Design-level Detection of Interactions in Aspect-Oriented Systems

Pouria Shaker and Dennis K. Peters

Faculty of Engineering and Applied Science,
Memorial University of Newfoundland, St. John’s, NL, Canada, A1B 3X5
{pouria, dpeters}@engr.mun.ca

Abstract. The behaviour of an aspect-oriented (AO) system is the woven behaviour of system concerns (core and aspect). Care must be taken that the woven behaviour of two concerns (possibly compound, i.e. the result of weaving two or more primitive concerns) satisfies both existing critical correctness properties of the behaviour of each individual concern and new desired correctness properties of the woven system. If this set of properties is inconsistent, we say that two or more of the concerns involved undesirably interact. We present a process for detecting undesirable concern interactions in AO systems at the design phase of the software development process, which combines light-weight syntactic analysis and formal verification of design models expressed in the UML and a simple domain-specific statechart weaving language (SWL).

1 Introduction

Separation of concerns (SOC) is the ability to deal with systems one concern at a time. With ideal SOC one can develop, test, and modify system concerns in isolation and evolve systems to handle new concerns without changing existing parts of the system. In 1972, Parnas suggested that this ideal can be approached through the technique of modularization [1]; that is, localizing each concern in a module. Over the years, programming paradigms have emerged to help developers achieve better SOC by providing better modularization mechanisms. The Object-Oriented (OO) paradigm is currently the most popular; its primary unit of modularity, the class, improves SOC by grouping together data and behaviour related to a single concern; however not all concerns of a system can be simultaneously localized in classes. Often in OO systems, concerns related to the primary functionality of the system (core concerns) are localized in classes, while other concerns (cross-cutting concerns) such as logging, caching, security, thread safety, etc. are scattered across several classes. The Aspect-Oriented (AO) paradigm takes another step towards ideal SOC by introducing a new unit of modularity: the aspect. Aspects localize the data and behaviour of cross-cutting concerns and specify points in the structure or execution of the core (join points) where aspect behaviour (advice) applies. A weaving mechanism interleaves the execution of the core with that of the aspects.

By untangling cross-cutting behaviour from core behaviour, the AO paradigm makes it easier to reason about individual concern behaviour. Reasoning about overall system behaviour however, becomes a challenge as it requires examining the woven behaviour of the core and the aspects, which may or may not be explicitly available to the developer in a comprehensible form (this depends on the workings of the weaving mechanism). This situation can give rise to unanticipated anomalies in the behaviour of the woven system. The desired properties of the woven behaviour of two concerns (possibly compound, i.e. the result of weaving two or more primitive concerns) are (1) existing critical correctness properties of the behaviour of each individual concern and (2) new correctness properties of the woven system; if this set of properties is inconsistent, we say that two or more of the concerns involved *undesirably* interact. In (1) we say *critical* correctness properties, to distinguish between *desired* and *undesired* interactions. The very purpose of weaving an additional concern may be to violate existing properties of constituent concerns in favor of achieving new properties for the woven system. In the remainder of the paper the term interaction will be used to mean undesired interaction. Less formally, each concern of a system represents a system goal; if two concerns have conflicting goals, then they interact. Consider an example from [2] involving an operating system core acted on by security and logging aspects. The security aspect encrypts arguments of method calls while the debugging aspect logs method calls for debugging purposes. If logging precedes security, security is compromised by a plain log file; and if security precedes logging, logging is compromised by an encrypted log file that is not very useful for debugging. Other examples of interacting concerns include logging vs. performance, or performance vs. persistence.

The sooner an error is found in the software development process the easier it is to fix. In this paper we present a process for detecting concern interactions at the design stage using both a syntactic analysis of the design model as well as formal verification of behavioural properties.

- We model core and aspect concerns separately using UML class and statechart diagrams and specify rules for weaving core and aspect behaviour in our proposed statechart weaving language (SWL). The SWL offers a rich join point model, support for specifying execution order for the core and aspects at join points as well as advice that can conditionally suppress core execution, and the introduction of new (signal or call) events in the core by aspects.
- We present a static analysis of the design model (UML + SWL) that produces a report listing potential sources of interaction.
- We present two algorithms for weaving core and aspect models into woven UML class and statechart diagrams. The first approach supports all features of the SWL but results in high verification complexity. The second approach lowers verification complexity at the expense of losing support for some SWL features.
- We propose the use of existing UML verification methods (such as [3] [4] [5]) to verify the woven model against behavioural properties (although formal verification of UML models is still a research topic, we view it as an available technology).
- We provide a proof of concept realization of the process where we express UML models in Textual UML Format (UTE) the proprietary input language of Hugo/RT [3]; we provide tool-support for automating the static analysis and weaving of the UTE model based on the SWL specifications; we use Hugo/RT to translate the woven UTE model into a Promela model that can be model-checked with Spin [6].

Our work is dinguished from previous efforts in design-level detection of concern interactions (see Section 6) in that it adopts mainstream design notation (the UML) combined with a simple domain-specific weaving language to minimize the learning curve in producing design models while combining the strategies of previous approaches, namely light-weight syntactic analysis and formal behavioural analysis, in detecting both core/aspect and aspect/aspect interactions.

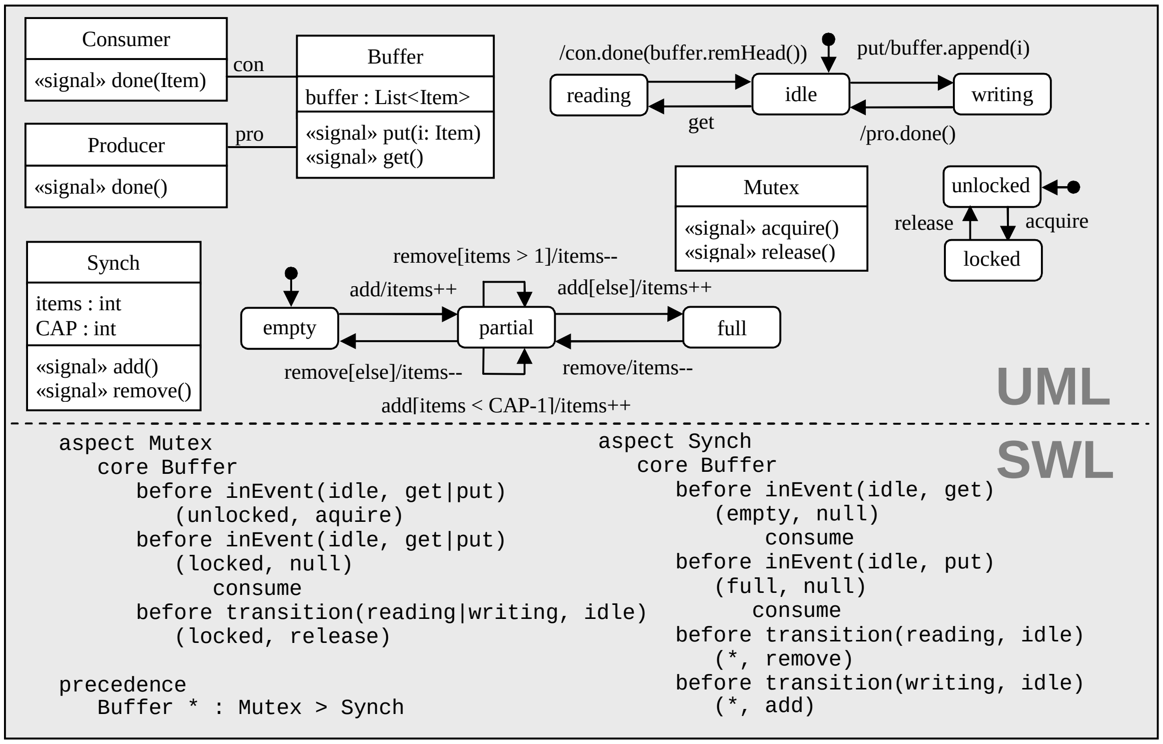
The rest of this paper is organized as follows: Section 2 presents AO modeling with UML and our proposed SWL with a bounded buffer case study (adopted from [7]). Section 3 describes the syntactic analysis on the design model. Section 4 describes our proposed weaving algorithms. Section 5 presents a summary and analysis of related work (additional references to related work have been made in other sections where appropriate). Section 6 presents the conclusion and directions for future work.

2 Aspect-Oriented Modeling Approach

The application of AO technology at the design stage of the software development process is termed aspect-oriented modeling (AOM); for related research see the series of AOM workshops [8]. We sought a modeling approach that makes use of mainstream design notation (such as the UML) and lends itself to formal verification. To this end we adopted an extension/modification of the approach in [9] which we present below. To illustrate our approach we will use the bounded buffer case study from [8]. We consider a FIFO buffer core regulated by synchronization and mutual exclusion aspects; synchronization ensures that get/put requests are blocked when the buffer is empty/full; mutual exclusion ensures that the buffer can be accessed by one client (producer or consumer) at a time. In the first step we model the structure and behaviour of the core and aspects separately using UML class and statechart diagrams (see Figure 1). Note that in Figure 1, objects communicate asynchronously via signal events. The behaviour associated with a signal event is fully specified by the action component of the transition triggered by the signal event; for example, the behaviour associated with the `add` signal event in the `Synch` class is fully specified by the action `item++`.

Next we specify weaving rules in our proposed SWL, the abstract grammar of which is shown in Figure 2. In a weaving rule specification we declare aspects and precedence rules governing aspect ordering. An aspect declaration points to the class that models the aspect and specifies which (core) classes the aspect crosscuts (a class cannot be declared as both an aspect and a core). For each core we can introduce new event receptions for the core statechart and specify advice that applies at join points raised by the core statechart. Supported join point kinds are (note that arguments of the join point kinds, underlined in the grammar of Figure 2, are regular expressions describing sets of state, event, or object names):

- `inEvent‹`*`st, ev`*`›`: corresponds to the run-to-completion processing of an event (whose name is) matched by `ev` by the core statechart when it is in a state matched by `st`.
- `outEvent‹`*`obj.ev`*`›`: corresponds to the execution of an event invocation action for which the event is matched by `ev` and the target object is matched by `obj`.
- `stateEntry/Exit‹`*`st`*`›`: corresponds to the execution of entry/exit actions of a core state matched by `st`.
- `transition‹`*`srcSt, dstSt`*`›`: corresponds to the execution of actions of a transition in the core statechart from a state matched by `srcSt` to a state matched by `dstSt`.



```
aspect Mutex
   core Buffer
      before inEvent(idle, get|put)
         (unlocked, aquire)
      before inEvent(idle, get|put)
         (locked, null)
            consume
      before transition(reading|writing, idle)
         (locked, release)

precedence
   Buffer * : Mutex > Synch
```

```
aspect Synch
   core Buffer
      before inEvent(idle, get)
         (empty, null)
             consume
      before inEvent(idle, put)
         (full, null)
            consume
      before transition(reading, idle)
         (*, remove)
      before transition(writing, idle)
         (*, add)
```

Fig. 1. Bounded buffer UML model and SWL weaving rules specification

The pointcut designators presented in the grammar essentially correspond to AspectJ's context matching pointcut designators (e.g. `call`, `execution`, etc.). Named pointcuts and the disjunctive composition of primitive pointcuts (via `||` operator) can be trivially supported (a disjunctive pointcut is equivalent to copies of the advice for each of the disjuncts); however, support for scoping pointcut designators (e.g. `cflow`) will require changes to the weaving algorithm and is left as future work.

Aspect advice is specified as a forest. A node `‹st, ev(args)|null›` is enabled if the aspect statechart is in a state that belongs to the node's *state-set*, that is the set of states matched by `st`. Execution of a node either introduces event `ev` with arguments `args` in the aspect statechart or does nothing (when the second argument is `null`). Aspects can access contextual data from `inEvent` and `outEvent` join points through their event arguments and use this data in advice by supplying them as arguments to advice node events. Advice execution can be described as a call to `executeAdvice(advice.rootNodes)` (see Figure 3). Note that Figure 3 provides an operational specification for advice execution; implementations for this specification are presented in Section 4. For the execution of advice as described in Figure 3 to be deterministic, we require that no two sibling nodes be enabled at the same time, or in other words, for state-sets of sibling nodes to be disjoint. Alternatively we could allow multiple enabled sibling nodes and always pick the first one. Advice can be specified to execute before or after its join point. Before advice can potentially change event arguments in `inEvent` and `outEvent` join points and can have special `consume` leaf nodes. Consume nodes are always enabled and cannot have siblings; their execution prevents the core or advice of lower precedence from seeing the join point.

```
    WeavingRules → Aspect+ Precedence*
          Aspect → className Core+
            Core → className EventIntroduction+ (BeforeAdvice | AfterAdvice)+
    BeforeAdvice → before JoinPoint (consume | BeforeAdviceNode+)
     AfterAdvice → after JoinPoint AfterAdviceNode+
BeforeAdviceNode → AdviceAction (consume | BeforeAdviceNode*)
 AfterAdviceNode → AdviceAction AfterAdviceNode*
    AdviceAction → ‹AspectStateExp, aspectEvent(Args) | null›
       JoinPoint → inEvent ‹CoreStateExp, CoreEventExp›
                   | outEvent ‹ObjExp.EventExp›
                   | (stateEntry | stateExit) ‹CoreStateExp›
                   | transition ‹CoreSrcStateExp, CoreDstStateExp›
      Precedence → coreName JoinPoint : aspectName (> aspectName)+
```

Fig. 2. SWL abstract grammar (underlined non-terminals are regular expressions describing sets of state, event, or object names)

```
executeAdvice(Set<AdviceNode> nodes)
   if nodes.hasEnabledNode
      nodes.enabledNode.execute();
      if !nodes.enabledNode.isLeaf
         executeAdvice(nodes.enabledNode.childNodes);
```

Fig. 3. Advice execution procedure

Intra-aspect advice precedence is determined by order of declaration. Inter-aspect advice precedence for aspects declared in the same file can be explicitly declared in weaving rule specifications at the resolution of join points; where not specified, order of declaration determines precedence. Inter-aspect advice precedence for aspects declared in separate files is non-deterministic (to make it deterministic, an inter-file precedence specification similar to a build file can be conceived). If more than one advice introduces an event of the same name but with differing argument lists to the core, only the event introduction by the advice of highest precedence takes effect. This is essentially a simple conflict resolution strategy adopted from AspectJ. Alternatively, the conflict can be reported and its resolution left to the developer. Figure 1 shows the weaving rule specification for the bounded buffer example.

Our modeling approach has many similarities to the *event-based AOP* (EAOP) work presented in [2]. In EAOP, aspect behaviour is expressed as a combination of basic rules. A basic rule is made up of a parameterized pointcut and advice that can reference pointcut parameters. A basic rule on its own constitutes a primitive aspect; compound aspects can be formed by the combination operators: *prefixing* (a basic rule to an aspect), *choice* (between two aspects), and *repetition* (of an aspect). Upon occurrence of a join point, the *applicable* basic rule for an aspect (if one exists) is determined as follows: in a prefixed aspect, the first prefix is the applicable basic rule provided its pointcut matches the join point; in a choice between two aspects, the applicable basic rule is that of the first aspect that has an applicable basic rule; in a repetition of an aspect, the applicable rule is that of a single iteration of the repetition. If an applicable basic rule is found, it is applied by executing its advice with pointcut parameters replaced by corresponding values from the join point, and it is removed from the aspect behaviour (for the current iteration of a repetition). The consumption of each join point by an aspect in this manner evolves the aspect from one state to the next. In our approach, aspects are explicitly modeled as finite state machines, and the execution of advice-trees cause state transitions in the aspect model. Structural elements of advice-trees can be mapped to elements of EAOP aspect specifications: advice nodes correspond to basic rules, the sibling relationship between nodes corresponds to a restricted choice operator (where at most one aspect in a choice can have an applicable basic rule for a given join point), the parent-child relationship between nodes corresponds to the prefixing operator, where the parent is the prefix for an aspect prefixed by the child, and finally, advice-trees by definition, are repetitious. In EAOP aspect weaving is expressed by the parallel composition of aspects. Two parallel aspects are said to interact if both have applicable basic rules for some join point. Conflict resolution operators are introduced that apply advice of conflicting basic rules in sequence, or suppress the advice of one basic rule in favor of the other. Our approach provides similar conflict resolution operators: advice precedence imposes a sequential ordering on interacting aspects as defined in [2]; additionally, the execution of consume nodes in advice-trees suppresses advice of aspects of lower precedence (conditional suppression is made possible by having both consume and non-consume leaf nodes in an advice-tree). Finally, in EAOP, the notion of inter-crosscut variables is introduced to allow basic rules of a complex aspect to exchange information. This idea is readily supported in our approach since aspects are modeled structurally as classes with data members that can serve as a medium to pass information within and between advice-trees.

Our modeling approach also has commonalities with composition filters [10] where core behaviour is enhanced by filter modules that can manipulate incoming/outgoing messages to/from the core. A filter has a type and a pattern; the pattern determines which messages the filter accepts/rejects while the type determines actions performed by the filter upon acceptance/rejection of a message (e.g. modifying the message, dispatching the message to the core, etc.). Filters can be composed sequentially. In our approach aspects can be viewed as filters and core join points as messages; advice pointcuts correspond to filter patterns, and the advice-tree specification combined with the aspect state machine constitute the filter type (this gives flexibility in specifying filter actions). Upon acceptance of a message, in addition to performing actions associated with its type, an aspect can conditionally consume the message. A rejected message is passed on to other aspects (if present) or to the message target. Unlike filters however, aspects (as modeled in our approach), in EAOP terms, are *stateful* in that they evolve with the acceptance of messages.

The expressive power of our modeling approach depends on that of its two main components: first, the modeling of core and aspect structure and behaviour; and second, the specification of rules for weaving core and aspect modules (structurally and behaviourally). The expressive power of the first component is essentially that of UML class diagrams for specifying structure, and UML statecharts for specifying behaviour. The expressive power of the second component is that of our SWL, a thorough characterization of which is yet to be performed. Such characterization should include detailed comparisons with existing formalisms as well as a wide range of sophisticated case studies; however, informal comparisons with formalisms such as EAOP [2] and composition filters [10] (as presented above), as well as AspectJ [11] (consider the concepts of event introduction, advice ordering, and advice precedence) provide some hints as to the expressiveness of our SWL.

3 Syntactic Analysis of the Design Model

A syntactic analysis of a design model specified in UML and SWL may reveal overlaps between advice in the same or in different aspects unnoticed by the developer. Such overlaps may be the source of interactions and so it useful to draw attention to them to allow revisions prior to the computationally expensive formal verification of the design. Our analysis performs the following steps:

- Unfold regular expressions in advice and precedence declarations, by analyzing aspect and core statecharts, producing concrete declarations from templates (warnings are generated for regular expressions with no matches).
- Detect and report advice applied to the same join point. In the report, advice are categorized by kind (before or after advice) and ordered by precedence within each category; instances where an advice of higher precedence *may* consume the join point preventing lower precedence advice from seeing it are highlighted; whether this actually happens may depend on run-time data.
- A join point that occurs within another join point will be consumed if its container is consumed. From the definitions of Section 2, it is apparent that `outEvent` join points can occur within `inEvent`, `stateEntry/Exit`, and `transition` join points, and `stateEntry/Exit` and `transition` join points can occur within `inEvent` join points; our analysis detects and reports such instances.

In the bounded buffer example, our analysis will report the following

- Multiple advice on join points `inEvent(idle, get)` and `inEvent(idle, put)` and the possibility of `Mutex` advice consuming the join points preventing `Synch` advice from seeing them.

- Multiple advice on join points `transition(reading, idle)` and `transition(writing, idle)`.

The number of shared join points may become unwieldy in a large application and/or in the presence of systemic aspects (e.g. logging) that may be applied to most/all join points. In such inherently complex cases, the static analysis itself as well of its usefulness to the developer may not scale up. To combat this situation, mechanisms could be envisioned to allow the developer to parameterize the static analysis to report what is deemed to be a significant subset of the set of shared join points. Since the syntactic analysis is performed on the unwoven model, its results are readily traceable to elements of aspect and core models.

4 Weaving Algorithms

We present two algorithms for weaving aspect and core UML models based on an SWL specification. The first approach is as follows:

- *Structural weaving:* for each core class we introduce a proxy class with an identical interface; unidirectional associations ending at the core are moved to the proxy (consider bidirectional associations as two unidirectional associations); the proxy has a bidirectional association with the core and a unidirectional association with all aspects applied to the core.
- *Behavioural weaving:* the proxy statechart handles the same events as the core as well as new events introduced by advice. When the proxy receives event *ev*, for each advice on a join point of form `inEvent(st,` *ev*`)`, it reacts by executing actions that map to the SWL advice in the manner shown in Figure 4; if no such advice exists, the event is simply passed on to the core. The proxy also handles special call events from the core corresponding to before and after state entry/exit, transition, and event invocation actions. When the proxy receives such events, if advice for corresponding join points is specified, advice actions (derived in a manner similar to `inEvent` join points) are executed; join point consumption is conveyed to the core by setting a global flag. This approach relies on synchronous communication between the proxy and core/aspect statecharts to ensure strict order in their interleaved execution; for this reason all signal events in the core and aspect statecharts are made into call events.

```
if(core.inState(cst))
   actions(advice.rootNodes) for highest precedence before advice
   if(!consume)
      actions(advice.rootNodes) for remaining before advice
   if(!consume)
      core.cev
      actions(advice.rootNodes) for all after advice
```

```
actions(Set<AdviceNode> nodes)
   if nodes has just one consume node
      consume = true
   otherwise for every node <ast, aev(args)|null> in nodes
      if(aspect.inState(ast))
         ast.aev(args) | skip
         actions(node.children) if node has children
```

Fig. 4. Actions for advice on join point `inEvent(cst, cev)`

In effect, the proxy is a localized implementation of advice-tree specifications in the language of statecharts: the proxy statechart intercepts incoming, outgoing, and internal events generated by the core and orchestrates the execution of core and aspect models as specified by advice-tree specifications (hence the need for both proxy and core modules). Figure 5 (1) shows the woven class diagram and a skeleton of the proxy statechart for the bounded buffer example; this woven model was expressed in UTE and translated into Promela using Hugo/RT and model-checked for absence of deadlock using Spin. Deadlock occurs in a system of communicating statecharts when each statechart in the system is stuck in a state waiting for an event that can only be triggered by another statechart in the system. Absence of deadlock is property desirable for any system; however, problem-specific properties such as “no writes/reads are possible to a full/empty buffer” and “only one reader/writer can use the buffer at a time” can (and should) also be checked. The verification report showed a high level of complexity in the Promela model (roughly $3x10^6$ states and three minute verification time). In an effort to reduce verification complexity we investigated in alternate weaving approach described below:

- *Structural weaving:* each core class is given unidirectional associations to each aspect class applied to it
- *Behavioural weaving:* for before advice on `inEvent(st, ev)` join points, the core statechart is augmented with additional states and transitions between `st` and the target states of transitions leaving `st` with event `ev` (see Figure 6). Augmenting the core statechart with after advice actions requires computing all possible configurations reached after the run-to-completion (RTC) step of processing `ev`, and appending after advice actions to the state-entry actions of the lowest level state(s – in the presence of and-states) of each such configuration; but which of the low-level states should be augmented? Another complication arises in the presence of guarded null transitions: whether the source `src` of the null transition is part of the post RTC configuration can be determined by a static evaluation of the null transition guard in the state entry actions of `src`; but if the guard makes reference to global variables in the presence of concurrent statecharts, the result of the condition evaluation may be invalidated before the state entry actions of `src` (including after advice actions) begin to execute! Due to such complications it is believed that after advice cannot be supported in this weaving approach.
- Advice for other join point kinds is realized by augmenting their corresponding actions with advice actions derived in a manner similar to the first weaving approach. Again, new events introduced by advice are added to the core and signal events of aspects are made into call events for synchronization purposes.

Here, the implementation of the advice-tree specification (as a statechart) is tangled with that of the core. Figure 5 (2) shows the woven class diagram and the skeleton of the woven core statechart for the bounded buffer example. Model-checking the woven model showed a reduction in the complexity of the Promela model (roughly 3×10^5 states and under one minute verification time).

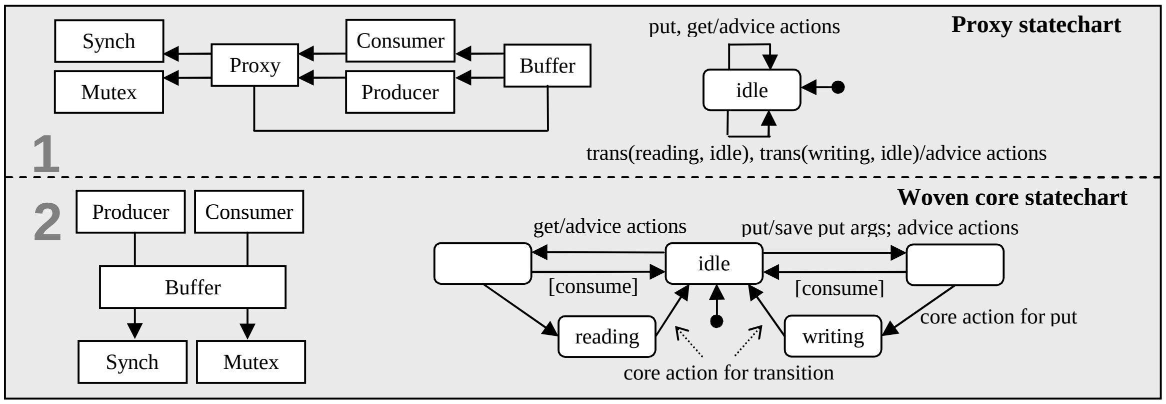
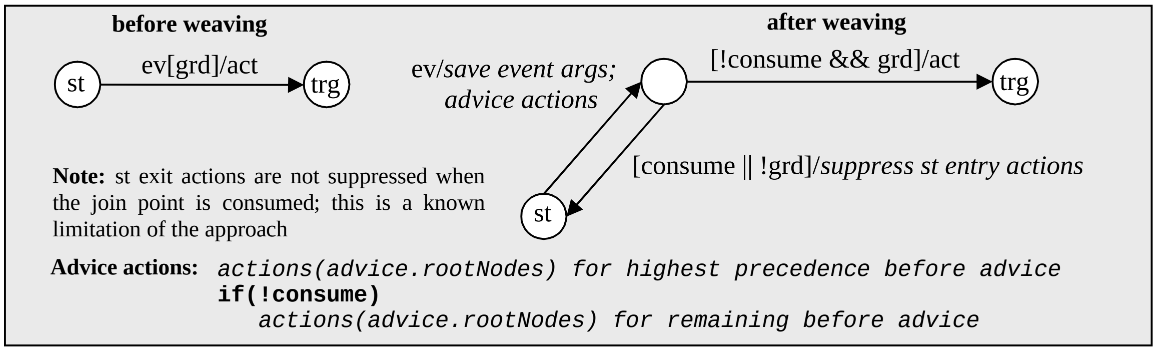


Fig. 5. Woven UML model for bounded buffer example using two weaving algorithms

The verification complexity of a statechart model increases with the number of component statecharts and the complexity of each component statechart (in terms of state and transition count of the flattened statechart, the count and type of data members, and the size of event queues). The first weaving approach adds a new component to the model: the proxy. The complexity of the proxy statechart in terms of number of states and transitions is a function of the number and size of advice-trees. If we model advice-trees with a chain of choice pseudo-states, each advice node adds a choice state and each branch adds a transition to a proxy statechart made up of a single idle state as shown in Figure 5 (1). The application of multiple advice at a given join point adds transitions linking pseudo-state chains of each advice-tree. In the second weaving approach, no new component is added but the additional states and transitions due to advice-trees are added to the core statechart. Additionally, intermediate states for consume advice as well as data members for event arguments and consume flags are added as shown in Figure 6. New states and transitions due to advice-trees have a greater impact on verification complexity in the first approach since they are added to a new component. Additional complexity may be introduced by translations performed by UML verifiers (we have experimented with only one such verifier: Hugo/RT; see results above). To assess the scalability of our approach with larger problems (involving more aspects, more interactions, and more complex advice), we must experiment with other UML verifiers and more sophisticated case studies. In our work, we perform offline design-time analyses; as such, we do not consider run-time performance overheads associated with the weaving approaches.

In terms of traceability of verification results to aspect and core models, both approaches suffer from the fact that verification is performed on the woven model. The first approach however, has the upper hand: the proxy localizes the implementation of advice-trees with statechart elements that are readily traceable to advice-tree specifications as described above. With the help of UML verifiers such as Hugo/RT, verification traces can be mapped to executions of the proxy statechart. With the second approach, advice-tree implementation is tangled with the core statechart hindering traceability. The direct mapping between additions to the core statechart and advice-tree specifications however, provides some relief.


before weaving
st
ev[grd]/act
trg
after weaving
ev/*save event args; advice actions*
[!consume && grd]/act
trg
[consume || !grd]/*suppress st entry actions*
st
Note: st exit actions are not suppressed when the join point is consumed; this is a known limitation of the approach
Advice actions:

```
actions(advice.rootNodes) for highest precedence before advice
if(!consume)
   actions(advice.rootNodes) for remaining before advice
```

Fig. 6. Weaving before advice for join point `inEvent(st, ev)` into core statechart

5 Related Work

A well-known and extensively researched instance of the concern interaction problem appears in telecommunication systems where features applied to base communication services may interact [12]. A popular example in telephony systems is the interaction between the ‘call forward when busy’ and ‘call waiting’ features when applied to the same call service. In the event of an incoming call when the subscriber’s line is busy, the behaviour of one feature compromises the behaviour of the other. There is much to be learned and adopted from research in the feature interactions domain; however, solutions to the concern interaction problem in AO systems (including this work) need to be more general and applicable across a range of domains (e.g. email, telecommunications, etc.). In [13] the feature interaction problem has been studied using AO technology where features are modeled as aspects applied to a base call service core. AspectJ [11] is used to encode the basic call model as a finite-state machine (FSM) and features as aspects that change the FSM. Program slicing is used to extract portions (slices) of the core affected by each aspect and overlapping slices are reported as potential interactions.

Research in concern interactions is harder to come by in AO literature. Perhaps the most explicit treatment of the subject is given in [2] where a formal language for specifying aspects along with support for the detection and resolution of aspect interactions is presented. Here, aspects are said to interact if they advise the same join point (defined as event patterns in the execution trace of the core). Linguistic support is provided to resolve interactions by composing aspect advice. The definition of interactions as presented in [2] is useful yet limited: it does not account for aspect/core interactions or indirect aspect interactions (e.g. aspects advising disjoint join points but using common data); also, it does not differentiate between desired and undesired interactions. These limitations can be addressed by formally verifying desired properties of core, aspect, and woven models before and after weaving. The definition of [2], however, remains useful for drawing the attention of the developer to potential sources of undesired interactions (but unfortunately not all of them, e.g. in the case of indirect interactions). We adopt the benefits of the definition in the syntactic analysis phase of our approach, and address its limitations via the formal verification of aspect, core, and woven models.

In [14], an analysis of AO programs is introduced that classifies interactions between aspect advice and core methods. Advice may directly interact with a method by augmenting, narrowing, or replacing its execution, or may indirectly interact with a method by using object fields also used by the method. The analysis of [14] does not consider aspect/aspect interactions. Some elements of the advice classification of [14] are made explicit in our SWL: for example, after advice and before advice without consume leaf nodes are augmenting advice; before advice with both consume and non-consume leaf nodes is narrowing advice; and before advice with only consume lead nodes is replacement advice. Our approach does not explicitly classify/report indirect interactions; rather, such interactions can be inferred indirectly from the result of the formal verification of the model.

The use of formal specification and verification of AO systems to detect concern interactions has been previously explored. The accuracy of results obtained from this approach will depend on the accuracy of the specification and verified properties. In [15], it is proposed to use program slicing on AspectJ programs and use the slices to construct models that can be analyzed to verify system properties (also, as in [13] disjoint slices imply no interactions). In [16], superimpositions are introduced as collections of generic parameterized aspects with formal specifications of assumed properties of core programs to which they can be applied, and desired properties of the woven program. Superimposition specifications are used to define proof-obligations of the correctness of woven programs and the feasibility of combining superimpositions. In [17], a behavioural interface specification language for AspectJ supporting the formal verification of AO programs is proposed. The work of [15] [16] [17] is intended to operate at the source code level, while our approach operates on design artifacts. Nevertheless, the main ideas of the aforementioned research can be adapted. For example, as in [15], slicing can be applied to woven statecharts to reduce verification complexity. Also, our SWL can be extended to allow aspect specifications to be augmented with assumed properties of the core to which they are applied as well as desired properties of the woven model as in [16] (currently such specifications are made in the context of the particular UML verification tool used).

Past research on the formal verification of AO designs includes the following: In [18], concerns are modeled as roles and weaving as role-merging. The models are formally specified and verified with Alloy. In [19], a run-time manager is proposed for dynamically weaving aspects with a core modeled as a labeled transition system; interactions are detected using run-time model-checking and resolved using adaptive strategies. In [20], techniques for modularly verifying aspect advice (modeled as a state machine) without access to the core are introduced. Our work differs from these efforts in two respects: First, it adopts main-stream design notation (the UML) to model the behaviour of core and aspect modules. Second, it introduces a simple SWL, separating the concern of modeling core and aspect modules from that of specifying how the modules are to be woven; such separation of concerns permits a light-weight analysis of the unwoven model prior to the computationally expensive formal verification of the woven model. In particular, our work differs from [19] in that it is an *offline* analysis: aspect and core models are woven at design time and the formal verification phase of our analysis operates on the woven model. This is in contrast to [19] which provides an *online* analysis, where new aspects can be woven in at run-time and interactions detected and resolved by monitoring the execution of the woven system. A main component of an online interaction detection framework is an adaptive model for code and aspect modules. Modeling the behaviour of the proxy or woven core module (depending on the weaving algorithm applied) with an adaptive statechart would constitute an important step towards evolving our approach to an online analysis.

Research on modular reasoning of AO systems [21][22][23][24] is aimed at eliminating the need for analyzing the entire system to understand the effect of applying an aspect to the core; such an understanding will aid developers in foreseeing and resolving interactions. To promote modular reasoning, *aspect-aware interfaces* for core modules as introduced in [21] can be derived from SWL specifications, and core module specifications can be augmented to include a list of advisable join points as in [22].

6 Conclusion and Future Research

We have presented a process for detecting concern interactions in AO systems at the design phase of the software development process, which combines light-weight syntactic analysis and formal verification of design models expressed in the UML and a simple domain-specific statechart weaving language (SWL). We have experimented with two approaches to weaving UML models based on SWL specifications; one approach supports all features of the SWL but produces woven UML models that result in complex Promela models when translated using Hugo/RT [3]. The second approach yields simpler woven Promela models but does not support after advice in SWL specifications. We are not certain whether the added complexity of the first approach is more largely due to translation anomalies in Hugo/RT or the inherent complexity of woven models produced by the approach. To determine this, other UML verification tools should be used; unfortunately vUML [4] is no longer available for download, but the IF toolset [5] can be experimented with. We have automated the second approach (tailored for use with Hugo/RT) and are trying examples from the feature interaction domain. In the future, the practicality of our process should be assessed against other sophisticated examples.

References

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

[2] Douence R., Fradet P., and Südholt, M. Composition, reuse and interaction analysis of stateful aspects. In: *Proc .of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, ACM, pp. 141-150, 2004.

[3] Schafer T., Knapp A., and Merz S. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[4] Lilius J. and Porres Paltor I. vUML: a tool for verifying UML models. In: *Technical Report 272*, Turku Centre for Computer Science, 1999.

[5] Ober I., Graf S., and Ober I. Validating timed UML models by simulation and verification. In: *Proc. of workshop on Specification and Validation of UML models for Real Time and Embedded Systems, SVERTS, associated with UML 2003*, technical report Verimag, 2003.

[6] Holzmann G. J. The Spin Model Checker -- Primer and Reference Manual. Addison-Wesley, Boston, Massachusetts, USA, 2004.

[7] The Aspect-Oriented Statechart Framework Project. http://www4.carthage.edu/faculty/mahoney//AOP/AOSF/.

[8] AOSD 2005. Aspect-Oriented Modeling, 6th Int'l Workshop, Chicago, Illinois, USA, http://dawis2.icb.uni-due.de/events/AOM_AOSD2005/.

[9] Mahoney M., Bader A., Elrad T., and Aldawud O. Using Aspects to Abstract and Modularize Statecharts. In: *The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*, 2004.

[10] Bergmans L. and Aksit M. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, 44 (10), 51—57, 2001.

[11] Kiczales, G. et al. Getting Started with AspectJ. *Communications of the ACM*, 44(10): 59—65, 2001.

[12] Calder M., Kolberg M., Magill M. et al. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41: 115—141, 2003

[13] Monga M., Beltagui F., and Blair L. Investigating feature interactions by exploiting aspect oriented programming. 2002

[14] Rinard M., Salcianu A., and Bugrara S. A Classification system and analysis for aspect-oriented programs. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2004.

[15] Blair L. and Monga M. Reasoning on AspectJ programmes. In: *Proceedings of Workshop on Aspect-Oriented Software Development*, German Informatics Society, Essen, Germany, pp. 45—50, 2003.

[16] Sihman M., and Katz S. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529—541, 2003.

[17] Zhao J. and Rinard M. Pipa: A behavioural interface specification language for AspectJ. In: *Fundamental Approaches to Software Engineering (FASE)*, Springer, pp. 150—165, 2003.

[18] Nakajima S. and Tamai T. Lightweight Formal Analysis of Aspect-Oriented Models. In: *UML2004 Workshop on Aspect-Oriented Modeling*, 2004.

[19] Pang J. and Blair L. An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Feature. In: *Proceedings 22 International Conference on Distributed Computing Systems Workshops*, IEEE, Los Alamitos, California, pp. 445—450, 2002.

[20] Krishnamurthi S., Fisler K., and Greenberg M. Verifying Aspect Advice Modularly. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, Newport Beach, California, 2004.

[21] Kiczales G. and Mezini M. Aspect-Oriented Programming and Modular Reasoning. In: *Proc. 27th Int'l Conf. Software Eng. (ICSE 05)*, ACM, pp. 49—58, 2005

[22] Aldrich J. Open Modules: Modular Reasoning about Advice. In: *Proc. 2005 European Conf. Object-Oriented Programming (ECOOP 05), LNCS 3586*, Springer, pp. 144—168, 2005.

[23] Sullivan K., Griswold W. G., et al. On the criteria to be used in decomposing systems into aspects. In: *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, ACM, Lisbon, Portugal, pp. 5—9, 2005.

[24] Clifton C. and Leavens G. Observers and assistants: A proposal for modular aspect-oriented reasoning. In: *Proc. FOAL Workshop*, 2002.