A Process for Design-Level Detection of Concern Interactions in Aspect-Oriented Systems

Pouria Shaker and Dennis K. Peters
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's, NL A1B 3X5
Email: {pouria, dpeters}@engr.mun.ca

Abstract— **We present a process for detecting concern interactions in Aspect-Oriented (AO) designs expressed in the UML and our domain specific weaving rule specification language (WRL). The process consists of two tasks: 1) A light-weight syntactic analysis of the AO model to reveal advice overlaps (e.g. instances where multiple advice applies to the same join point) as potential sources of interaction. 2) Verification of desired model properties before and after weaving to confirm/reject findings of task 1 and/or to reveal new interactions. At the heart of task 2 is a weaving process that maps an unwoven Aspect-Oriented model to a behaviourally equivalent woven Object-Oriented model.**

I. Introduction

A. The Aspect-Oriented Paradigm

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.

B. Concern Interactions

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 this paper the term *interaction* will be used to mean undesired interaction. A simple example of concern interactions from [2] is the interaction between logging and encyrption aspects applied to some core system. The encryption aspect encrypts the content of messages passed within the core, while the logging aspect logs the messages for debugging purposes. If logging precedes encryption, encryption is compromised by a plain log file; and if encryption precedes logging, logging is compromised by an encrypted log file that is not very useful for debugging. More sophisticated instances of concern interactions have been identified in various domains such as telephony, email, middleware, multimedia, etc. references to which can be found in [3].

C. Research Objective and Overview

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 (see Fig. 1). The process assumes AO designs expressed in the UML and our domain specific weaving rule specification language (WRL). Here, the data and behaviour of concerns are modeled separately using UML class and statechart diagrams, and rules for weaving concern behaviour are specified in WRL.

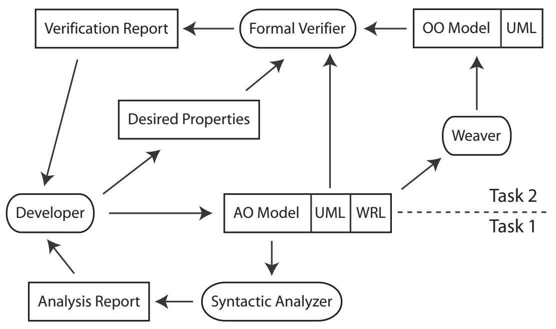


Fig. 1. Design-level concern interaction detection process overview

WRL defines a join point model on UML statecharts and supports the following:

- Before and after advice (before advice can conditionally consume the advised join point)
- Assignment of aspect instances to core instances
- Aspects of aspects
- Aspect composition by a precedence operator on advice

The process consists of two tasks:

- *Task 1:* The AO model is syntactically analyzed to reveal advice overlaps; e.g. instances where multiple advice is applied to the same join point. Such overlaps can be the source of interactions and can easily be overlooked by the developer. Examination of the analysis report by the developer may lead to revisions of the AO model.
- *Task 2:* A weaving process is applied to the AO model the output of which is a woven OO model expressed in the UML. Existing UML verification techniques (such as [4], [5], and [6]) are applied before (on the UML component of the AO model) and after weaving, against desired properties specified by the developer to detect interactions as defined in Section I-B. The verification report may reveal indirect interactions not exposed by task 1, and may be used to determine whether advice overlaps revealed by task 1 do indeed correspond to interactions. It should be noted that formal verification of UML is still a research topic; however, we view it as an available technology.

II. Case Study

To illustrate our process we will use a well-known example from the domain of feature interactions in telephony systems adopted from [7]. Here, the telephony system is comprised of a set of users (telephone receivers), a network switch, and a set of control software modules (one per user). All communication between users and control software modules goes through the switch. In its basic form, a control software module manages a simple connection between its user and another party by communicating with its user and the other party's control software module. In modern telephony systems, users can enhance their control software module by subscribing to various features such as *call forwarding* (CF), which forwards incoming calls to a third party, and *originating call screening* (OCS), which prevents outgoing calls to users on a screening list. In some instances, features fail to co-exist: i.e. features intefere with one another's operation or in other words, they interact. As an example, imagine that user 1 has subscribed to OCS, with a screen on user 3, and that user 2 has subscribed to CF, with all calls forwarded to 3. If 1 calls 2, and the call is forwarded to 3 due to 2's CF, then 1's OCS is compromised, and if the call is not forwarded due to 1's OCS, 2's CF is compromised. Hence the two features interact. In the remainder of this paper we will see how our process can be used to detect this interaction. The case study will be referred to as *FITEL* for *feature interactions in telephony systems*.

III. A Restricted UML

In this section we describe a subset of the UML that is of interest in our process, largely based on [8]. An OO UML model is a set of classes and their initial instantiations. A class has a name, data, and behaviour. Class data is a set of variables called attributes, and class behaviour is a statechart. The initial instantiantion of a class is an instance of that class with initial values for its attributes. A statechart consists of

- A hierarcy of states of types *and* or *or* with rules: 1) the root is an *or* state 2) along any path from the root to a leaf, state types alternate between *and* and *or* 3) leafs are necessarily *and* states. For every *or* state, one child is designated as its initial state. When an *and* state is entered, the initial state of its children are automatically entered.
- A set of events that the statechart can receive, of types *signal* for asynchronous (non-blocking) communication, or *call* for synchronous (blocking - i.e., sender blocks until event processing completes) communication.
- A set of transitions of form $src \xrightarrow{e[g]/act} dst$, where src/dst, e, g, and act are the transition's source/destination state (necessarily of type *and*), trigger, guard, and (optionally *labeled*) action respectively. A transition with trigger $*$ is called a *null transition*.

The active *configuration* σ of a statechart s is the set of states in which it resides (i.e. its active states). The following rules apply: the root is always active; if an *and* state is active, then so are all of its children; if an *or* state is active then so is exactly one of its children; if a state is active, then so are all of its ancestors. The execution state of s is a tuple $\langle \sigma, \nu, q \rangle$ where σ is the active configuration, ν is a map from variables in the scope of s to their values, and q is the event queue of s. The initial execution state is $\langle \sigma_0, \nu_0, q_0 \rangle$ where σ_0 is given by initial states and the above rules on configurations; ν_0 is given by the model's initial instantiation; and $q_0 = \emptyset$. An execution state is *stable* if no state in the active configuration is the source of a null transition and is *transient* otherwise. Events in q are processed one-by-one in FIFO order in a *run to completion* (RTC) step so long as q is not empty. The RTC processing of an event e takes s from one stable execution state to the next, $\langle \sigma_i, \nu_i, q_i \rangle \xrightarrow{e} \langle \sigma_{i+1}, \nu_{i+1}, q_{i+1} \rangle$, by the following process (adopted from [9]):

1) *All enabled transitions are identified:* A transition is enabled if its source state is in σ_i, it is triggered by e, and its guard is true with respect to ν.
2) *Enabled transitions are fired:* Firing a transition $src \xrightarrow{e[g]/act} dst$ causes s to leave src, execute act updating ν_i, and enter dst upating σ_i. Two enabled transitions are in conflict if their source states have an ancestory relation (multiple enabled transitions with the same source state are disallowed). Between conflicting transitions, only the transition whose source state is lowest in the state tree fires.
3) *Null transitions are handled:* If Step 2 lands s in a transient execution state, $*$ is dispatched causing all enabled null transitions to fire as per Step 2. This loop continues until a stable execution state is reached. Intermediate steps that occur within an RTC step are called *microsteps*.

Fig. 2 shows the data and behaviour of FITEL's OO model. Event receptions of all classes excluding CF and OCS are signal events while those of CF and OCS are call events. The initial instantiation of the model is as follows:

os1 : Switch = $\{ cn := ocn, un := oun \}$for n = 1, 2, and 3
ocn : Controln = $\{ x := os1, id := n \}$ for n = 1, 2, and 3
ou1 : User_Caller = $\{ x := os1, id := 1, call := 2 \}$
oun : User_Callee = $\{ x := os1, id := n \}$ for n = 2 and 3
ocf : CF = $\{ x := os1, fw \mapsto 3 \}$
oocs : OCS = $\{ x := os1, screen \mapsto 3 \}$

IV. The Aspect-Oriented Model

In this section we present a language for expressing AO models, which is an extension/modification of that of [10]. We express AO models in UML and WRL. The UML part models data and behaviour for each concern , and the WRL part specifies how concerns cross-cut one another by mapping join points in the behaviour of an instance of one class (the core) to advice of instances of other classes (aspects). The WRL join point model follows:

- *Event join point:* Is a tuple (σ, e) and corresponds to the RTC processing of event e by the core statechart, when it is initially in a configuration σ_i, where $\sigma \subseteq \sigma_i$. For σ to be valid, it must be the subset of *some* configuration of the core.
- *Action join point:* Is a label l and corresponds to the execution of the action labeled l in the core statechart. The WRL join point model can be extended to include join points for specific actions (e.g. event invocation).

A join point can expose contextual data from the core that may be used in advice. The context of event join point (σ, e) is the arguments of e. By default, action join points have no context; however, context can be defined for extensions (e.g. an event invocation join point can expose parameters of the invocation). WRL advice is the aspect statechart's evolution in response to a join point. Several possible evolutions are described as a *tree* of evolution steps (or *advice nodes*), with

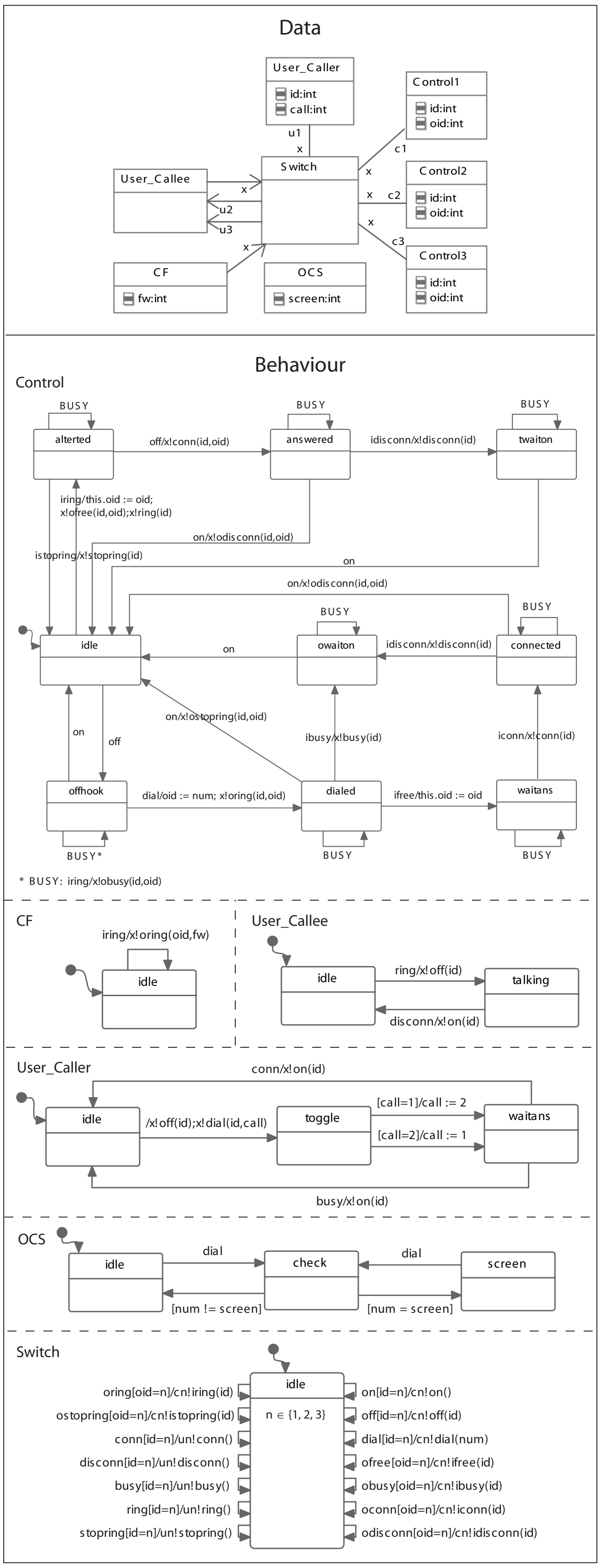


Fig. 2. *FITEL* case study: OO model

each path from the root to a leaf corresponding to one possible evolution. Advice can be specified to apply *before* or *after* the join point. Advice nodes other than the root can take one of two forms:

- *Action node:* Is a tuple (σ, act) where act is either an event invocation action $e(params)$ or `skip`. If $act = e(params)$ the node's action is a single evolution step of the aspect statechart by the execution of $e(params)$, which leads to the RTC processing of event e with arguments $params$ (expressions over the advised join point's context) by the aspect statechart. If $act =$ `skip`, the node's action is to do nothing. The node's action is performed only if the node is *enabled*: i.e., the aspect statechart is initially in a configuration σ_i, where $\sigma \subseteq \sigma_i$. For σ to be valid, it must be the subset of *some* configuration of the aspect.
- *Consume node:* A special node con that does not evolve the aspect statechart; rather it halts advice execution and consumes the advised join point.

The following restrictions apply:

1) Consume nodes must be leaves, cannot have siblings, and can only appear in *before* advice
2) Sibling action nodes cannot be concurrently enabled for any configuration

Upon occurence of the advised join point, the aspect statechart evolves through a sequence of steps described by action nodes of the advice tree along a path that is traced as follows: starting from the root and until a leaf is reached or the path is *blocked*, the path is extended by the enabled child of its tail and the action described by the enabled child is performed (note that consume nodes are always enabled). If the tail has no enabled children, the path is blocked. We allow a class to be both a core (to be advised) and an aspect (to advise), hence the possibility of *aspects of aspects*, with the following restrictions: 1) aspect classes cannot receive *signal* events and 2) two classes may not mutually (transitively) advise one another. The WRL supports basic aspect composition by allowing the odering of a set of advice from one or more aspect(s) on a given join point.

The WRL for *FITEL* is shown in Fig. 3 in an arbitrary syntax, and is explained below:

- *Weaving rule 1:* Before core instance $\texttt{oc1} : \texttt{Control1}$ can process event $\texttt{dial}(\texttt{num} : \texttt{int})$ when it is in state `offhook`, aspect instance $\texttt{oocs} : \texttt{OCS}$ processes the event. If as a result, `oocs`'s statechart lands in state `screen`, the event is consumed (preventing the core from seeing it). Otherwise, `oc1` processes the event as usual.
- *Weaving rule 2:* Before core instance $\texttt{oc2} : \texttt{Control2}$ can process event $\texttt{iring}(\texttt{oid} : \texttt{int})$ regardless of its current state, aspect instance $\texttt{ocf} : \texttt{CF}$ processes the event. Then, unconditionally, the event is consumed.

V. Syntactic Analysis of the AO model

The syntactic analysis of the AO model reveals the following to the developer:

1) When multiple advice applies to the same join point.

```
Aspect OCS
  Core Control1
    before ({offhook},dial)
      ({idle},dial(num))
        ({idle}, skip)
        ({screen}, skip)
          consume
      ({screen},dial(num))
        ({idle}, skip)
        ({screen}, skip)
          consume
    ObjectMap
      oc1 -> oocs
```

Weaving Rule 1

```
Aspect CF
  Core Control2
    before ({root},iring)
      ({idle},iring(oid))
        consume
    ObjectMap
      oc2 -> ocf
```

Weaving Rule 2

Fig. 3. *FITEL* case study: WRL

2) When one advice consumes a join point preventing other advice from executing. This could happen in two cases:
   a) A *before* advice consumes a join point preventing all *before* advice (of lower precedence) and all *after* advice that apply to the same join point from executing.
   b) An advice consumes an *event* join point preventing all advice on *action* join points that may occur *within* the event join point from executing. We say that an action join point $jp_{act} = l$ may occur within an event join point jp_{ev} if the set of transitions T that may be triggered upon occurence of jp_{ev} includes the transition whose action is labeled with l. The computation of T may require a complete exploration of the core statechart. To make the computation feasable, one could specify a maximum depth on the exploration, as the goal is to simply alert the user of *possible* interactions.

As stated in Section I-C, such *advice overlaps* are *potential* sources of *aspect* interaction. However, not all aspect interactions are due to advice overlaps and not all advice overlaps cause aspect interactions. *FITEL* is an example of the former: the syntactic analysis of the *FITEL* AO model reveals nothing, while as we know, interactions do exist in this model. Additionally, advice overlaps do not point to aspect/core interactions.

VI. The Weaving Process

In this section we informally describe the transformation of an AO model to a behaviorally equivalent OO model. We present two versions of the transformation below:

WP1. This version serves two purposes: 1) it gives an operational semantics for WRL in UML and 2) it describes an unoptimized weaving process. The woven OO model produced by WP1 is a modified version of the UML part of the AO model as prescribed by its WRL. The modifications are:

1) Adding one proxy class per core class whose event join points are advised. The purpose of the proxy is to implement advice on event join points of the core. The proxy data are associations to the core and all aspects that advise event join points of the core, and per advised event join point, a flag for the consumption of the join point by some *before* advice. Its statechart has one state

with a set of self transitions, one per core event reception e. If e does not correspond to an advised event join point, its self transition action is a simple invocation of e on the core. Otherwise, if e corresponds to advised event join point (σ, e), its self transition action is:

```
consume := false;
if(core.σ  ⊆  σ)
  /* for each before advice in order of precedence */
  if(!consume)
    advice action
  ...
  if(!consume)
    core.e
    /* for each after advice in order of precedence */
    advice action
    ...
```

The action for advice adv is given by $\mathrm{advAct}(adv.root.children)$ where $\mathrm{advAct}(nodes)$ is:

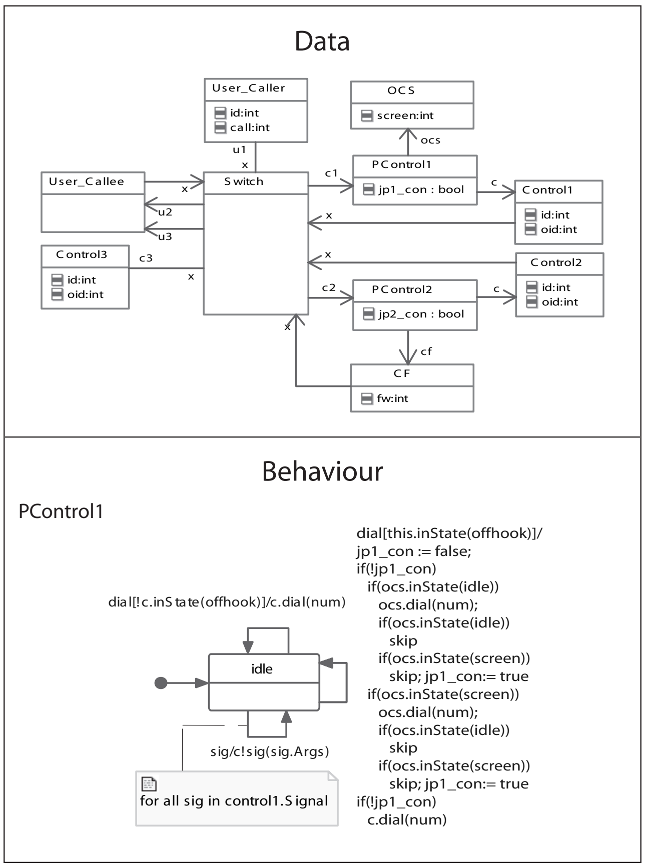
```
/* if nodes has just one consume node  */
consume := true

/* otherwise for each node (σ, act) in nodes*/
if(aspect.σ  ⊆  σ)
  act; advAct(node.children)
```

2) In each core whose event join points are advised, changing signal events corresponding to advised event join points into call events, to enable synchronous communication with the core's proxy.
3) Transferring references to core classes whose event join points are advised to their proxy.
4) Adding to the model's initial instantiation, a proxy instance per instance of core classes whose event join points are advised, and transferring initial references to such core instances to their proxy.
5) For each core whose action join points are advised, adding references to all aspects that advise its action join points and per advised action join point, wrapping its action with advice actions and adding a join point consumption flag.
6) Setting initial aspect references of proxy instances and instances of core classes whose action join points are advised, to the appropriate aspect instance as prescribed by the AO model's WRL.

Fig. 4 shows the data and part of the behaviour of *FITEL*'s woven model using WP1. Here, `PControl1` and `PControl2` are proxy classes for `Control1` and `Control2` respectively. The behaviour of `PControl2` has not been shown for brevity and is derived similar to `PControl1`. The behaviour of all other classes are the same as in the AO model, with the exception that the `dial` and `iring` signal receptions of `Control1` and `Control2` respectively are made into call receptions. The initial instantiation of the woven model becomes:

$\mathtt{os1} : \mathtt{Switch} = \{\ cn := opcn, un := oun\ \}$ for n = 1, 2, and 3
$\mathtt{opc1} : \mathtt{PControl1} = \{\ c := oc1, ocs := oocs, jp1\_con := false\ \}$
$\mathtt{opc2} : \mathtt{PControl2} = \{\ c := oc2, cf := ocf, jp2\_con := false\ \}$
... (other initial instantiations remain the same)

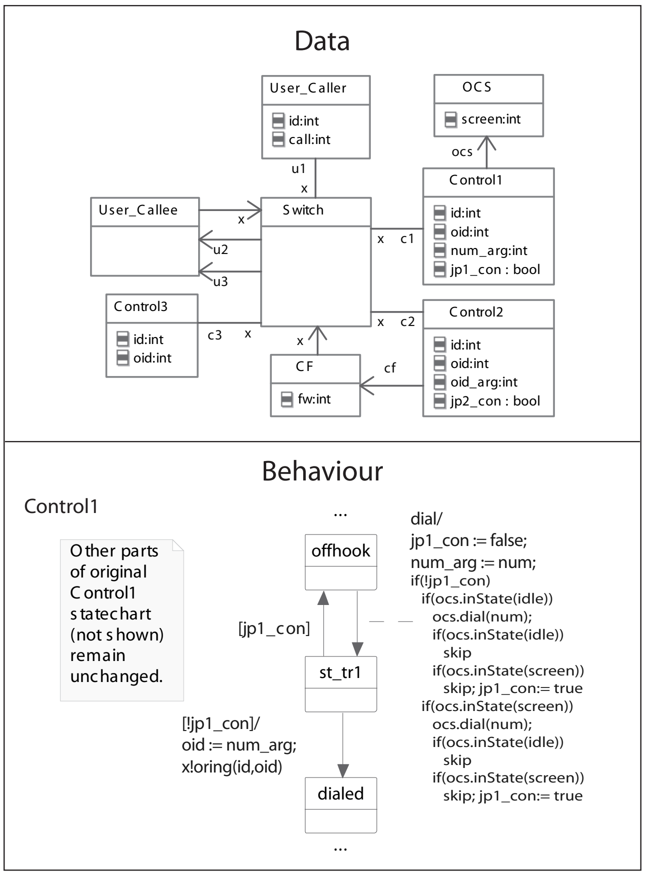


```
dial[this.inState(offhook)]/
jp1_con := false;
if(!jp1_con)
  if(ocs.inState(idle))
    ocs.dial(num);
    if(ocs.inState(idle))
      skip
    if(ocs.inState(screen))
      skip; jp1_con:= true
  if(ocs.inState(screen))
    ocs.dial(num);
    if(ocs.inState(idle))
      skip
    if(ocs.inState(screen))
      skip; jp1_con:= true
if(!jp1_con)
  c.dial(num)
```

Fig. 4. *FITEL* case study: Woven model (WP1)

WP2. This version describes an optimized weaving process whose outcome is a woven OO model better suited to formal verification. The key to the optimization is to move class elements that implement advice on event join points of a core from the core proxy to the core itself, removing the need for proxies. In the absence of proxies, such elements have a lesser impact on the size of a flat finite state automata that simulates the woven OO model. This implies lower verification complexity. Unfortunately, the benefits of the optimization come with a cost: loss of support for *after* advice on event join points, since *after* advice applies after the completion of a join point, and while it is possible to observe when an advised event join point of a core completes from its proxy (this is when the call action that triggers the join point completes), it is not possible to do so from within the core in the presence of concurrency. To accomodate UML verification tools such as [6] that do not support concurrent regions in UML statecharts (i.e. *and* states that have more than one child) we require that statecharts in the UML part of the AO model: 1) do not have concurrent regions, and 2) do not have *conflicting* transitions (see Section III). The *FITEL* AO model and (we believe) many other useful AO models statisfy these conditions. The modifications made by WP2 to the UML part of the AO model as prescribed by its WRL are:

1) For each core whose event join points are advised,



```
dial/
jp1_con := false;
num_arg := num;
if(!jp1_con)
  if(ocs.inState(idle))
    ocs.dial(num);
    if(ocs.inState(idle))
      skip
    if(ocs.inState(screen))
      skip; jp1_con:= true
  if(ocs.inState(screen))
    ocs.dial(num);
    if(ocs.inState(idle))
      skip
    if(ocs.inState(screen))
      skip; jp1_con:= true
```

Fig. 5. *FITEL* case study: Woven model (WP2)

adding references to all aspects that advise its event join points and per advised event join point $jp_{ev} = (\sigma, e)$, adding a consume flag and place-holders for arguments of e, and replacing transitions that may be triggered upon occurence of jp_{ev} with a set of transitions that *delay* jp_{ev} by a microstep. Advice actions are embedded in the one microstep delay. Restriction 1 (on the UML part of the AO model) above, ensures that exactly *one* member of a possible *set* of conflicting transitions can be triggered upon occurence of an advised event join point, and restriction 2 ensures that this set has only one member. So it is possible to precisely determine the transition in the core statechart where advice actions should be embedded.

2) Modification 5 of WP1.
3) Setting initial aspect references of core classes to the appropriate aspect instance as prescribed by the AO model's WRL.

Fig. 5 shows the data and part of the behaviour of *FITEL*'s woven model using WP2. The behaviour of `Control2` has not been shown for brevity and is derived similar to `Control1`. The behaviour of all other classes are the same as in the AO model. The initial instantion of the woven model becomes:

$$\begin{aligned}
&\texttt{oc1 : Control1} = \{ \quad x := os1, id := 1, num\_arg := 0,\\
&\qquad\qquad\qquad\qquad ocs := oocs, jp1\_con := false \}\\
&\texttt{oc2 : Control2} = \{ \quad x := os2, id := 2, oid\_arg := 0,\\
&\qquad\qquad\qquad\qquad cf := ocf, jp2\_con := false \}
\end{aligned}$$

... (other initial instantiations remain the same)

VII. Evaluation

This section evaluates the proposed process by informal analytical arguments concerning the verification complexity of the woven model generated by the weaving processes, the expressiveness of the AO modeling language, and the traceability of results from the static analysis and verification tasks to the AO model; and by empirical results obtained from applying the process to *FITEL*.

a) Verification complexity.: WP1 and WP2 differ both in the *number* of class elements they introduce to implement advice on *event* join points, and in *where* they allocate such elements in the woven model. The increased verification complexity of the woven model compared to the UML part of the unwoven model depends on both the *number* and *allocation* of advice elements (for event join points). The number of advice elements depends on the AO model, while for a given number of advice elements, the impact of the allocation of these elements on the verification complexity of the woven model decreses from WP1 to WP2. The decrease in verification complexity comes at the expense of support for AO model features: From WP1 to WP2 we lose support for *after* advice, concurrent regions, and conflicting transitions.

b) Expressiveness of AO model.: We will assess the expressiveness of the AO modeling language described in Section IV by comparing it to the (in our opinion, expressive) aspect definition language of Event-based AOP (EAOP) [2]. In EAOP, an aspect in its most basic form, is a rule that maps a join point in the *execution trace* of the core to advice. Aspects can be composed by recursion, choice, sequential, and (adapted) parallel composition operators. Compound aspects are state machines that evolve from one consituent aspect to another (based on the composition operators) in response to join points. Aspects themselves can contribute join points to the execution trace; that is, they can be advised by other aspects. In our modeling approach, a basic aspect is a state machine that reacts to join points by executing one or more advice trees (before or after the join point), where each advice tree prescribes one or more evolution steps based on the join point context (and optionally, the consumption of the join point in the case of before advice). Aspects can only be composed sequentially, but at the granularity of advice; that is, order is imposed on advice (within a before/after category) and not on aspects. As explained in Section IV aspects of aspects are also supported in our approach. We believe, there is no fundamental difficulty is changing the weaving processes to support the rich composition operators of EAOP; however, the affect of such support on the verification complexity of the woven model is an important consideration.

	$WP1$		$WP2$	
	$States$	$Prop_{ocs}$	$States$	$Prop_{ocs}$
OCS	≈ 160000	✓	57000	✓
$OCS + CF$	550000+	–	≈ 120000	×

Fig. 6. *FITEL* verification results using IFx

On another note, with static weaving processes such as ours, *per instance* advice (i.e. advice on a particular instance of a core class rather than on all instances of the core) cannot be supported. *FITEL* is an example of where per instance advice is needed: ideally, we would model the basic control software with a single class (say `Control`) and assign features to specific instances of this class. Per instance advice can be (perhaps not attractively) simulated statically by duplicating the core class for each advised core instance. This method has been applied to *FITEL* in Section III by duplicating `Control` per user (`Control1` $- 3$).

c) Traceability: Static analysis is performed on the unwoven model, and as such, its results are readily traceable to elements of the AO model. Formal verification however, is performed on the woven model. Assuming the UML verifier tool presents error scenarios in UML (rather than in a language that the tool translates UML to, e.g. Promela [4] or IF [6] - and both [4] and [6] do so), traceability of error scenarios to the unwoven AO model deteriorates from WP1 to WP2, as class elements that implement advice on *event* join points become less localized (advice elements for *action* join points have the same allocation for all approaches). Regardless the weaving process used, the direct mapping from advice elements to advice trees allows reasonable traceability.

d) Empirical Results: We used IFx [6] to verify the correctness property:

$$Prop_{ocs} = \text{‘No connection from user 1 to user 3 is possible’}$$

of *FITEL*’s woven model as required by the OCS subscription of user 1, once with only *Weaving Rule 1* (only weave OCS), and once with both *Weaving Rule 1 & 2* (weave both OCS and CF) of Fig. 3 on a machine with 2GB of memory. Fig. 6 tabulates the results using weaving processes WP1 and WP2. The column *States* is the size of the state space of an IF model equivalent to *FITEL*’s woven model and is a measure of the woven model’s verification complexity. Note that $Prop_{ocs}$ is satisfied with only OCS woven, but fails to satisfy with both OCS and CF woven: this indicates an interaction between these two features (recall that this was not captured in the syntactic analysis task).

Note from Fig. 6 that the state-space using WP1 (and with both OCS and CF woven) is too large to fit even in 2GB of memory. It appears that with current UML verification technology, WP1 is not feasible for moderately complex models, and should be used only if the model requires *after* advice and is relatively simple. WP2 on the other hand, does appear feasible, and we believe a large set of useful AO models satisfy its restrictions.

VIII. Related Work

Detection and resolution of feature interactions in telephony systems (e.g. *FITEL*) has been an active research area for many years [11]. Using AO technology to *detect* feature interactions has also been studied [12], where AspectJ [13] (a popular AO programming language) is used to encode the control software as a finite state machine (FSM) and features as aspects that change the FSM (or core). Program slicing is used to identify the part (*slice*) of the core changed by each aspect and overlaps in aspect slices are reported as interactions between features encoded by the aspects.

The detection and resolution of concern interactions in generic AO systems is a relatively newer research area. EAOP [2] (whose aspect definition language we described briefly in Section VII) defines *aspect* interaction as aspects advising the same join point (the aspect composition operators serve as linguistic support to resolve interactions). In Section V we used the term *advice overlap* for this definition, and explained how it may fail to capture important interactions in a system (e.g. *FITEL*). We illustrated via *FITEL* how task 2 of our process can detect such interactions. In Section VII we pointed out that our approach falls short of EAOP in linguitic support for interaction resolution, due to less aspect composition operators.

An analysis of AO programs that classifies interactions between aspect advice and core methods is presented in [14] (interactions between aspects are not considered). Advice can interact with a method *directly* by augmenting, narrowing, or replacing its execution, or *indirectly* by using object fields also used by the method. Direct interactions can be found by task 1 of our process: before advice that does not consume and after advice are augmenting advice, before advice that may consume is narrowing advice, and before advice that always consumes is replacement advice. Indirect interactions can be found by task 2.

A classification of aspect interactions into *conflict*, *dependency*, *reinforcement*, and *mutex* interactions and a means of documenting them is presented in [3]. Our definition of concern interaction is closest to the *conflict* category; i.e., interactions due to *semantic inteference* between concerns.

Research on the formal specification and verification of AO systems includes the following. In [15], program slicing is applied to AspectJ programs and slices are used to construct models whose formal verification is feasible. 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 (BISL) for AspectJ that supports the formal verification of AO programs is proposed. While [15], [16], and [17] apply to AO systems at the source code level, the following research targets AO designs. 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 uses a practitioner-friendly AO modeling language made up of a main-stream design language (the UML) and a simple (and intuitive) domain specific language (WRL). Second, the computationally expensive formal verification is preceeded by a light-weight syntactic analysis. Our work particularly differs from [19] in that it is an *offline* process: all tasks (e.g. syntactic analysis, weaving, and formal verification) are performed at *design time*. In contrast, [19] presents an *online* process, where aspects can be woven and interactions detected and resolved at *run time*.

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.

IX. Conclusions and Future Work

We have presented a process for detecting concern interactions in AO designs expressed in UML (for modeling concern data and behaviour) and WRL (a domain specific language for specifying how concerns crosscut). The process consists of two tasks: 1) A syntactic analysis of the unwoven AO model to alert the developer of potential sources of interaction. 2) Verifying properties of the model before and after weaving of concerns to confirm/reject findings of task 1 and/or to reveal new interactions. At the heart of task 2 is a weaving process that maps an unwoven AO model to a behaviourally equivalent woven OO model. We present three weaving processes: WP1 (supports all features of WRL; yeilds a woven model of high verification complexity), WP2 (does not support after advice; verification complexity of woven model is generally lower than WP1), and WP2.1 (does not support after advice, concurrent regions, and conflicting transitions; verification complexity of woven model is generally lower than WP2), the choice of which is driven by required WRL features and the complexity of the AO model. For the (moderately complex) *FITEL* case study, we observed that using IFx [6] for UML verification, WP1 and WP2 are not feasible (due to verification complexity and lack of support for concurrent regions by the verification tool respectively) while WP2.1 is feasible. We propose the following directions for future research:

- Further optimizations to the weaving processes
- Improving expressivity of our AO modeling language by
  - Implementing EAOP [2] composition operators and studying their affect on verification complexity
  - Adding *introductions* (ala AspectJ [13]) to WRL
- Expermenting with more case studies to empirically evaluate the effect of the weaving processes on verification complexity, and the expressivity of our AO modeling language
- Investigating UML verification tools to determine the feasibility of verifying larger models

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications ACM*, pp. 1053–1058, Dec. 1972.

[2] R. Douence, P. Fradet, and M. Südholt, "Composition, reuse and interaction analysis of stateful aspects," in *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, K. Lieberherr, Ed. ACM Press, March 2004, pp. 141–150.

[3] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid, "Classifying and documenting aspect interactions," in *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, Eds. Bonn, Germany: Published as University of Virginia Computer Science Technical Report CS–2006–01, March 2006, pp. 23–26.

[4] T. Schäfer, A. Knapp, and S. Merz, "Model checking UML state machines and collaborations." *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 3, 2001.

[5] I. P. Paltor and J. Lilius, "vUML: A tool for verifying UML models," in *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*, R. J. Hall and E. Tyugu, Eds. IEEE, 1999.

[6] I. Ober, S. Graf, and I. Ober, "Validating timed UML models by simulation and verification," in *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS 2003), a satellite event of UML 2003, San Francisco, October 2003*, October 2003.

[7] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff, "Second feature interaction contest," in *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, M. H. Calder and E. H. Magill, Eds. Amsterdam, Netherlands: IOS Press, May 2000, pp. 293–324.

[8] A. Wasowski, "Flattening statecharts without explosions," in *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM press, 2004, pp. 257–266.

[9] D. Harel and H. Kugler, "The rhapsody semantics of statecharts (or, on the executable core of the UML)," *Lecture notes in computer science*, 2004.

[10] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud, "Using aspects to abstract and modularize statecharts," in *The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*, O. Aldawud, G. Booch, J. Gray, J. Kienzle, D. Stein, M. Kandé, F. Akkawi, and T. Elrad, Eds., October 2004.

[11] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Comput. Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[12] M. Monga, F. Beltagui, and L. Blair, "Investigating feature interactions by exploiting aspect oriented programming," Lancaster University, Lancaster, LA1 4YR, Tech. Rep. comp-002-2003, 2003.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Comm. ACM*, vol. 44, no. 10, pp. 59–65, October 2001.

[14] M. Rinard, A. Salcianu, and S. Bugrara, "A classification system and analysis for interactions in aspect-oriented programs," in *Foundations of Software Engineering (FOSE)*. ACM Press, October 2004.

[15] L. Blair and M. Monga, "Reasoning on AspectJ programmes," in *3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society*, B. Bachmendo, S. Hanenberg, S. Herrmann, and G. Kniesel, Eds., March 2003.

[16] M. Sihman and S. Katz, "Superimpositions and aspect-oriented programming," *The Computer Journal*, vol. 46, no. 5, pp. 529–541, September 2003.

[17] J. Zhao and M. Rinard, "Pipa: A behavioral interface specification language for AspectJ," in *Proc. Fundamental Approaches to Software Engineering (FASE'2003), LNCS 2621*. Springer-Verlag, April 2003, pp. 150–165.

[18] S. Nakajima and T. Tamai, “Weaving in role-based aspect-oriented design models,” in *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004*, A. Moreira, A. Rashid, E. Baniassad, B. Tekinerdoğan, P. Clements, and J. Araújo, Eds., 2004.

[19] J. Pang and L. Blair, “An adaptive run time manager for the dynamic integration and interaction resolution of features,” in *Proc. 2nd Int’l Workshop on Aspect Oriented Programming for Distributed Computing Systems (ICDCS-2002), Vol. 2*, M. Akşit and Z. Choukair, Eds., July 2002.

[20] S. Krishnamurthi, K. Fisler, and M. Greenberg, “Verifying aspect advice modularly,” in *Foundations of Software Engineering (FOSE).* ACM Press, October 2004.

[21] G. Kiczales and M. Mezini, “Aspect-oriented programming and modular reasoning,” in *ICSE ’05: Proceedings of the 27th international conference on Software engineering*. New York: ACM Press, 2005, pp. 49–58.

[22] J. Aldrich, “Open modules: A proposal for modular reasoning in aspect-oriented programming,” in *FOAL: Foundations Of Aspect-Oriented Languages*, C. Clifton, R. Lämmel, and G. T. Leavens, Eds., March 2004, pp. 7–18.

[23] e. a. Sullivan K., Griswold W. G., “On the criteria to be used in decomposing systems into aspects,” in *ESEC/FSE ’05: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 2005, pp. 5–9.

[24] C. Clifton and G. T. Leavens, “Observers and assistants: A proposal for modular aspect-oriented reasoning,” in *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, R. Cytron and G. T. Leavens, Eds., March 2002, pp. 33–44.

[25] *Foundations of Software Engineering (FOSE).* ACM Press, October 2004.