by the precedence operator cannot be changed at runtime, the composition of aspects can be over-restrictive, and unresponsive to runtime conditions.

In previous work on detection and resolution of feature interactions, Composition Frames have been proposed and formalized as a way to compose features and resolve feature interactions at runtime. Extending the work, we have now proposed that Composition Frames can be used to compose aspects and resolve aspect interactions at runtime. The proposed approach has been illustrated with an aspect-oriented implementation of a simple example from the smart home application.

In our implementation, the problem world domains are first implemented as classes in the base system. Features are implemented as aspects that access class variables and call methods of classes. When aspects access shared variables, perhaps implicitly through method calls, they indicate the length of time for which they want exclusive access to the shared variables. The length of time can often be derived as part of the feature specifications. Respecting the principle of separation of concerns, aspects do not communicate with each other about their intention for exclusive access. Composition Frames are implemented as distinct aspects that monitor method calls by other aspects and when an interaction is detected, attempt to resolve the interaction. Composition Frames provide a number of semantics by which the aspects can be composed at runtime and in a way responsive to runtime conditions. This gives developers additional mechanisms for composing aspects.

Acknowledgements Feedback from the anonymous review process has helped improve this chapter. This work is partially funded by a Microsoft Software Engineering Innovation Foundation (SEIF) Award, by Science Foundation Ireland grant 10/CE/I1855 and by the European Research Council.

References

1. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Comput. Netw. **41**, 115–141 (2003). DOI 10.1016/S1389-1286(02)00352-3
2. Cameron, E., Griffeth, N., Lin, Y.J., Nilson, M., Schnure, W., Velthuijsen, H.: A feature-interaction benchmark for in and beyond. Communications Magazine, IEEE **31**(3), 64 –69 (1993). DOI 10.1109/35.199613
3. Chitchyan, R., Rashid, A., Rayson, P., Waters, R.: Semantics-based composition for aspect-oriented requirements engineering. In: Proceedings of the 6th international conference on Aspect-oriented software development, pp. 36–48. ACM, NY, USA (2007)
4. Hall, R.J.: Fundamental nonmodularity in electronic mail. Automated Software Engineering **12**(1), 41–79 (2005)
5. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. ACM Press & Addison Wesley (2001)
6. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis management systems: a case study for aspect-oriented modeling. Transactions on aspect-oriented software development VII pp. 1–22 (2010)
7. Kolberg, M., Magill, E.H., Wilson, M.: Compatibility issues between services supporting networked appliances. IEEE Communications Magazine **41**(11), 136–147 (2003)
8. Laney, R., Barroca, L., Jackson, M., Nuseibeh, B.: Composing requirements using problem frames. In: Proceedings of 12th IEEE International Conference Requirements Engineering (RE'04), pp. 122–131. IEEE Computer Society (2004)
9. Laney, R.C., Tun, T.T., Jackson, M., Nuseibeh, B.: Composing features by managing inconsistent requirements. In: L. du Bousquet, J.L. Richier (eds.) ICFI, pp. 129–144. IOS Press (2007)
10. Mussbacher, G., Whittle, J., Amyot, D.: Semantic-based interaction detection in aspect-oriented scenarios. In: RE, pp. 203–212. IEEE Computer Society (2009)
11. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: Proceedings of the 1st international conference on Aspect-oriented software development, AOSD '02, pp. 141–147. ACM, New York, NY, USA (2002)
12. Sanen, F., Truyen, E., Joosen, W., Jackson, A., Nedos, A., Clarke, S., Loughran, N., Rashid, A.: Classifying and documenting aspect interactions. In: Y. Coady, D.H. Lorenz, O. Spinczyk, E. Wohlstadter (eds.) Proceedings of the Fifth AOSD Workshop on Aspect, Components, and Patterns for Infrastructure Software, pp. 23–26 (2006)
13. The AspectJ Team: The AspectJ Programming Guide. Xerox Corporation (2001). URL http://www.eclipse.org/aspectj/doc/next/progguide/index.html
14. Tun, T.T.: Aspect compoistion using composition frames: Java program listings. Tech. Rep. TR2012/09, The Open University (2012)
15. Tun, T.T., Trew, T., Jackson, M., Laney, R.C., Nuseibeh, B.: Specifying features of an evolving software system. Softw., Pract. Exper. **39**(11), 973–1002 (2009)
16. Velthuijsen, H.: Issues of non-monotonicity in feature-interaction detection. In: K.E. Cheng, T. Ohta (eds.) FIW, pp. 31–42. IOS Press (1995)
17. Weston, N., Chitchyan, R., Rashid, A.: A formal approach to semantic composition of aspect-oriented requirements. In: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference, pp. 173–182. IEEE Computer Society, Washington, DC, USA (2008)